

LA CARA OCULTA DE  
**DELPHI 6**

**Ian Marteens**  
INTUITIVE SIGHT



*Para Maite,  
que sabe oír mi silencio;  
y para la pequeña María,  
que se aferra a mi dedo  
cuando siente que viene el Hombre de Arena*

*Ian ॐ*



# INDICE

<b>PRÓLOGO DEL AUTOR</b>	<b>23</b>
¿UN LIBRO DE TRANSICIÓN?	23
AGRADECIMIENTOS	24
<b><u>PROGRAMACIÓN COM</u></b>	<b><u>25</u></b>
<b>INTERFACES: LA TEORÍA</b>	<b>27</b>
EL ORIGEN DE LOS TIPOS DE INTERFAZ	27
DEMOCRACIA O PEDIGRÍ	29
¿HERENCIA MÚLTIPLE?	31
TIPOS DE INTERFAZ	32
INTERFACES Y CLASES	33
IMPLEMENTACIÓN DE INTERFACES	34
CONFLICTOS Y RESOLUCIÓN DE MÉTODOS	35
LA INTERFAZ MÁS PRIMITIVA	37
LA ASIGNACIÓN POLIMÓRFICA	39
TIEMPO DE VIDA	40
COMPATIBILIDAD AL PIE DE LA LETRA	43
IDENTIFICADORES GLOBALES ÚNICOS	47
INTROSPECCIÓN	49
PREGUNTAS A LA INTERFAZ	51
<b>INTERFACES: EJEMPLOS</b>	<b>53</b>
EXTENSIONES PARA EL ENTORNO DE DESARROLLO	53
¿TE LLAMO O ME LLAMAS?	55
IMPLEMENTANDO LA EXTENSIÓN	58
INTERFACES Y EVENTOS	61
MARCOS E INTERFACES	64
UN REPASO A LAS VMTs	67
REPRESENTACIÓN DE INTERFACES	69
IMPLEMENTACIÓN DE INTERFACES POR DELEGACIÓN	72
IMPLEMENTACIONES DINÁMICAS	74
<b>EL MODELO DE OBJETOS COMPONENTES</b>	<b>77</b>
LOS OBJETIVOS DEL MODELO	77
UN ACUERDO DE MÍNIMOS	78
TRANSPARENCIA DE LA UBICACIÓN	79
CLASES, OBJETOS E INTERFACES	81

CONVENIOS GRÁFICOS	83
CÓMO OBTENER UN OBJETO	84
DETECCIÓN DE ERRORES	85
COM Y EL REGISTRO DE WINDOWS	86
IDENTIFICADORES DE PROGRAMA	87
FÁBRICAS DE CLASES	89
<b>SERVIDORES DENTRO DEL PROCESO</b>	<b>93</b>
TIPOS DE SERVIDORES	93
SERVIDORES DENTRO DEL PROCESO	94
BIBLIOTECAS DE TIPOS	97
EL LENGUAJE DE DESCRIPCIÓN DE INTERFACES	99
LA DIRECTIVA <b>SAFECALL</b>	102
IMPLEMENTACIÓN DE INTERFACES PARA COM	104
REGISTRANDO EL SERVIDOR	107
UN CLIENTE PARA LA CALCULADORA	108
<b>SERVIDORES FUERA DEL PROCESO</b>	<b>111</b>
DIFERENCIAS TÉCNICAS	111
MODELOS DE CREACIÓN DE INSTANCIAS	113
LA TABLA DE OBJETOS ACTIVOS	114
CREAR O RECICLAR	117
APARTAMENTOS	119
COMPATIBILIDAD ENTRE APARTAMENTOS	121
BOMBAS DE MENSAJES	123
EXPERIMENTOS CON LA CONCURRENCIA	124
LA SALVACIÓN ES UNA FÁBRICA DE CLASES	127
EL MODELO LIBRE	129
<b>AUTOMATIZACIÓN OLE</b>	<b>131</b>
¿POR QUÉ EXISTE LA AUTOMATIZACIÓN OLE?	131
LA INTERFAZ <i>IDISPATCH</i>	132
INTERFACES DUALES	134
CONTROLADORES DE AUTOMATIZACIÓN CON VARIANTES	135
PROPIEDADES OLE Y PARÁMETROS POR NOMBRE	137
UN EJEMPLO DE AUTOMATIZACIÓN	138
LA LISTA DE MENSAJES	142
EVENTOS COM	144
CÓMO DELPHI SOPORTA LOS EVENTOS	147
SINCRONIZACIÓN, LISTAS DE OBJETOS Y OTRAS TONTERÍAS	148
IMPORTACIÓN DE BIBLIOTECAS DE TIPOS	150
COMPONENTES PARA SERVIDORES OLE	152
IMPLEMENTANDO EL CLIENTE	155

<b>EL LENGUAJE SQL</b>	<b>157</b>
<b>SISTEMAS DE BASES DE DATOS</b>	<b>159</b>
ACERCA DEL ACCESO TRANSPARENTE A BASES DE DATOS	159
INFORMACIÓN SEMÁNTICA = RESTRICCIONES	162
RESTRICCIONES DE UNICIDAD Y CLAVES PRIMARIAS	163
INTEGRIDAD REFERENCIAL	164
¿QUÉ TIENE DE MALO EL MODELO RELACIONAL?	165
BASES DE DATOS LOCALES Y SERVIDORES SQL	166
CARACTERÍSTICAS GENERALES DE LOS SISTEMAS SQL	168
<b>INTERBASE</b>	<b>171</b>
INSTALACIÓN Y CONFIGURACIÓN	171
LA CONSOLA DE INTERBASE	172
CONEXIONES CON EL SERVIDOR	173
LAS BASES DE DATOS DE INTERBASE	176
INTERACTIVE SQL	177
CREACIÓN DE BASES DE DATOS	178
DIALECTOS	181
TIPOS DE DATOS	182
REPRESENTACIÓN DE DATOS EN INTERBASE	184
CREACIÓN DE TABLAS	185
COLUMNAS CALCULADAS	185
VALORES POR OMISIÓN	186
RESTRICCIONES DE INTEGRIDAD	186
CLAVES PRIMARIAS Y ALTERNATIVAS	188
INTEGRIDAD REFERENCIAL	189
ACCIONES REFERENCIALES	190
NOMBRES PARA LAS RESTRICCIONES	191
DEFINICIÓN Y USO DE DOMINIOS	192
CREACIÓN DE ÍNDICES	192
MODIFICACIÓN DE TABLAS E ÍNDICES	194
CREACIÓN DE VISTAS	195
CREACIÓN DE USUARIOS	195
ASIGNACIÓN DE PRIVILEGIOS	197
PRIVILEGIOS E INTEGRIDAD REFERENCIAL	198
PERFILES	199
UN EJEMPLO COMPLETO DE <i>SCRIPT</i> SQL	200
<b>CONSULTAS Y MODIFICACIONES</b>	<b>203</b>
LA INSTRUCCIÓN SELECT: EL LENGUAJE DE CONSULTAS	203
LA CONDICIÓN DE SELECCIÓN	204
OPERADORES DE CADENAS	205
EL VALOR NULO: ENFRENTÁNDONOS A LO DESCONOCIDO	206

ELIMINACIÓN DE DUPLICADOS	207
PRODUCTOS CARTESIANOS Y ENCuentROS	207
ORDENANDO LOS RESULTADOS	209
SÓLO QUIERO LOS DIEZ PRIMEROS...	210
EL USO DE GRUPOS	212
FUNCIONES DE CONJUNTOS	213
LA CLÁUSULA HAVING	214
EL USO DE SINÓNIMOS PARA TABLAS	215
SUBCONSULTAS: SELECCIÓN ÚNICA	216
SUBCONSULTAS: LOS OPERADORES <i>IN</i> Y <i>EXISTS</i>	217
SUBCONSULTAS CORRELACIONADAS	218
EQUIVALENCIAS DE SUBCONSULTAS	219
ENCUENTROS EXTERNOS	221
LA CURIOSA SINTAXIS DEL ENCUENTRO INTERNO	222
LAS INSTRUCCIONES DE ACTUALIZACIÓN	223
LA SEMÁNTICA DE LA INSTRUCCIÓN UPDATE	224
ACTUALIZACIONES PARCIALES	225
VISTAS	225
<b>PROCEDIMIENTOS ALMACENADOS Y TRIGGERS</b>	<b>229</b>
¿PARA QUÉ USAR PROCEDIMIENTOS ALMACENADOS?	229
CÓMO SE UTILIZA UN PROCEDIMIENTO ALMACENADO	231
EL CARÁCTER DE TERMINACIÓN	231
PROCEDIMIENTOS ALMACENADOS EN INTERBASE	232
PROCEDIMIENTOS QUE DEVUELVEN UN CONJUNTO DE DATOS	236
RECORRIENDO UN CONJUNTO DE DATOS	237
TRIGGERS, O DISPARADORES	239
LAS VARIABLES <i>NEW</i> Y <i>OLD</i>	240
MÁS EJEMPLOS DE <i>TRIGGERS</i>	241
<i>TRIGGERS</i> Y VISTAS	242
GENERADORES	244
EXCEPCIONES	246
ALERTADORES DE EVENTOS	248
FUNCIONES DE USUARIO EN INTERBASE	250
<b>TRANSACCIONES</b>	<b>253</b>
¿POR QUÉ NECESITAMOS TRANSACCIONES?	253
EL ÁCIDO SABOR DE LAS TRANSACCIONES	255
TRANSACCIONES IMPLÍCITAS Y EXPLÍCITAS	256
NIVELES DE AISLAMIENTO DE TRANSACCIONES	258
REGISTROS DE TRANSACCIONES Y BLOQUEOS	261
LECTURAS REPETIBLES MEDIANTE BLOQUEOS	262
VARIACIONES SOBRE EL TEMA DE BLOQUEOS	264
EL JARDÍN DE LOS SENDEROS QUE SE BIFURCAN	265



¿BLOQUEOS O VERSIONES?	268
<b>MICROSOFT SQL SERVER</b>	<b>271</b>
HERRAMIENTAS DE DESARROLLO EN EL CLIENTE	271
UN SERVIDOR, MUCHAS BASES DE DATOS	272
DOS NIVELES DE IDENTIFICACIÓN	274
DOS MODELOS DE SEGURIDAD	275
ADIÓS A LOS DISPOSITIVOS	276
LOS FICHEROS DE UNA BASE DE DATOS	277
PARTICIONES Y EL REGISTRO DE TRANSACCIONES	278
GRUPOS DE FICHEROS	279
VARIOS FICHEROS EN EL MISMO GRUPO	280
ALGUNAS CONSIDERACIONES FÍSICAS	281
TIPOS BÁSICOS	282
TIPOS DE DATOS DEFINIDOS POR EL PROGRAMADOR	283
CREACIÓN DE TABLAS Y ATRIBUTOS DE COLUMNAS	283
TABLAS TEMPORALES	285
INTEGRIDAD REFERENCIAL	286
INDICES	288
PROCEDIMIENTOS ALMACENADOS	288
CURSORES	289
TRIGGERS EN TRANSACT-SQL	291
INTEGRIDAD REFERENCIAL MEDIANTE <i>TRIGGERS</i>	293
TRIGGERS ANIDADOS Y TRIGGERS RECURSIVOS	294
<b>ORACLE</b>	<b>297</b>
SOBREVIVIENDO A SQL*PLUS	297
INSTANCIAS, BASES DE DATOS, USUARIOS	299
TIPOS DE DATOS	300
CREACIÓN DE TABLAS	301
INDICES EN ORACLE	302
ORGANIZACIÓN FÍSICA DE LAS TABLAS	303
PROCEDIMIENTOS ALMACENADOS EN PL/SQL	304
CONSULTAS RECURSIVAS	306
PLANES DE OPTIMIZACIÓN EN ORACLE	307
CURSORES	308
TRIGGERS EN PL/SQL	309
LA INVASIÓN DE LAS TABLAS MUTANTES	310
PAQUETES	312
ACTUALIZACIÓN DE VISTAS MEDIANTE <i>TRIGGERS</i>	314
SECUENCIAS	315
TIPOS DE OBJETOS	317

<b>DB2 UNIVERSAL DATABASE</b>	<b>321</b>
ARQUITECTURA Y PLATAFORMAS	321
AISLAMIENTO DE TRANSACCIONES	322
TIPOS DE DATOS	323
CREACIÓN DE TABLAS Y RESTRICCIONES	324
INDICES	325
TRIGGERS	326
CONSULTAS RECURSIVAS	328
PROCEDIMIENTOS ALMACENADOS	329
 <b>COMPONENTES DE ACCESO A DATOS</b>	 <b>331</b>
 <b>INTERFACES DE ACCESO A BASES DE DATOS</b>	 <b>333</b>
¿NOS BASTA CON SQL?	333
PROGRAMACIÓN PARA TIPOS DUROS	335
NAVEGACIÓN ES MÁS QUE RECUPERACIÓN	337
CON ESTADO, O SIN ÉL	338
LA SOLUCIÓN ¿PERFECTA?	339
LOS CONTRINCANTES	340
SISTEMAS DE ACCESO NATIVOS	341
ODBC: CONECTIVIDAD ABIERTA	341
OLE DB: EL ESPERADO SUCESOR	342
ADO: SENCILLEZ Y POTENCIA	343
BDE: EL MOTOR DE DATOS DE BORLAND	343
DB EXPRESS: DE VUELTA A LOS ORÍGENES	344
 <b>MyBASE: NAVEGACIÓN</b>	 <b>347</b>
JERARQUÍA DE LOS CONJUNTOS DE DATOS	347
CONJUNTOS DE DATOS CLIENTES	349
DATOS PROVENIENTES DE FICHEROS	350
CONEXIÓN CON COMPONENTES VISUALES	353
UNA SENCILLA DEMOSTRACIÓN	355
MyBASE	357
UN EJEMPLO CON ACCIONES	358
APERTURA Y CIERRE	361
LECTURA DE PROPIEDADES DURANTE LA CARGA	362
EL BUFFER DE DATOS Y LA POSICIÓN DEL CURSOR	365
NAVEGACIÓN	366
EL ALFA Y LA OMEGA	367
UN VIAJE DE IDA Y VUELTA	369
ENCAPSULAMIENTO DE LA ITERACIÓN	370
ACCIONES DE NAVEGACIÓN	372
EL ATAQUE DE LOS CLONES	374

TRANSFORMACIONES XML	376
<b>ACCESO A CAMPOS</b>	<b>379</b>
CREACIÓN DE COMPONENTES DE CAMPOS	379
CLASES DE CAMPOS	380
NOMBRE DEL CAMPO Y ETIQUETA DE VISUALIZACIÓN	381
ACCESO A LOS CAMPOS POR MEDIO DE LA TABLA	382
EXTRAYENDO INFORMACIÓN DE LOS CAMPOS	383
LAS MÁSCARAS DE FORMATO Y EDICIÓN	385
LOS EVENTOS DE FORMATO DE CAMPOS	386
LAS PROPIEDADES <i>TEXT</i> Y <i>DISPLAYTEXT</i>	388
CARACTERES CORRECTOS	389
CAMPOS CALCULADOS	390
CAMPOS DE REFERENCIA	391
EL ORDEN DE EVALUACIÓN DE LOS CAMPOS	395
CREACIÓN DE TABLAS PARA MYBASE	395
INFORMACIÓN SOBRE CAMPOS	396
<b>CONTROLES DE DATOS</b>	<b>399</b>
CONTROLES <i>DATA-AWARE</i>	399
ENLACES DE DATOS Y NOTIFICACIONES	401
CREACIÓN DE CONTROLES DE DATOS	401
LOS CUADROS DE EDICIÓN	402
EDITORES DE TEXTO	403
TEXTOS NO EDITABLES	404
COMBOS Y LISTAS CON CONTENIDO FIJO	405
COMBOS Y LISTAS DE BÚSQUEDA	407
ESENCIA Y APARIENCIA	409
CASILLAS DE VERIFICACIÓN Y GRUPOS DE BOTONES	409
IMÁGENES EXTRAÍDAS DE BASES DE DATOS	410
LA TÉCNICA DEL COMPONENTE DEL POBRE	411
PERMITIENDO LAS MODIFICACIONES	412
BLOB, BLOB, BLOB...	414
LA CLASE <i>TBLOBSTREAM</i>	415
<b>REJILLAS Y BARRAS DE NAVEGACIÓN</b>	<b>417</b>
EL USO Y ABUSO DE LAS REJILLAS	417
EL FUNCIONAMIENTO BÁSICO DE UNA REJILLA DE DATOS	418
OPCIONES DE REJILLAS	419
COLUMNAS A LA MEDIDA	420
GUARDAR Y RESTAURAR LOS ANCHOS DE COLUMNAS	422
LISTAS DESPLEGABLES Y BOTONES DE EDICIÓN	424
NÚMEROS VERDES Y NÚMEROS ROJOS	425
MÁS EVENTOS DE REJILLAS	427

LA BARRA DE DESPLAZAMIENTO DE LA REJILLA	428
REJILLAS DE SELECCIÓN MÚLTIPLE	428
BARRAS DE NAVEGACIÓN	430
HABÍA UNA VEZ UN USUARIO TORPE, MUY TORPE...	430
AYUDAS PARA NAVEGAR	431
EL COMPORTAMIENTO DE LA BARRA DE NAVEGACIÓN	431
REJILLAS DE CONTROLES	433
<b>INDICES, FILTROS Y BÚSQUEDA</b>	<b>435</b>
INDICES EN MEMORIA	435
CAMPOS CALCULADOS INTERNOS	437
BÚSQUEDAS	439
FILTROS	441
FILTROS RÁPIDOS	443
FILTROS LATENTES	444
CAMPOS DE ESTADÍSTICAS	446
GRUPOS Y VALORES AGREGADOS	449
<b>RELACIONES MAESTRO/DETALLES</b>	<b>451</b>
REPRESENTACIÓN DE ENTIDADES COMPLEJAS	451
NAVEGACIÓN SOBRE DETALLES	454
USO Y ABUSO DE LA RELACIÓN MAESTRO/DETALLES	457
CONJUNTOS DE DATOS ANIDADOS	459
ANCLANDO LOS DETALLES	462
<b>ACTUALIZACIONES EN MYBASE</b>	<b>465</b>
EL DIAGRAMA DE ESTADOS	465
AUTOEDICIÓN	468
INSERCIÓN	468
MÉTODOS ABREVIADOS	469
ELIMINANDO REGISTROS	470
EL REGISTRO DE ACTUALIZACIONES	471
EL ESTADO DE UNA FILA	472
DISTINTOS GRADOS DE ARREPENTIMIENTO	474
PUNTOS DE CONTROL	475
VALIDACIONES A NIVEL DE CAMPO	476
PROPIEDADES DE VALIDACIÓN EN CAMPOS	477
EVENTOS DE VALIDACIÓN EN CAMPOS	479
VALIDACIONES A NIVEL DE REGISTRO	480
INICIALIZACIÓN DE CAMPOS	482
EL EVENTO DE INICIALIZACIÓN	484

<b>HERENCIA VISUAL Y PROTOTIPOS</b>	<b>487</b>
DECISIONES INICIALES DE DISEÑO	487
INICIANDO EL PROYECTO	488
CREACIÓN Y DESTRUCCIÓN AUTOMÁTICA	489
APERTURA CONTROLADA DE CONJUNTOS DE DATOS	490
ASOCIANDO UNA VENTANA DE EDICIÓN	491
AUTOMATIZANDO LA ENTRADA DE DATOS	493
LA VENTANA PRINCIPAL	494
DE LO SUBLIME A LO CONCRETO	496
 <b>INTERFACES DE ACCESO A SQL</b>	 <b>499</b>
 <b>DB EXPRESS: CONEXIONES</b>	 <b>501</b>
LA ESTRUCTURA DE DB EXPRESS	501
LOS CONTROLADORES	502
FICHEROS DE CONFIGURACIÓN	503
CONEXIONES, EN GENERAL	506
LA CONTABILIDAD DE LAS CONEXIONES	508
EL COMPONENTE <i>TSQLError</i>	509
PARÁMETROS DE CONEXIONES	511
EL ORIGEN DE LOS PARÁMETROS	514
EL DIÁLOGO DE CONEXIÓN	515
UN SENCILLO ALGORITMO DE CODIFICACIÓN	516
EJECUCIÓN DE COMANDOS	519
EJECUCIÓN DE UN <i>SCRIPT</i>	520
MANEJO DE TRANSACCIONES	521
AGÍTESE ANTES DE USAR	524
NUNCA LLUEVE A GUSTO DE TODOS	525
EL MONITOR DE DB EXPRESS	526
 <b>DB EXPRESS: CONJUNTOS DE DATOS</b>	 <b>529</b>
LOS CONJUNTOS DE DATOS EN DB EXPRESS	529
NUEVOS TIPOS DE CAMPOS	533
CARGA DE DATOS EN UN CONTROL DE LISTAS	533
CONSULTAS PARAMÉTRICAS	536
PARÁMETROS REPETIDOS	538
GENERACIÓN EXPLÍCITA DE SENTENCIAS SQL	540
COMPILACIÓN DE CONSULTAS	541
PREPARACIÓN EN DB EXPRESS	542
EL MISTERIO DE LOS METADATOS	544
PARÁMETROS CREADOS AL VUELO	545
PARÁMETROS PARA LA EJECUCIÓN DIRECTA	547
RELACIONES MAESTRO/DETALLES EN DB EXPRESS	548

ORDENACIÓN DE LA TABLA DEPENDIENTE	549
DIAGRAMAS	551
RECORRIDO MAESTRO/DETALLES	552
EXPORTAR UNA FILA	554
CONSULTAS ENLAZADAS	555

## **EL MOTOR DE DATOS DE BORLAND 559**

QUÉ ES, Y CÓMO FUNCIONA	559
EL ADMINISTRADOR DEL MOTOR DE DATOS	560
EL CONCEPTO DE ALIAS	562
PARÁMETROS DEL SISTEMA	562
PARÁMETROS DE LOS CONTROLADORES PARA BD LOCALES	564
BLOQUEOS OPORTUNISTAS	565
PARÁMETROS COMUNES A LOS CONTROLADORES SQL	567
CONFIGURACIÓN DE INTERBASE	569
CONFIGURACIÓN DE ORACLE	570
CONFIGURACIÓN DE MS SQL SERVER Y SYBASE	572
CREACIÓN DE ALIAS PARA BASES DE DATOS LOCALES Y SQL	573

## **ACCESO A DATOS CON EL BDE 575**

LA ARQUITECTURA DE OBJETOS DEL MOTOR DE DATOS	575
SESIONES	576
EL COMPONENTE <i>TDATABASE</i>	577
TABLAS Y CONSULTAS	579
NAVEGACIÓN SOBRE TABLAS SQL	580
BÚSQUEDAS Y FILTROS SOBRE COMPONENTES DE TABLAS	584
LA IMPLEMENTACIÓN DE LAS CONSULTAS	585
EXTENSIONES PARA LOS TIPOS DE OBJETOS DE ORACLE 8	586
CONTROL EXPLÍCITO DE TRANSACCIONES	589
ACTUALIZACIONES EN CACHE	591
CONFIRMACIÓN DE LAS ACTUALIZACIONES	592
EL ESTADO DE ACTUALIZACIÓN	593
UN EJEMPLO INTEGRAL	595
CÓMO ACTUALIZAR CONSULTAS “NO” ACTUALIZABLES	597
EL EVENTO <i>ONUPDATERECORD</i>	599

## **BDE: DESCENSO A LOS ABISMOS 601**

INICIALIZACIÓN Y FINALIZACIÓN DEL BDE	601
EL CONTROL DE ERRORES	603
SESIONES Y CONEXIONES A BASES DE DATOS	604
CREACIÓN DE TABLAS	605
REESTRUCTURACIÓN	608
ELIMINACIÓN FÍSICA DE REGISTROS BORRADOS	610
CURSORES	611

UN EJEMPLO DE ITERACIÓN	613
PROPIEDADES	615
LAS FUNCIONES DE RESPUESTA DEL BDE	616
<b>INTERBASE EXPRESS</b>	<b>619</b>
HISTORIA DEL SUFRIMIENTO HUMANO	619
CONEXIONES EN INTERBASE EXPRESS	620
TRANSACCIONES COMO AMEBAS	622
PARÁMETROS DE TRANSACCIONES	624
CONJUNTOS DE DATOS	625
LA CACHÉ DE INTERBASE EXPRESS	626
EL MONITOR DE IB EXPRESS	627
EL AS BAJO LA MANGA, O CRÓNICAS DEL JUEGO SUCIO	629
CONSULTAS EN INTERBASE EXPRESS	630
EJECUCIÓN DIRECTA DE COMANDOS	632
LOS COMPONENTES DE ADMINISTRACIÓN	634
COPIAS DE SEGURIDAD	636
<b>ADO Y ADO EXPRESS</b>	<b>639</b>
OLE DB Y ADO	639
LAS CLASES DE ADO	640
APLICACIONES BIEN CONECTADAS	641
CADENAS DE CONEXIÓN	643
FICHEROS DE ENLACES A DATOS	644
LA BIBLIOTECA DE TIPOS DE ADO	646
CONEXIONES DESDE JSCRIPT	646
CONJUNTOS DE REGISTROS	648
ADO EXPRESS: SU JERARQUÍA	650
CONEXIONES ADO	651
PROPIEDADES DINÁMICAS	653
EJECUCIÓN DIRECTA	654
EJECUCIÓN ASÍNCRONA	655
TRANSACCIONES EN ADO	656
CONTROL DE ERRORES	657
CONJUNTOS DE DATOS EN ADO EXPRESS	659
UBICACIÓN DEL CURSOR	660
TIPOS DE CURSORES	662
CONTROL DE LA NAVEGACIÓN	663
UNA CONSULTA, VARIAS RESPUESTAS	664
FILTROS, ORDENACIÓN Y RECUPERACIÓN ASÍNCRONA	665
IMPORTACIÓN Y EXPORTACIÓN	667
ACTUALIZACIONES EN LOTE	667

**DATASNAP****671****PROVEEDORES (I)****673**

LA CONEXIÓN BÁSICA	673
DUPLICACIÓN DEL ESQUEMA RELACIONAL	675
ROBO DE INFORMACIÓN	676
COGE EL DINERO Y CORRE	679
PARÁMETROS EN LAS CONSULTAS	680
MI PRIMERA APLICACIÓN EN TRES CAPAS	682
INSTRUCCIONES A LA MEDIDA	684
ABUSOS PELIGROSOS	687
MENTIRAS, MENTIRAS COCHINAS Y ESTADÍSTICAS	688
LECTURA INCREMENTAL	690
LA VARIANTE EXPLICADA POR BORLAND	691
MI VARIANTE PREFERIDA	694
PETICIONES EXPLÍCITAS	697

**PROVEEDORES (II)****701**

SIMBIOSIS	701
DATOS CLIENTES EN DB EXPRESS	703
RELACIONES MAESTRO/DETALLES	705
LA VERDAD SOBRE PERROS Y GATOS	707
LA TEORÍA DE LA LECTURA POR DEMANDA	709
EL PROBLEMA DE LAS CONSULTAS DE DETALLES	710
LOS VALORES MÁS RECIENTES	712
RELECTURA DE REGISTROS	714

**RESOLUCIÓN****717**

GRABACIÓN DE DATOS CON PROVEEDORES	717
UN EJEMPLO MUY SIMPLE DE GRABACIÓN	719
RESOLUCIÓN SQL	721
LA IDENTIFICACIÓN DE LA TABLA BASE	722
CONFIGURACIÓN DE LOS CAMPOS	724
MÁS USOS DE <i>PROVIDERFLAGS</i>	726
ASIGNACIÓN AUTOMÁTICA DE CLAVES	727
TRANSACCIONES Y RESOLUCIÓN	732
ANTES Y DESPUÉS	733
IDENTIDADES EN SQL SERVER	735
TOMANDO LA INICIATIVA	737
ACTUALIZACIONES MAESTRO/DETALLES	738
ALTA DE PEDIDOS	740
MANTENIMIENTO DE LA INTEGRIDAD REFERENCIAL	743
BUSCAR, LISTAR Y EDITAR	744
ERRORES DURANTE LA RESOLUCIÓN	746



CONCILIACIÓN	747
<b>SERVIDORES DE CAPA INTERMEDIA</b>	<b>751</b>
MÓDULOS REMOTOS	751
EL MODELO DE CONCURRENCIA	754
LA INTERFAZ <i>LAppSERVER</i>	755
EXPORTACIÓN DE PROVEEDORES	756
TIPOS DE CONEXIONES	758
CONEXIONES DCOM	759
MARSHALING A TRAVÉS DE ZÓCALOS	759
CONFIGURACIÓN DE <i>TSocketConnection</i>	761
INTERCEPTORES	763
CONEXIONES A TRAVÉS DE INTERNET	764
CONFIGURACIÓN DE <i>TWebConnection</i>	766
LA CACHÉ DE MÓDULOS	767
EXTENDIENDO LA INTERFAZ DEL SERVIDOR	768
UTILIZANDO LA INTERFAZ DEL SERVIDOR	771
ALGUIEN LLAMA A MI PUERTA	772
SEGURIDAD POR MEDIO DE <i>TOKENS</i>	774
CAMBIO DE CONEXIÓN	775
CONEXIONES COMPARTIDAS	776
LAS CONEXIONES LOCALES	779
BALANCE DE CARGA	780
 <b>INTERNET</b>	 <b>785</b>
 <b>EL PROTOCOLO HTTP</b>	 <b>787</b>
PROTOCOLO Y LENGUAJE	787
EL MODELO DE INTERACCIÓN EN LA WEB	788
LOCALIZACIÓN DE RECURSOS	789
VARIOS TIPOS DE PETICIONES	791
ESPIANDO LAS PETICIONES	792
EL COMANDO POST	794
FORMULARIOS	795
LA ACTUALIZACIÓN DE UNA PÁGINA	796
CODIFICACIÓN DE DATOS	798
LAS CABECERAS HTTP	800
EL RESULTADO DE UNA PETICIÓN	800
SERVIDORES HTTP	802
INTERNET INFORMATION SERVER	803
DIRECTORIOS VIRTUALES	805
MÓDULOS ISAPI	807
PERSONAL WEB SERVER	808
WEB APP DEBUGGER	809

<b>INTRODUCCIÓN A HTML</b>	<b>811</b>
APRENDA HTML EN 14 SEGUNDOS	811
ETIQUETAS A GRANEL	812
FORMATO BÁSICO	813
CAMBIO DE ASPECTO FÍSICO O SEMÁNTICO	815
ENLACES	816
TABLAS HTML	818
TRUCOS CON TABLAS	819
FORMULARIOS HTML	821
CONTROLES DE EDICIÓN HTML	822
HOJAS DE ESTILO	824
ATRIBUTOS PARA ESTILOS	826
CLASES PARA ETIQUETAS	827
CLASES ESPECIALES Y OTRAS TRIQUINIUELAS	827
ESTILOS ENLAZADOS Y EN LÍNEA	829
 <b>FUNDAMENTOS DE JAVASCRIPT</b>	 <b>831</b>
LA ESTRUCTURA DE JAVASCRIPT	831
EJECUCIÓN DURANTE LA CARGA	832
EVENTOS HTML	834
DOM ES UNA RUINA	835
VALIDACIÓN DE FORMULARIOS	836
EXPRESIONES REGULARES	837
JUEGO DE ESPEJOS	839
 <b>WEBBROKER</b>	 <b>841</b>
APLICACIONES CGI E ISAPI	841
MÓDULOS WEB	842
EL PROCESO DE CARGA Y RESPUESTA	845
LA CACHÉ DE MÓDULOS	846
ACCIONES	848
GENERADORES DE CONTENIDO	850
ETIQUETAS TRANSPARENTES	852
ETIQUETAS CON PARÁMETROS	853
¿ETIQUETAS ANIDADAS?	854
PROPIEDADES DE UNA PETICIÓN	855
RECUPERACIÓN DE PARÁMETROS	857
GENERADORES DE TABLAS	858
NO SÓLO DE HTML VIVE LA WEB	859
REDIRECCIÓN	861
 <b>MANTENIMIENTO DEL ESTADO</b>	 <b>863</b>
INFORMACIÓN SOBRE EL ESTADO	863

UN SIMPLE NAVEGADOR	865
¿LE APETECE UNA GALLETA?	869
NAVEGACIÓN POR GRUPOS DE REGISTROS	870
APLICACIONES BASADAS EN CONEXIONES	875
<b>PÁGINAS ACTIVAS EN EL SERVIDOR</b>	<b>881</b>
ASP: PÁGINAS ACTIVAS EN EL SERVIDOR	881
LOS OBJETOS GLOBALES DE ASP	883
ASP Y OBJETOS ACTIVEX	884
MANTENIMIENTO DEL ESTADO	886
CREACIÓN DE OBJETOS PARA ASP EN DELPHI	887
UN EJEMPLO ELEMENTAL	889
ACTIVEFORMS: FORMULARIOS EN LA WEB	891
INTERNET EXPRESS: POR QUÉ NO	894
<b>WEBSNAP: CONCEPTOS BÁSICOS</b>	<b>899</b>
¿EN QUÉ CONSISTE WEBSNAP?	899
LOS COMPONENTES DE WEBSNAP	900
EL ASISTENTE PARA APLICACIONES	901
MÓDULOS DE PÁGINAS Y MÓDULOS DE DATOS	903
GENERACIÓN DE CONTENIDO	905
PRODUCTORES DE PÁGINAS CON SUPERPODERES	907
SCRIPTS EN WEBSNAP	908
ADAPTADORES	910
CAMPOS Y ACCIONES DE ADAPTADORES	911
ACCIONES Y FORMULARIOS	914
INTERPRETACIÓN DEL CONCEPTO DE ACCIÓN	916
ACCIONES, BOTONES Y REDIRECCIÓN	918
PRODUCCIÓN ORIENTADA A ADAPTADORES	920
LISTAS DE ERRORES	923
<b>WEBSNAP: CONJUNTOS DE DATOS</b>	<b>927</b>
ADAPTADORES PARA CONJUNTOS DE DATOS	927
REJILLAS EN HTML	930
PAGINACIÓN	931
BÚSQUEDA Y PRESENTACIÓN	933
MODOS Y EDICIÓN	935
SESIONES	937
VARIABLES DE SESIÓN	939
UNA TIENDA CON WEBSNAP	939
AÑADIR A LA CESTA	941
USUARIOS	944
PERMISOS DE ACCESO Y ACCIONES	945
VERIFICACIÓN DE ACCESO A PÁGINAS	946

VERIFICACIÓN SOBRE UNA BASE DE DATOS	948
LA PLANTILLA ESTÁNDAR	950

## **SERVICIOS WEB 953**

SIMPLE OBJECT ACCESS PROTOCOL	953
CLIENTES SOAP	955
INTERFACES INVOCABLES	957
SERVIDORES SOAP	959
CREACIÓN DE INTERFACES PARA SOAP	960
IMPLEMENTACIÓN DE LA INTERFAZ REMOTA	961
PONIENDO A PRUEBA EL SERVICIO	965
NUEVAS CLASES TRANSMISIBLES	966
MÓDULOS REMOTOS JABONOSOS	967

## **LEFTOVERTURE 969**

### **IMPRESIÓN DE INFORMES CON QUICKREPORT 971**

LA FILOSOFÍA DEL PRODUCTO	971
PLANTILLAS Y EXPERTOS PARA QUICKREPORT	972
EL CORAZÓN DE UN INFORME	973
LAS BANDAS	975
EL EVENTO <i>BEFOREPRINT</i>	976
COMPONENTES DE IMPRESIÓN	977
EL EVALUADOR DE EXPRESIONES	978
UTILIZANDO GRUPOS	980
ELIMINANDO DUPLICADOS	982
INFORMES <i>MASTER/DETAIL</i>	983
INFORMES COMPUESTOS	984
PREVISUALIZACIÓN A LA MEDIDA	985
LISTADOS AL VUELO	987
FILTROS DE EXPORTACIÓN	989
ENVIANDO CÓDIGOS BINARIOS A UNA IMPRESORA	990

### **GRÁFICOS 993**

GRÁFICOS Y BIORRITMOS	993
EL COMPONENTE <i>TDBCHART</i>	997
COMPONENTES NO VISUALES DE <i>DECISION CUBE</i>	999
REJILLAS Y GRÁFICOS DE DECISIÓN	1000
USO Y ABUSO DE <i>DECISION CUBE</i>	1002
MODIFICANDO EL MAPA DE DIMENSIONES	1003
TRANSMITIENDO UN GRÁFICO POR INTERNET	1004

ANTICLIMAX	1007
INDICE ALFABETICO	1009



# Prólogo del Autor

**A**CABO DE REGRESAR DE LA librería. Quería comprar algo sobre sistemas operativos, pero he terminado gastándome el dinero en varios libros de “divulgación científica”: uno sobre dinosaurios, otro sobre cosmología, un tercero sobre algo que parece psicología... Y me he puesto a pensar: yo no soy loquero, no me gano la vida detrás de un telescopio, y si viera un dinosaurio vivo me iría a corriendo a la consulta del loquero. ¿Qué demonios hago gastándome la pasta en estas cosas? Pero, como me creo una persona práctica, la pregunta se ha transformado en: ¿por qué no escribo yo mismo libros de divulgación científica, que traten sobre Informática? Acto seguido, me he levantado del sofá y me he puesto frente a una de las estanterías donde se cubren de polvo mis libros de Informática, y ha sido entonces que me he dado cuenta ... de que la mayoría de ellos son precisamente eso: libros de divulgación.

Delphi lleva ya seis versiones de existencia, y los libros que tenían quinientas páginas en la primera versión se han convertido en monstruos de mil quinientas o dos mil páginas. Con pocas excepciones, además, todo ese volumen no da sino para tratar superficialmente los temas que más de moda estén en el momento. Por eso he querido que La Cara Oculta sea un libro especializado, dentro de lo posible. Aquí usted encontrará información, trucos y sugerencias relacionadas con la programación en Delphi de aplicaciones para bases de datos. Sólo eso, y creo que es bastante.

## ¿Un libro de transición?

A pesar de las buenas intenciones la práctica actual de la programación se ha vuelto a complicar bastante. Digo “vuelto” porque hemos conocido una etapa similar, antes de la aparición de Visual Basic, Delphi y los entornos de programación RAD. Recuerdo el alivio que significó para mí el poder abandonar C++ por un lenguaje con la misma potencia, pero infinitamente más fácil de utilizar. Bien, se ha completado el círculo: hay que atesorar demasiados conocimientos para poder escribir aplicaciones con éxito hoy día. Quizás se esté acercando otra revolución informática...

Delphi 6 ha introducido algunos cambios importantes, al estimular el uso de DB Express y DataSnap para el acceso a datos. Hace tres años, un buen libro sobre bases de datos en Delphi debía incluir, obligatoriamente, un capítulo sobre el API de bajo nivel del BDE. Ahora ya no es necesario. Sin embargo, algunas de las nuevas técnicas necesitan madurar un poco. Le advierto que encontrará a lo largo del libro alguna fea costura y varias cicatrices, en los lugares en que he tenido que realizar injertos y trasplantes. Le pido perdón por adelantado; es lo que sucede cuando lo nuevo está naciendo, y lo viejo no se ha ido aún por completo.

También he tenido que dejar material importante fuera de esta edición. No pude incluir el capítulo sobre Internet Express que ya estaba redactado, porque las aplicaciones de ejemplo que funcionaban perfectamente en Delphi 5 dejaron de hacerlo repentinamente. Me hubiera gustado dedicar más espacio a COM y COM+, pero el libro se habría transformado en otro muy distinto. Finalmente, he vuelto a ignorar CORBA; con los vientos que soplan, no creo que sea una omisión significativa.

## **Agradecimientos**

No es retórica barata: mi deuda principal es con mis lectores. Quiero agradecer la paciencia de todos los que han esperado por esta edición. Cada vez más, soy yo quien recibo mensajes con trucos, información práctica y sugerencias. Mi otra fuente importante de inspiración es The Delphi Magazine, la mejor revista de este pequeño mundo délfico. ¡Que vengan muchos años más de TDM, Chris!

Quiero agradecer también a Borland, y en particular a David Intersimone, que me hicieron llegar a toda prisa la versión 6.5 de InterBase cuando recién comenzaba a anunciarse en Estados Unidos. Y a Tony Hong, de XMethods, por permitirme hacer referencia a los servicios Web creados por su empresa. Debo dar las gracias también a Jimmy Tharpe, de Used Disks, por su disposición a echar una mano en lo de WebSnap.

Por último, debo reconocer que este libro no habría existido, como no existió una cara oculta para Delphi 5, de no haber sido por la fe y el apoyo incansable de Maite y Mari Carmen... no encuentro palabras para expresar lo que les debo. Y eso que acabo de escribir un libro de mil páginas.

En fin, que comience el espectáculo.

Ian Marteens  
[www.marteens.com](http://www.marteens.com)  
Madrid, 2002 AD





## **Programación COM**

---

- **Interfaces: la teoría**
- **Interfaces: ejemplos**
- **El Modelo de Objetos Componentes**
- **Servidores dentro del proceso**
- **Servidores fuera del proceso**
- **Automatización OLE**

# Parte



# Interfaces: la teoría

**L**OS TIPOS DE INTERFACES SON ACTUALMENTE el recurso de programación peor comprendido por la mayoría de los desarrolladores. No en lo que respecta a la teoría alrededor de ellos, porque en eso han ayudado los incontables libros de Java escritos al final del pasado siglo, sino en su aplicación en el trabajo cotidiano. Parte de la culpa está en que para muchas personas las interfaces van indisolublemente ligadas a la tecnología COM. Si nuestras aplicaciones no necesitan COM, entonces, ¿para qué perder el tiempo con estas tontas interfaces?

Creo que no es necesario aclararle que tal actitud es errónea. A pesar de que las interfaces en Delphi aparecieron ligadas a COM, se han convertido en un arma importante del arsenal de la Programación Orientada a Objetos. Y este capítulo intentará justificar la afirmación anterior.

## El origen de los tipos de interfaz

Uno de los ideales de la programación es lo que suelo llamar la *metáfora electrónica*: la posibilidad de crear una aplicación a partir de pequeños módulos prefabricados y versátiles, del mismo modo que un aparato electrónico se monta conectando piezas de diferentes fabricantes. Para poder utilizar cada uno de estos módulos, debería ser necesario conocer sólo la función de sus patillas de conexión, pero no su arquitectura interna.

La Programación Orientada a Objetos, tal como se perfiló en la década de los 80s, fue el primer intento serio y exitoso de plasmar esa visión. El papel de los componentes electrónicos lo desempeñaban las *clases*. Gracias a la *encapsulación*, se podían esconder los detalles de la implementación. La *herencia* era un mecanismo que permitía ampliar las posibilidades de un módulo de software sin necesidad de comenzar desde cero. Y el *polimorfismo* garantizaba que, hasta cierto punto, pudiésemos enchufar en un determinado zócalo cualquier pieza que tuviese el número de patillas adecuado. Solamente hasta cierto punto...

### RAPID APPLICATION DEVELOPMENT (RAD)

Un paso importante se dio en los 90s, al surgir primero Visual Basic y luego Delphi. Siguiendo la explicación electrónica, podríamos imaginar que, antes de la aparición de estos entornos de programación, para ensamblar cada nueva pieza dentro de un apa-

rato había que soldar laboriosamente cada conexión. O dicho en términos informáticos: la inicialización de los objetos y sus relaciones dentro de una aplicación consumía una cantidad desproporcionada de código. Los entornos RAD, gracias a sus potentes técnicas de serialización y persistencia, permitieron que este ensamblaje se efectuase a golpe de ratón, de una manera más “visual”; aportaron el equivalente de las placas donde se colocan los circuitos integrados. Por otro lado, Java es un ejemplo de lenguaje para el que ha sido extraordinariamente difícil lograr esta forma de programación.

Pero las clases tienen un problema importante que, aunque fue identificado por algunos trabajos teóricos, escapó durante mucho tiempo al área de atención de los diseñadores de lenguajes prácticos. Tomemos como punto de partida el siguiente fragmento extraído de la declaración de una clase real de Delphi:

```
type
  TStrings = class(TPersistent)
  public
    function Add(const S: string): Integer;
    procedure Delete(Index: Integer);
    function IndexOf(const S: string): Integer;
    property Strings[Index: Integer]: string; default;
  end;
```

Esta declaración nos dice que en el extremo de un puntero de tipo *TStrings* encontraremos “algo” que puede ejecutar alegremente un *Add*, un *Delete* y un *IndexOf* y que si decimos un número, la propiedad *Strings* nos puede devolver la cadena que está en esa posición. Aparentemente, no puede haber especificación más abstracta para una lista de cadenas de caracteres.

Me va a permitir que utilice una analogía que funciona excelentemente con muchos recursos de la programación: cuando usted tiene en su mano una variable de tipo *TStrings*, se establece un contrato entre usted y el objeto que se encuentra al final del puntero. No es un pacto con el diablo, ni mucho menos: usted obtiene todos los beneficios... siempre que sepa qué puede pedirle al objeto. Si pide algo que el objeto no puede satisfacer, y si el compilador no protesta antes, es posible que termine en el Infierno de los Violadores de Acceso.

Lamentablemente, nos han colado varias cláusulas ocultas dentro del contrato. ¿Recuerda que le dije que solamente estaba mostrando un fragmento de la declaración de la clase? No lo decía solamente porque faltasen métodos y propiedades públicos. Incluyo otro fragmento de la declaración:

```
type
  TStrings = class(TPersistent)
  private
    FDefined: TStringsDefined;
    FDelimiter: Char;
    FQuoteChar: Char;
    FUpdateCount: Integer;
    FAdapter: IStringsAdapter;
    // ...
  protected
```

```

function Get(Index: Integer): string; virtual; abstract;
function GetCapacity: Integer; virtual;
function GetCount: Integer; virtual; abstract;
// ...
end;

```

Hay un par de trampas que nos acechan: los atributos privados de la clase y las funciones virtuales no públicas. Usted podrá exclamar: ¡claro, pero son recursos ocultos, y no afectan al código fuente!, que es lo mismo que han venido argumentando los teóricos durante mucho tiempo. Eso es cierto, pero también es cierto lo siguiente: que al incluir estas funciones y atributos internos estamos imponiendo *restricciones* en la implementación física de los objetos compatibles polimórficamente con la clase *TStrings*:

- 1 Cualquier objeto que podamos asignar a un puntero de tipo *TStrings* debe tener los campos *FDefined*, *FDelimiter* y demás en la misma posición en que los ha definido la clase *TStrings*.
- 2 Si utilizamos tablas de métodos virtuales (VMT) para implementar nuestras clases, como hace Delphi, estamos exigiendo también que ciertas celdas de esa tabla alberguen obligatoriamente punteros a las funciones virtuales internas.

#### UN POCO DE JERGA TEORICA

Los teóricos suelen explicar este problema diciendo que al heredar de una clase en la programación orientada a objetos “tradicional” estamos heredando el *tipo* de la clase ancestro, que viene determinado por sus métodos y atributos, a la vez que la *implementación* de la misma. Sin embargo, para que funcionase el *polimorfismo*, al menos en teoría, solamente debería bastarnos heredar el tipo.

## Democracia o pedigrí

Estas limitaciones, aparentemente nimias, traen como consecuencia que todos los objetos compatibles con *TStrings* deben compartir parte de la estructura física de esta clase. En la programación orientada a objetos que conocemos, esto se logra exigiendo la consabida regla de la compatibilidad polimórfica:

- A una variable de tipo *TStrings* solamente podemos asignarle objetos pertenecientes a la propia clase (imposible en este caso, por tratarse de una clase abstracta) o pertenecientes a una clase derivada por herencia de la misma.

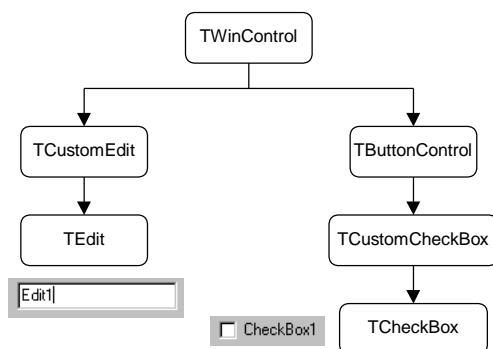
¿Le parece normal? No lo es, sin embargo, a no ser por la fuerza de la costumbre. Y voy a explicarlo con un ejemplo extraído del mundo “real”:

Supongamos que usted necesita los servicios de un albañil. Un albañil es un *objeto* que nos hace un presupuesto, perpetra una chapuza, deja todo sucio y finalmente pasa una elevada factura. Cada una de las cuatro operaciones mencionadas corresponde a un método aplicable sobre el objeto. Respóndame ahora: ¿exigiría al aspirante a alba-

ñil que descienda de una antigua y acreditada familia de albañiles, o sencillamente que sepa hacer bien su trabajo? Quizás en la Edad Media no tendríamos más opción que la primera, pero en el mundo moderno solamente se exige la segunda.

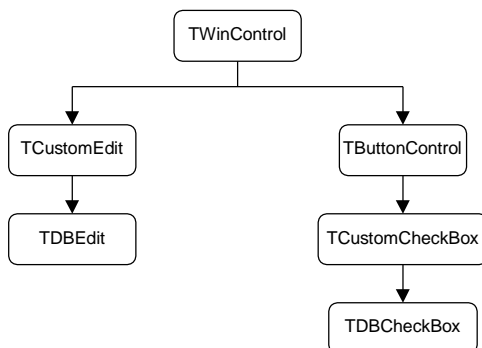
Pero esa condición absurda sobre los antepasados es lo que exige la programación tradicional. Cuando pedimos un objeto que tenga las capacidades propias de una lista de cadenas, estamos también exigiendo que sus ancestros desciendan de la clase base *TStrings*. Pero en Delphi, y en Java también, dicho sea de paso, las clases pueden tener un único ancestro, porque solamente admiten la herencia simple.

El conflicto resultante se ve claramente en Delphi con los controles asociados a datos (*data aware controls*). Observe el siguiente fragmento de la jerarquía de clases de Delphi:



En el diagrama se muestra el árbol genealógico de los componentes *TEdit*, un editor de textos de una sola línea, y de *TCheckBox*, una casilla de verificación. El antepasado común más reciente de ambos es la clase *TWinControl*, que contiene la especificación abstracta de todos los controles visuales de Windows.

Veamos ahora como Borland dotó tiempo atrás, en la era de Delphi 1, a estos componentes con posibilidades de acceso a bases de datos:



Obviemos el detalle de que se utilizaron las clases *TCustomEdit* y *TCustomCheckBox* como base para la extensión, porque no es importante para esta explicación. Lo relevante es que Borland tuvo que proceder independientemente con cada uno de los dos, y añadió por separado las propiedades, métodos y eventos necesarios. Por ejemplo, las propiedades *DataSource* y *DataField*, que identifican el conjunto de datos y el campo del mismo al que se asocia el componente.

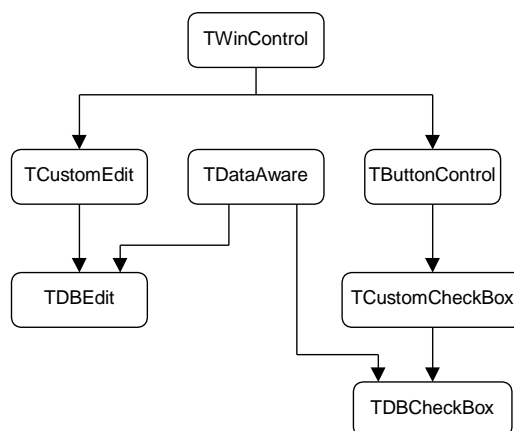
Supongamos entonces que necesitamos una función o procedimiento que trabaje con *cualquier* control con acceso a datos. Queremos pasarle el control como parámetro, para hacer algo con sus propiedades *DataSource* y *DataField*, por ejemplo. ¿Existe alguna clase común a *TDBEdit* y *TDBCheckBox* que defina las propiedades mencionadas? No, no la hay. En el estado actual de la jerarquía de clases de Delphi, no existe una solución elegante para este problema.

## ¿Herencia múltiple?

Un lenguaje como C++ habría resuelto la dificultad utilizando *herencia múltiple*. Primero, definiría una clase *TDataAware*, con los métodos y propiedades comunes a todos los controles con acceso a bases de datos:

```
type
  TDataAware = class           // ¡¡¡ESTO NO ES DELPHI!!!
    property DataSource: TDataSource;
    property DataField: string;
    // ... y faltarían varios detalles ...
  end;
```

Para obtener entonces un control con acceso a datos “mezclaríamos” una clase base perteneciente a un control con la nueva clase *TDataAware*:



Bajo estas suposiciones, si quisiéramos un procedimiento que actuase sobre un control de datos arbitrario lo declararíamos de este modo:

```
procedure MachacarControlDatos (Ctrl: TDataAware);
```

¿Quiere decir esto que nuestra solución consiste en organizar manifestaciones frente a la sede de Borland para exigir a gritos que implementen la herencia múltiple? ¡Ni mucho menos! El hecho de que Delphi y Java sigan aferrándose a la herencia simple no es casual, ni achacable a la pereza de sus diseñadores. En primer lugar, la herencia múltiple introduce en un lenguaje más problemas de los que resuelve. No es el lugar para discutir el por qué, pero verifique cuán complejas son las reglas de un lenguaje como C++ respecto a la semántica de la herencia múltiple. La implementación de este recurso es bastante complicada. Y para rematar, seguimos teniendo la dependencia oculta respecto al formato de atributos y métodos internos, pero ahora multiplicada por el número de ancestros de la clase.

### IEFFEL, C++ Y LA HERENCIA MULTIPLE

Hay dos dificultades principales con la herencia múltiple. La primera se presenta cuando una clase hereda dos métodos con el mismo nombre desde dos de sus ancestros. La segunda ocurre en el caso en que una clase es ancestro indirecto de otra a través de dos vías distintas. Supongamos que existe una clase *TControl* en determinada biblioteca de la cual derivan las clases *TCtrlEditor* y *TCtrlLista*, y que la clase de los combos *TCtrlCombo* se crea heredando de esas dos clases intermedias. ¿Quiere comprobar si es un problema fácil o no? Analice las soluciones que ofrecen C++ y Eiffel: más diferentes y arbitrarias no podrían ser.

## Tipos de interfaz

En Borland, prefirieron introducir los *tipos de interfaz*. Un tipo de interfaz, o simplemente una *interfaz*, se declara con una sintaxis similar a la siguiente:

```
type
  IAparatoElectrico = interface
    function VoltajeNecesario: Double;
    function PotenciaNominal: Double;
    function GetEncendido: Boolean;
    procedure SetEncendido(Valor: Boolean);
  end;
```

Aparentemente, una declaración de un tipo de interfaz es muy similar a la declaración de un tipo de clase. Pero deben cumplirse las siguientes reglas:

- Dentro de un tipo de interfaz no pueden declararse variables, sino solamente métodos y propiedades.
- Las propiedades, por la restricción anterior, deben utilizar métodos solamente para sus formas de acceso.
- Una interfaz no ofrece una implementación para sus métodos; se limita a las declaraciones de los mismos.
- Todo lo que se declara dentro de un tipo de interfaz es obligatoriamente público. No existen secciones privadas y protegidas, como en las declaraciones de clases.



La siguiente declaración muestra cómo podríamos haber declarado propiedades en la interfaz anterior:

```
type
  IAparatoElectrico = interface
    // Métodos
    function VoltajeNecesario: Double;
    function PotenciaNominal: Double;
    function GetEncendido: Boolean;
    procedure SetEncendido(Valor: Boolean);
    // Propiedades
    property Encendido: Boolean
      read GetEncendido write SetEncendido;
  end;
```

Si intentamos comprender qué es un tipo de interfaz a partir de conceptos de programación ya conocidos, encontraremos que lo más parecido es una declaración de clase *abstracta*, en la que no existan atributos ni propiedades, todos los métodos sean virtuales y se hayan declarado con la directiva **abstract**, que evita que tengamos que asociarles una implementación. Y en gran medida, se trata de una comparación acertada.

¿Recuerdas que mencioné, en un recuadro anterior, la dualidad entre la herencia de tipos y de implementaciones? Una solución relativamente sencilla a este problema, en los lenguajes que ofrecen herencia múltiple pero no interfaces, sería exigir que la herencia múltiple sea posible cuando sólo uno de los ancestros sea una clase “concreta”. De hecho, al no existir interfaces en C++, se simulan mediante clases que solamente contienen métodos virtuales puros. La simulación mencionada es necesaria para poder utilizar COM desde ese lenguaje.

Una de las consecuencias de las reglas de juego anteriores es que si creásemos una unidad y en ella sólo declarásemos un tipo de interfaz, al ser compilada no se generaría código ejecutable alguno; únicamente información para el compilador. Ya veremos qué tipo de información.

## Interfaces y clases

Sabemos que las *clases* también son un caso especial de tipo de datos, que se utilizan en tiempo de ejecución para crear *objetos*. ¿En dónde encajan entonces nuestras recién presentadas interfaces? ¿Se pueden utilizar para crear objetos? Categóricamente, no. Las relaciones entre estos tres conceptos son un poco más complejas:

- 1 Una clase puede *implementar* una o varias interfaces.
- 2 Para crear objetos, sigue siendo necesario partir de una clase.
- 3 Si un objeto pertenece a la clase *C* que implementa la interfaz *I*, se puede asignar ese objeto a una variable de interfaz de tipo *I*.

Parece un enredo, pero podemos aclararlo utilizando un ejemplo concreto:

- “Alguien” ha definido las interfaces *IAparatoElectrico* e *IProducto*. Un *aparato eléctrico* se puede encender, apagar y consume potencia eléctrica; no sabemos cómo lo hace en concreto. Un *producto* es algo que se puede vender, que va asociado a un tipo de impuesto de ventas, y que tiene un plazo de garantía.
- También “alguien” ha creado la clase *TImpresora*<sup>1</sup>. La clase implementa, de una forma que tengo que explicar todavía, las dos interfaces mencionadas. Con toda probabilidad, también implementará una interfaz llamada *IImpresora*; observe que he cambiado la letra inicial.
- Cuando compré mi impresora, el dependiente de la tienda asignó mentalmente ese objeto a una variable de tipo *IProducto*, y realizó con él las operaciones típicas en una venta. Las operaciones habían sido declaradas inicialmente en la interfaz, pero recibieron código concreto dentro de la clase *TImpresora*. Ese día también compré un cable de red de diez metros en la misma tienda. Para el empleado, se trataba también de otro *IProducto*, que esta vez, no era un *IAparatoElectrico*.
- Posteriormente, ya en mi oficina, he utilizado principalmente las interfaces *IAparatoElectrico* e *IImpresora*. Ultimamente, los métodos de la segunda interfaz solamente aciertan a lanzar excepciones cuando los llamo. He intentado recuperar la interfaz *IProducto* para aprovechar la garantía. Pero viendo el nulo caso que me hacen en la tienda, estoy pensando en asignar el condenado objeto en una variable de tipo *LArmaContundente* y llamar a varios de sus métodos con el empleado en el primer parámetro...

## Implementación de interfaces

Si la operación de *implementar* una interfaz por parte de una clase fuese complicada, quizás no merecería la pena utilizar interfaces... pero es algo extraordinariamente fácil. Supongamos que queremos declarar la clase *TTostadora*, y que ésta debe ofrecer la funcionalidad de un aparato eléctrico y de un producto comercial. Observe la siguiente declaración:

```
type
  // !!!Faltan métodos importantes!!!
  TTostadora = class(TObject, IAparatoElectrico, IProducto)
    // ...
  protected // Métodos de la interfaz IAparatoElectrico
    function VoltajeNecesario: Double;
    function PotenciaNominal: Double;
    function GetEncendido: Boolean;
    procedure SetEncendido(Valor: Boolean);
  protected // Métodos definidos por IProducto
    function Precio: Currency;
    function Garantia: Integer;
    // ... más declaraciones ...
  end;
```

---

<sup>1</sup> Tengo una frente a mí, con un atasco de papel impresionante...

En primer lugar, se ha ampliado la sintaxis de la cláusula de herencia., a continuación de la palabra reservada **class**. Aunque hay tres identificadores de tipos dentro de la cláusula, solamente el primero puede referirse a una clase; el resto, deben ser tipos de interfaz. He utilizado como clase base nuestro conocido *TObject*, aunque no es lo usual, como veremos más adelante. Si una clase implementa interfaces y descende directamente de *TObject*, no se puede omitir el nombre de la clase base, al contrario de lo que sucede en clases que no implementan interfaces. Supongo que esta restricción sintáctica se ha introducido para facilitar el trabajo del compilador.

La forma más simple de *implementar* una interfaz mencionada en la cláusula de herencia consiste en proporcionar métodos cuyos nombres y prototipos coincidan con los definidos por la interfaz. En el ejemplo anterior, la clase *TTostadora* introduce todos los métodos de *LApaparatoElectrico*: *VoltajeNecesario*, *PotenciaNominal* y los dos relacionados con el encendido. Sin embargo, sería también correcto heredar esos métodos de la clase ancestro. Note que no hemos hecho nada respecto a la propiedad *Encendido* de *LApaparatoElectrico*, porque se trata simplemente de una facilidad sintáctica, lo que los anglosajones denominan *syntactic sugar*.

¿En qué sección de visibilidad de la clase deben encontrarse los métodos usados en la implementación de interfaces? En este ejemplo los he colocado en **protected**, pero en realidad podía haberlos definido en cualquier otra sección. ¿Por qué **protected**? Porque a pesar del empeño de Borland en generar secciones **private** en formularios, marcos y módulos de datos, en la buena programación orientada a objetos no se emplea **private** excepto en circunstancias extremas. Un recurso **protected** puede ser utilizado por clases derivadas, mientras que uno privado termina su carrera en la clase actual. Normalmente no podemos predecir si el método que estamos a punto de declarar va a ser necesario o no para alguna clase derivada. Personalmente, sólo uso **private** para los métodos y atributos utilizados en la definición de propiedades, porque en ese caso sigo teniendo acceso indirecto a los recursos privados a través de dichas propiedades.

#### ADVERTENCIA

Hay que tener cuidado con un caso especial. Si definimos un método en la sección **private** de una clase y declaramos una clase derivada en otra unidad, el método privado no puede utilizarse para implementar interfaces en esa clase derivada.

Por último, hay que tener bien presente que, si vamos a implementar una interfaz, hay que suministrar implementaciones a *todos* los métodos de la interfaz. No se admiten implementaciones parciales.

## Conflictos y resolución de métodos

Ahora estudiaremos el potente mecanismo que convierte a las interfaces en un recurso más flexible que la herencia múltiple. Primero veamos los detalles sintácticos, presentando una forma alternativa de declarar la clase *TTostadora*.

```

type
  // ;;;Siguen faltando métodos importantes!!!
  TTostadora = class(TObject, I AparatoElectrico, IProducto)
    // ...
  protected // Métodos definidos por IProducto
    function PrecioProducto: Currency;
    function TiempoGarantia: Integer;
    function IProducto.Precio = PrecioProducto;
    function IProducto.Garantia = TiempoGarantia;
    // ... etcétera ...
  end;

```

La clase *TTostadora* ya no define los métodos *Precio* y *Garantia* que necesitaba *IProducto*, pero sí define *PrecioProducto* y *TiempoGarantia*, con los prototipos adecuados. Entonces le echamos una mano a la interfaz, para que sepa que *PrecioProducto* corresponde a su método *Precio*, y *TiempoGarantia* a su *Garantia*.

Usted podría preguntarse para qué complicarnos, aparentemente, la vida de esta forma: si sabíamos que *TTostadora* iba a implementar *IProducto*, deberíamos haber dado nombres correctos a los métodos desde el primer momento. Pero es ahí donde está el fallo de razonamiento. En la programación “real” es muy posible que un método de implementación de interfaces haya sido definido tiempo atrás en una clase ancestro. En aquel momento recibió un nombre diferente, pero ahora reconocemos que es un buen candidato para asociarse a un método de cierta interfaz.

De todos modos, el motivo más importante para contar con una cláusula de resolución de métodos es la posibilidad de que una clase implemente simultáneamente dos interfaces, y que existan métodos comunes a ambas con el mismo nombre. Veamos dos casos concretos, muy sencillos, pero que ilustran dos estrategias diferentes:

- 1 Alguien ha definido la interfaz *IVela*, no para las velas de los barcos, sino para las fuentes de luz utilizadas en las cenas románticas. La interfaz *IVela* declara el método *SetEncendido*. Más tarde, algún genio taiwanés concibe la idea kitsch de la clase *TVelaElectrica*, que debe implementar las interfaces *IAparatoElectrico* e *IVela*. Naturalmente, encender la vela corresponde al mismo acto físico que encender el aparato. Podemos entonces crear un único método *SetEncendido* en *TVelaElectrica*; no necesitaremos entonces cláusula de resolución alguna. La “esencia” de la vela se ha fundido con la “esencia” del aparato eléctrico, y por ese motivo comparten la implementación de varios de sus métodos.
- 2 En el segundo caso, las “esencias” no se mezclan. Tomemos las interfaces *Iconductor* e *IPoliciaTrafico*: ambas definen una propiedad *Multas*; en realidad, definen los dos métodos de acceso correspondiente, pero me referiré a la propiedad para simplificar. *Paco* es un objeto de la clase *TPoliciaTrafico*, que implementa las dos interfaces mencionadas, porque un policía de carreteras debe también conducir. Las multas de *Iconductor* se refieren a las que sufre el propio policía cuando está fuera de servicio, mientras que las de *IPoliciaTrafico* hacen referencia a las que él inflige a los demás. Por lo tanto, los métodos de implementación deben ser completamente independientes. La clase podría definir, por ejemplo, *GetMultas*-

*Conductor* y *GetMultasPolicia*. Haría falta entonces una cláusula de resolución de métodos:

```
function IConductor.GetMultas = GetMultasConductor;
function IPoliciaTraficoGetMultas = GetMultasPolicia;
```

- 3 Sin embargo, en el mismo ejemplo del policía, si el modelo es detallado, ambas interfaces pueden también declarar un método *Respirar*. Los pulmones del policía son los mismos que los del conductor, es decir: los mismos de *Paco*. Quiere decir que es posible, al implementar simultáneamente dos interfaces, que una parte de los métodos se fusionen entre sí, pero que la otra requiera implementaciones independientes.

Hay más detalles en el uso de cláusulas de resolución de métodos, en relación con una característica denominada *implementación por delegación*. Dejaremos su estudio para más adelante.

## La interfaz más primitiva

Me temo que voy a complicar un poco el cuadro. Resulta que las interfaces se relacionan entre sí mediante un mecanismo muy similar a la herencia simple. Me resisto a llamarlo “herencia”, porque como veremos más adelante no tiene muchas de las implicaciones de la herencia de clases que conocemos. Pero a falta de otro nombre más apropiado, seguiremos hablando de *herencia entre interfaces*.

Cuando definimos un tipo de interfaz, podemos indicar que el nuevo tipo *extiende* la definición de otro ya existente:

```
type
    IAparatoPortatil = interface (IAparatoElectrico)
        function NivelBaterias: Double;
    end;

    IImpresora = interface (IAparatoElectrico)
        procedure CargarPapel;
        procedure Imprimir(D: IDocumento);
    end;
```

Con esto estamos indicando que un aparato portátil nos ofrece siempre los mismos métodos que un aparato eléctrico *más* la función *NivelBaterias*. Lo mismo podemos decir de *IImpresora*. Sin embargo, podríamos haber declarado la interfaz de impresoras sólo con sus métodos específicos. De esa forma dejaríamos el campo abierto a las impresoras nucleares, si es que existen. Más adelante, profundizaremos en las consecuencias de una u otra decisión de diseño.

Observe que el parámetro *D* del método *Imprimir* recibe un tipo de interfaz, si confiamos en el nombre del tipo que he utilizado. Cualquier objeto de una clase que implemente la interfaz *IDocumento* podría ser impreso, sin importar cuál es su línea de herencia o árbol genealógico.

Es lógico entonces que nos preguntemos si existe alguna interfaz que sirva de ancestro a todas las restantes interfaces, del mismo modo que en Delphi todas las clases descienden directa o indirectamente de la clase especial *TObject*. O si no existe tal interfaz, como ocurre con las clases de C++. La respuesta es afirmativa: existe una interfaz primaria, llamada *IInterface*. Su declaración es la siguiente:

```
type
  IInterface = interface
    ['{00000000-0000-0000-C000-000000000046}']
    function QueryInterface(const IID: TGUID;
      out Obj): HRESULT; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  end;
```

No me pregunte ahora qué demonios significa el engendro de la tercera línea; por ahora bastará con que sepa que es un número de 128 bits que sirve de identificación a la interfaz.

Los tres métodos declarados en *IInterface* se agrupan en dos áreas de funcionalidad:

- 1 *\_AddRef* y *\_Release* deben implementar un contador de referencias interno al objeto que implementa interfaces. Su propósito es garantizar la liberación de objetos que no están siendo referidos por nadie. Este concepto es completamente ajeno al espíritu de Pascal, en el que la liberación de memoria dinámica era completamente manual, pero también es extraño a Java, donde se utiliza la recogida automática de basura. Delata el origen de *estas* interfaces en el mundo de la programación COM.
- 2 *QueryInterface* es un método de *introspección*, que nos permite saber, teniendo un puntero de interfaz de tipo *X*, si el objeto correspondiente ofrece también una implementación del tipo *Y*. Su existencia también está determinada por las necesidades de la programación COM.

Dentro de poco, profundizaremos en esas dos áreas. Antes de terminar la sección quiero mencionar la existencia de un sinónimo del tipo *IInterface*, llamado *IUnknown*:

```
type
  IUnknown = IInterface;
```

En realidad, este segundo nombre es el que recibe la interfaz básica en COM. Recuerde, sin embargo, que COM es específico de Windows, y que Borland tiene un producto llamado Kylix que se ejecuta en Linux. Al parecer, llamar *IInterface* a algo que a todas luces es simplemente un *IUnknown* es una concesión para que a los fanáticos del sistema del pingüino no se les atraganten las interfaces como traicioneras espinas de pescado polar.

Como todos los tipos de interfaz descienden directa o indirectamente de *IInterface*, todos ellos incluyen también los tres métodos básicos que acabamos de presentar.

Eso también significa que cualquier clase que pretenda implementar al menos una interfaz debe proporcionar una implementación para dichos métodos. Delphi nos ayuda mediante la clase predefinida *TInterfacedObject*, que contiene una implementación estándar de *QueryInterface*, *\_AddRef* y *\_Release*.

## La asignación polimórfica

Vamos a cerrar el círculo, por fin, viendo cómo funcionan las asignaciones de objetos a variables de tipo interfaz. Según lo que hemos visto hasta el momento, existen dos tipos de variables que contienen punteros a objetos en Delphi:

- 1 Variables de tipo clase.
- 2 Variables de tipo de interfaz.

Con los medios que conocemos hasta el momento, todo objeto comienza su vida cuando se ejecuta un constructor de la clase a la que pertenecerá; el constructor devuelve un puntero de tipo clase.

### NOTA

Los objetos COM se crean mediante funciones del sistema que devuelven un puntero de tipo interfaz. Claro, detrás del proceso de creación hay siempre un constructor, pero las funciones de COM disimulan su existencia.

La regla de la asignación polimórfica entre variables y punteros de tipo clase siguen siendo las de toda la vida:

- Si la asignación  $a := b$  es correcta,  $a$  es una variable de clase de tipo  $X$ , y  $b$  es una expresión que devuelve una referencia a objeto declarada mediante el tipo  $Y$ , entonces la clase  $Y$  es igual a  $X$ , o es un descendiente directo o indirecto de la misma.

O, en otras palabras, podemos asignar sobre una variable de la clase *TAnimal* referencias a objetos de las clases *TPerro*, *TGato*, etc. Pero ahora tenemos una regla para validar las asignaciones desde una variable o expresión de tipo clase en una variable de tipo interfaz:

- Si la asignación  $a := b$  es correcta,  $a$  es una variable del tipo de interfaz  $X$ , y  $b$  devuelve una referencia a objeto declarada mediante la clase  $Y$ , entonces la clase  $Y$  implementa la interfaz  $X$ , o desciende de una clase que la implementa.

Traducido: en una variable de tipo *LAparatoElectrico* podemos asignar un objeto de la clase *TTostadora* o de cualquiera de sus descendientes. Sencillo.

Finalmente, tenemos la regla que rige las asignaciones entre punteros de tipo interfaz. Tiene la misma forma que la regla de asignación entre variables de tipo clase:

- Si la asignación  $a := b$  es correcta,  $a$  es una variable del tipo de interfaz  $X$ , y  $b$  es un puntero de interfaz de tipo  $Y$ , entonces la interfaz  $Y$  es igual a  $X$ , o es un descendiente directo o indirecto de  $X$ .

## Tiempo de vida

Antes de seguir con las reglas de compatibilidad de tipos, programemos un pequeño ejemplo, para mostrar un comportamiento de Delphi que puede sorprender. Inicie una nueva aplicación, y añádale una unidad como la siguiente:

```
unit Animales;
interface

type
  IAnimal = interface
    procedure Hablar;
  end;

  TVaca = class(TInterfacedObject, IAnimal)
    destructor Destroy; override;
    procedure Hablar;
  end;

implementation
uses Dialogs;

destructor TVaca.Destroy;
begin
  ShowMessage('¡Me han matado!');
  inherited Destroy;
end;

procedure TVaca.Hablar;
begin
  ShowMessage('¡Muuu!');
end;

end.
```

Luego, añada un botón en la ventana principal y programe su evento *OnClick* de la siguiente manera:

```
procedure TwndPrincipal.Button1Click(Sender: TObject);
var
  I: IAnimal;
begin
  I := TVaca.Create;
  I.Hablar;
end;
```

No parece que sea muy complejo: la llamada al constructor de *TVaca* crea un objeto de esa clase y devuelve una referencia al mismo. Como *TVaca* implementa la interfaz *IAnimal*, se puede asignar esa referencia directamente en una variable de este último tipo. Finalmente, llamamos al método *Hablar* a través del puntero a la interfaz. Nada



más. Pero ejecute el ejemplo; comprobará que al finalizar la respuesta a *OnClick*, ¡la vaca es destruida!

Si recuerda la existencia de los métodos *\_AddRef* y *\_Release*, comprenderá lo que sucede: Delphi mantiene una variable dentro de los objetos para contar cuántas variables de interfaz están apuntando al mismo. Esto es un requisito de COM, pero en realidad COM se limita a exigirnos que llamemos a *\_AddRef* cada vez que el objeto sea asignado a un puntero de interfaz, y a *\_Release* cuando una variable que apuntaba a nuestro objeto cambie su valor. Es muy peligroso realizar esas llamadas manualmente, por lo que Delphi nos echa una mano y genera a nuestras espaldas el código necesario. Por ejemplo, el método anterior incluye las siguientes instrucciones “ocultas”:

```
// ;;;Esto es pseudo código: no tomar en serio!!!

procedure TwndPrincipal.Button1Click(Sender: TObject);
var
    I: IAnimal;
begin
    I := nil;
    try
        I := TVaca.Create;           // El contador se inicializa con 0
        I._AddRef;                   // El contador se incrementa en 1
        I.Hablar;
    finally
        if I <> nil then
            I._Release;             // El contador cae a 0 ...
    end;                             // ... y se destruye el objeto.
end;
```

¿Cuántas veces le repitió su profesor de Pascal que las variables locales no se inicializan solas, por arte de magia? El profe tenía razón, porque es lo que sigue sucediendo con casi todos los tipos de datos de Delphi... excepto los tipos de interfaz, las cadenas de caracteres y *Variant*. Las variables de estos tipos especiales siempre se inicializan con ceros: el valor **nil** para las interfaces, la cadena vacía para el tipo **string** y la constante *Unassigned* para los variantes.

Pero la inicialización mencionada se lleva a cabo para que Delphi pueda perpetrar otra herejía: aplicar algo parecido a un destructor sobre esas variables. En el caso de los tipos de interfaz, si se ha asignado algo en la variable, se llama al método *\_Release* para decrementar el contador de referencias del objeto correspondiente. La llamada a dicho método no implica siempre la muerte del objeto. Observe la siguiente variación sobre el ejemplo, que utiliza una variable global:

```
var                               // Una variable global (inicializada con nil)
    BichoGlobal: IAnimal;

procedure TwndPrincipal.Button1Click(Sender: TObject);
var
    I: IAnimal;
begin
    I := TVaca.Create;
```

```

    I.Hablar;
    BichoGlobal := I; // El rumiante salva su pellejo
end;

```

En ese caso, el contador de referencia sube hasta el valor máximo de 2, justo antes de finalizar el método, y se queda con el valor 1 después de terminar, por lo que no se destruye. Si hacemos clic otra vez sobre el botón y creamos una nueva vaca, la vaca anterior se destruye cuando asignamos la variable local sobre la global. Es así porque la asignación añade las siguientes instrucciones automáticas:

```

if BichoGlobal <> nil then
    BichoGlobal. Release; // La vaca vieja
    BichoGlobal := I;
if BichoGlobal <> nil then
    BichoGlobal. AddRef; // La nueva vaca

```

La vaca se destruirá al finalizar la aplicación, pero usted no verá aparecer el diálogo de *ShowMessage* en el monitor. Si desconfía de mi palabra, ponga un punto de ruptura en el destructor de *TVaca*, para que compruebe que el puntero de instrucciones pasa por ahí. Como sugerencia, busque una explicación para este fenómeno.

También podríamos salvar la vida del animal de forma explícita, aunque no es una técnica recomendable. Entre otras razones, porque si no utilizamos instrucciones adicionales en el código de finalización de la unidad, la vaca no morirá al terminar la aplicación:

```

procedure TwndPrincipal.Button1Click(Sender: TObject);
var
    I: IAnimal;
begin
    I := TVaca.Create;
    I.Hablar;
    I._AddRef; // ;;; No es recomendable !!!
end;

```

Como última prueba, ejecute el siguiente código:

```

procedure TwndPrincipal.Button1Click(Sender: TObject);
var
    V: TVaca;
begin
    V := TVaca.Create;
    V.Hablar;
end;

```

En este caso, la vaca muge pero no muere, porque en vez de agarrar el objeto con “pinzas” de tipo interfaz, hemos utilizado las “pinzas” tradicionales, de tipo clase. Claro, el alma en pena de la vaca vagará indefinidamente por la memoria RAM hasta que finalice la ejecución del programa.

Podemos resumir así el contenido de la sección:

- Existen en Delphi dos sistemas paralelos de referencias a objetos: a través de variables de tipo interfaz o por medio de variables de tipo clase.
- El segundo sistema es el que siempre hemos utilizado en Delphi.
- El sistema de variables de interfaz implementa la destrucción automática de objetos mediante contadores de referencias. Delphi genera automáticamente las instrucciones que incrementan y decrementan el valor del contador.
- Aunque existe una regla de compatibilidad que nos permite “saltar” de un sistema a otro, no es una técnica recomendable, porque complica el seguimiento del tiempo de vida de los objetos. O utilizamos el sistema de interfaces o el de variables de tipo clase; nunca ambos a la vez sobre las mismas clases.

## Compatibilidad al pie de la letra

El ejemplo de esta sección nos mostrará una radical diferencia existente entre las interfaces y la herencia múltiple de clases. Si lo prefiere, copie la aplicación del ejemplo anterior en un nuevo directorio y modifique las declaraciones de tipos de la unidad *Animales* del siguiente modo:

```
type
  IAnimal = interface
    procedure Hablar;
  end;

  IRumiante = interface(IAnimal)
    procedure Rumiar;
  end;

  TVaca = class(TInterfacedObject, IRumiante)
    destructor Destroy; override;
    procedure Hablar;
    procedure Rumiar;
  end;
```

Esta vez la clase *TVaca* implementa una interfaz que, a su vez, ha sido definida a partir de otro tipo de interfaz. Puede implementar el método *Rumiar* de las vacas con cualquier tontería:

```
procedure TVaca.Rumiar;
begin
  ShowMessage('Ñam-ñam');
end;
```

¿Qué tal si añade un botón a la ventana principal, intercepta su evento *OnClick* con la siguiente respuesta e intentar compilar la aplicación?

```
// ;;; No compila !!!
procedure TForm1.Button1Click(Sender: TObject);
var
  I: IAnimal;
begin
  I := TVaca.Create;
```

```

        I.Hablar;
    end;

```

El compilador se quejará de que la vaca no es compatible con un puntero a un animal. Pero esto destruye la analogía entre una interfaz y una clase abstracta: en definitiva, un rumiante *es* un animal, ¿no? Para mostrar lo extraño que es este comportamiento, incluiré un ejemplo más o menos equivalente en C++:

```

class IAnimal {
public:
    virtual void Hablar() = 0;
};
class IComida {
public:
    virtual void Probar() = 0;
};
class IRumiante: public IAnimal {
public:
    virtual void Rumiar() = 0;
};
class TVaca: public IComida, public IRumiante {
public:
    void Hablar() {
        ShowMessage(";Moo!");
    }
    void Rumiar() {
        ShowMessage(";Ñam!");
    }
    void Probar() {
        ShowMessage(";Qué bueno el filete!");
    }
};

```

Si su C++ le falla, no se preocupe: aquí se define una clase vaca que hereda de las clases *abstractas* o diferidas *IComida* e *IRumiante*; esta última descende por herencia simple de *IAnimal*. Se trata de clases abstractas porque solamente contienen métodos virtuales sin implementación. La igualación a cero en la declaración de los métodos es el sistema de C++ para declarar que un método no tiene implementación, y que debe proporcionársele una en algún descendiente de la clase. Eso es precisamente lo que hace la clase *TVaca*: proporcionar código a los métodos “puros” *Probar*, *Hablar* y *Rumiar*.

Para demostrar que en C++ sí se puede asignar una vaca en un puntero a un animal, tendríamos que crear un manejador de eventos como el siguiente:

```

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    TVaca* gertrude = new TVaca;
    IAnimal* animal = gertrude;
    animal->Hablar();
}

```

El ejemplo anterior no sólo compila, sino que además hace lo que todos esperamos: el animal muge. A propósito, no estaba pensando en ninguna Gertrude en particular.

Simplemente me pareció un nombre adecuado para el animal. A mi vaca la imagino robusta, rubicunda y con una particular afición por las salchichas (de cerdo, por supuesto) y la cerveza en jarra.

No hacía falta, sin embargo, irnos a C++ para revelar la paradoja. Nos habría bastado con probar este método alternativo:

```
// ;;; Sí compila !!!
procedure TForm1.Button1Click(Sender: TObject);
var
    R: IRumiante;
    A: IAnimal;
begin
    R := TVaca.Create;
    A := R;
    A.Hablar;
end;
```

No podemos asignar una vaca en un animal directamente, ¡pero si pasamos la referencia primero a un rumiante Delphi, no pone trabas! ¿Qué está pasando? ¿Acaso hay falta de fósforo en la dieta de los empleados de Borland?

Siento decepcionarle, pero en Borland han razonado bien. Tomemos como ejemplo las siguientes declaraciones:

```
type
    IAeropuerto = interface
        function Nombre: string;
        // ... etcétera ...
    end;
    IAeropuertoOrigen = interface(IAeropuerto)
        // ... etcétera ...
    end;
    IAeropuertoDestino = interface(IAeropuerto)
        // ... etcétera ...
    end;

    TRuta = class(TInterfaceObject,
        IAeropuertoOrigen, IAeropuertoDestino)
        // ... etcétera ...
    end;
```

Supongamos que tenemos un objeto de tipo *TRuta* en nuestras manos, y admitamos por un momento que la siguiente asignación fuese aceptada por Delphi:

```
procedure TrazarRuta(R: TRuta);
var
    A: IAeropuerto;
begin
    A := R;
    ShowMessage(A.Nombre);
end;
```

¿Qué nombre de aeropuerto debería mostrar *ShowMessage*, el de partida o el de llegada? No hay respuesta posible; antes, tendríamos que plantearnos otra pregunta de

igual importancia: la clase *TRuta* implementa dos interfaces derivadas de *IAeropuerto*. ¿Quiere eso decir que internamente deben existir dos “instancias” de aeropuertos dentro de la clase, para que los métodos comunes de ambas interfaces funcionen de forma independiente? Para este ejemplo concreto, está claro que sí, que queremos que las partes comunes a *IAeropuertoOrigen* e *IAeropuertoDestino* sean independientes.

Pero observe este otro ejemplo, similar en su estructura al anterior:

```
type
  IPersona = interface
    function Nombre: string;
    // ... etcétera ...
  end;
  IConductor = interface(IPersona)
    // ... etcétera ...
  end;
  IPolicia = interface(IPersona)
    // ... etcétera ...
  end;

  TPoliciaTrafico = class(TInterfaceObject,
                        IConductor, IPolicia)
    // ... etcétera ...
  end;
```

En este caso, la parte común a *IConductor* e *IPolicia* es una persona, y está muy claro que es la misma entidad representando dos papeles. Ahora ya puedo explicar por qué no se permite asignar una vaca sobre un puntero a animal:

- Si permitiésemos asignar un puntero a *TVaca* en un *IAntimal*, también tendríamos que permitir asignar un *TRuta* en un *IAeropuerto*; pero eso introduciría ambigüedades semánticas muy importantes.

De todos modos, ya conocemos una forma de solucionar el problema: asignamos la referencia *TVaca* en una variable intermedia *IRumiante*, para luego hacer la asignación deseada sobre el *IAntimal*. Aplique la técnica a *TRuta*: en la asignación intermedia va a tener que seleccionar entre *IAeropuertoOrigen* e *IAeropuertoDestino*, y esa decisión es la que elimina la ambigüedad.

Pero tenemos otra forma más directa de lograr el mismo efecto, incluyendo explícitamente la interfaz base en la lista de interfaces implementadas por la clase:

```
type
  // Ya se permiten las asignaciones directas sobre IAnimal
  TVaca = class(TInterfaceObject, IRumiante, IAnimal)
    // ... etcétera ...
  end;
```

En el caso de la vaca, no hay dudas acerca de que los métodos comunes de *IAntimal* e *IRumiante* deben compartir la misma implementación.

```
type
```

```

TRuta = class(TInterfacedObject, IAeropuerto,
              IAeropuertoOrigen, IAeropuertoDestino)
// ... etcétera ...
end;

```

En el caso de las rutas, si incluimos explícitamente la interfaz *IAeropuerto* en la lista de interfaces implementadas por la clase, tendremos que utilizar cláusulas de resolución de métodos para indicar si el “aeropuerto por omisión” se refiere al de origen, al de llegada ... o a una entidad completamente diferente que nos podríamos inventar.

## Identificadores globales únicos

Cuando presenté la declaración de la interfaz básica *IInterface*, advertí sobre la existencia de un número muy extraño, situado a continuación de la palabra clave **interface**:

```
[ '{00000000-0000-0000-C000-000000000046}' ]
```

Si cuenta los dígitos, comprobará que tiene treinta y dos de ellos. Como se trata de dígitos hexadecimales, deducimos que la extraña cadena representa un número de 128 bits. Estos números reciben el nombre genérico de *Identificadores Globales Unicos* (*Global Unique Identifiers*), pero es habitual que nos refiramos a ellos por sus siglas en inglés: *GUID*<sup>2</sup>.

Su existencia se debe a una exigencia de COM, que pide que cada tipo de interfaz vaya asociado a uno de esos números gigantescos; como veremos en el siguiente capítulo, lo mismo sucede con otras entidades que maneja esta API. La necesidad viene dada por uno de los objetivos de COM: permitir el intercambio de clases programadas en distintos lenguajes. El nombre del tipo de interfaz la identifica sobradamente... pero sólo dentro del lenguaje en que es definida. Además, lo que puede ser un identificador válido en un lenguaje de programación, puede ser inaceptable para otro.

Por lo tanto, las interfaces deben ir asociadas a un número que se pueda garantizar que sea único... no sólo dentro de su ordenador o su empresa, sino a nivel planetario. No nos servirían los habituales enteros de 32 bits; las direcciones IP en Internet utilizan actualmente ese formato, y ya sabemos que están a punto de desbordarse. En vez de dar un paso tímido hasta los 64 bits, es preferible armarse de valor y saltar a los 128.

En realidad, para garantizar la unicidad de esos valores tendremos que referirnos a algún mecanismo de generación de los mismos. Ese algoritmo existía antes de la llegada de COM, y era utilizado por el API de RPC (*Remote Procedure Calls*). Dentro de Windows, la función que genera GUIDs se encuentra en la unidad *ActiveX*, y su prototipo es el siguiente:

---

<sup>2</sup> Pronunciado con una diéresis en la 'u'.

```
function CoCreateGuid(out guid: TGUID): HRESULT; stdcall;
```

El tipo *TGUID*, por su parte, se define en *System*:

```
type
  PGUID = ^TGUID;
  TGUID = packed record
    D1: LongWord;
    D2: Word;
    D3: Word;
    D4: array [0..7] of Byte;
  end;
```

Y la función que convierte ese registro binario en el extravagante formato de cadena que hemos visto pegado a *IInterface*, junto con su inversa, están definidas en *ComObj*:

```
function StringToGUID(const S: string): TGUID;
function GUIDToString(const ClassID: TGUID): string;
```

Por lo tanto, para generar una cadena que represente un identificador global único necesitamos una función parecida a la siguiente:

```
function GenerarGUID: string;
var
  G: TGUID;
begin
  CoCreateGUID(G);
  Result := GUIDToString(G);
end;
```

No hay que ponerse tan solemne para pedirle un GUID al ordenador. Si vamos al Editor de Código de Delphi, y pulsamos las teclas CTRL+SHIFT+G, aparecerá uno de estos números como por arte de magia:

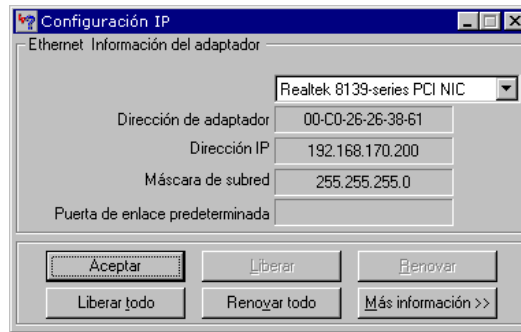
```
'{4DBC6B40-893A-11D5-BA53-00C026263861}'
```

El valor que he mostrado ha sido generado realmente en el ordenador donde estoy escribiendo. Haga usted mismo la prueba varias veces y comprobará:

- 1 Que no va a obtener el mismo número que yo obtuve (ni siquiera uno parecido).
- 2 Que cada vez que pulse esas teclas, obtendrá un valor diferente.

La explicación del primer punto es sencilla: al generar un GUID, se incluye en el mismo un valor de 48 bits extraído de la tarjeta de red de su ordenador, y conocido generalmente como la *dirección MAC*. Se supone, para empezar, que ese número de por sí ya es único. Cada fabricante de tarjetas de red tiene su propio rango asignado de direcciones MAC, y debe gastar uno de ellos por cada tarjeta que fabrique. Para averiguar con total certeza el número MAC de su tarjeta, debe ejecutar la aplicación *winipcfg* desde la línea de comandos, si es que está ejecutando Windows 9x/ME:





En Windows NT y Windows 2000 debe teclear el siguiente comando en la línea de comandos:

```
C:\WinNT\ipconfig /all
```

En mi caso, los ordenadores con los que estoy trabajando tienen, cada uno, dos tarjetas de red. El sistema operativo, por supuesto, escoge solamente una de ellas para generar GUIDs. El problema, sin embargo, podría presentarse cuando se trabaja en un ordenador que no tiene tarjeta de red alguna. En primer lugar, no creo que sea algo importante, porque cualquier profesional que esté desarrollando aplicaciones COM con toda seguridad tiene acceso a algún tipo de red. En segundo lugar, si realmente no existe la tarjeta, el sistema operativo “inventa” un valor apropiado tomando bits prestados de la BIOS de la máquina. Es prácticamente imposible, en la práctica, que el número de 48 bits obtenido sea igual en dos ordenadores diferentes.

Pero además de garantizar que el GUID sea único respecto a otros ordenadores, el algoritmo tiene en cuenta la hora del sistema más un contador interno para garantizar que no se obtengan valores iguales en un mismo ordenador, pero en dos momentos distintos.

## Introspección

Pongamos por caso que tiene usted “en su mano” un puntero de tipo *IAntimal*. ¿Habrá alguna forma de saber si ese animal es una *IVaca*, que es una interfaz derivada a partir de *IAntimal*? Pidamos un poco más, ¿se puede averiguar si el animal es comestible, es decir, si el mismo objeto que implementa *IAntimal* también proporciona métodos a la interfaz *IComida*? En el mundo de las referencias de tipo clase, la primera pregunta se resuelve utilizando el operador **as**. En cambio, la segunda pregunta no tiene sentido en Delphi, porque éste no soporta la herencia múltiple.

Para las referencias de tipo interfaz, el operador **as** es uno de los medios disponibles para averiguar la presencia de otras interfaces dentro de un objeto. Como puede adivinar, en el operando izquierdo podemos situar un puntero a objeto o un puntero a interfaz, indistintamente, y en el operando derecho debe ir un nombre de tipo de interfaz. Si la conversión no se puede realizar, se produce una excepción en tiempo

de ejecución, con el mensaje *Interface not supported*. Si la referencia a convertir es un puntero vacío, el operador no falla y devuelve también un puntero vacío. Al parecer, todo sigue funcionando igual que antes. Sin embargo...

... COM vuelve a plantear sus exigencias sobre el guión. El operador **as** de Delphi, aplicado a las referencias de objetos, utiliza en tiempo de ejecución información codificada durante la compilación sobre los tipos de datos presentes en el módulo ejecutable o DLL generado. Es cierto que su implementación es sencilla, pero lo que voy a explicar también es aplicable a C++, lenguaje en el que se utiliza el operador **dynamic\_cast** con fines parecidos.

Supongamos que tenemos una variable de tipo *TObject*, y queremos saber si el objeto al que apunta pertenece a una clase derivada de *TComponent*. Gracias a la colaboración del compilador, el enlazador y el sistema de carga de aplicaciones del sistema operativo, el programa “sabrá” en tiempo de ejecución que el descriptor de la clase *TComponent* se encuentra, digamos, en la dirección \$12345678 de memoria. Por lo tanto, la implementación de **as** toma los cuatro primeros bytes del objeto, que apuntan al descriptor de la clase a la que pertenece el objeto. Si ese puntero coincide con la dirección del descriptor *TComponent*, ya tenemos respuesta afirmativa. En caso contrario, hay que buscar en la clase ancestro de la clase a la que pertenece el objeto. De todo esto, lo importante es que la aplicación determina con mucha facilidad que el descriptor de la clase *X* se encuentra en la dirección de memoria *Y*.

Pero uno de los objetivos de COM es reutilizar clases y objetos entre aplicaciones. Su aplicación puede haber recibido un puntero de interfaz de otra aplicación que quizás ni se encuentre en el mismo ordenador. No podemos entonces identificar el tipo de interfaz mediante una identidad generada localmente por el compilador para “esa” aplicación en particular.

La solución de COM, que es la que adopta Delphi, consiste en asociar GUIDs a los tipos de interfaces. Para COM, lo verdaderamente importante de una interfaz no es su nombre, que es modificable, sino el GUID al que va asociada. Por este motivo, si queremos hacer conversiones dinámicas con la interfaz *IComida* tenemos obligatoriamente que incluir un GUID en su declaración:

```
type
  IComida = interface
    ['{53604160-957F-11D5-BA53-00C026263861}']
    // El GUID fue generador con Ctrl+Shift+G
    procedure Probar;
    function Calorias: Integer;
  end;
```

Si se nos olvida asociar el GUID, obtendremos un curioso mensaje de error, en tiempo de compilación, al intentar aplicar el operador **as**:

*“Operator not applicable to this operand type”*

Es el mensaje de error más confuso que conozco en Delphi. Debería ser sustituido por algo más esclarecedor, como “idiota, se te ha olvidado el GUID”, o algo así.

### Y YO ME PREGUNTO

¿No sería posible que Borland generase automáticamente un GUID para las declaraciones de interfaz que no lo incluyan? Naturalmente, ese GUID tendría validez solamente dentro de esa aplicación; de todos modos, si el programador no ha incluido explícitamente un GUID es porque no tiene intenciones de utilizar la interfaz en COM. Puede que me equivoque, y que la generación del GUID introduzca algún tipo de problema que no alcanzo a ver, por supuesto.

## Preguntad a la interfaz

El operador **as** es un extremista: si no puede realizar la conversión de tipo, se lava las manos y lanza una excepción. Naturalmente, existen situaciones en las que simplemente queremos saber si *X* soporta *Y*, quizás para decidir si utilizar una variante de un algoritmo u otra. Como en el mundo de las referencias a objetos existe un operador **is** para esos menesteres, podríamos esperar que también se pudiera aplicar a la referencias de interfaces. Lamentablemente, no es posible. Si intentamos utilizar **is** con un tipo de interfaz a su derecha, obtendremos el conocido mensaje “*Operator not applicable, etc*”, y esta vez será totalmente en serio.

La solución, sin embargo, ya la hemos presentado antes, aunque sin explicar su funcionamiento. La forma más ortodoxa de saber si el objeto al que apunta una interfaz implementa otra interfaz es ... preguntarle a la propia interfaz. Vale, es una broma, he traducido literalmente el nombre del método *QueryInterface*. Repito su prototipo:

```
type
  IInterface = interface
    function QueryInterface(const IID: TGUID;
      out Obj): HRESULT; stdcall;
    // ... etcétera ...
  end;
```

Sí, ya sé que las tinieblas deben haberse oscurecido, pero vamos a ver qué fácil es utilizarlo con un ejemplo:

```
var
  Gertrude: IRumiante;
  MiCena: IComida;
begin
  Gertrude := TVaca.Create;
  if Succeeded(Gertrude.QueryInterface(IComida, MiCena)) then
    MiCena.Probar;
end;
```

- En primer lugar, hay que reconocer que *QueryInterface* se diferencia de **is** en que hace falta suministrarle una variable puntero de interfaz, para que deje en ella el puntero resultado de la conversión de tipo, si fuese realizable.

- La variable que recibe el resultado se pasa en el segundo parámetro del método. Observe que dicho parámetro utiliza el convenio **out** de transferencia de valores. Además, debe saber que *QueryInterface* incrementa en uno el contador de referencias del objeto, al realizar internamente una asignación a la variable de interfaz.
- El problema que más desconcierta a primera vista es que *QueryInterface* necesita un GUID para su primer parámetro: el número asociado a la declaración del tipo. ¿Qué hacemos, copiamos y pegamos el GUID desde la declaración, con el correspondiente riesgo de errores? No hace falta, porque Delphi permite utilizar el nombre del tipo como si del propio GUID se tratase.
- Por último, en la unidad *Windows* se define la “macro” *Succeeded*, con más letras repetidas que el Mississippi. Sí, en C es una macro, pero Delphi la implementa como una función que recibe un entero de 32 bits y devuelve *True* si su bit más significativo vale cero. *QueryInterface*, como casi todas las funciones de COM, devuelve un valor entero para indicar si se ha producido un error o no. Si no hay errores, el bit más significativo es cero, pero el resto de los bits no tienen porque serlo; es posible que contengan un valor “informativo” en ese caso.

No obstante, no tenemos motivos para estar contentos del todo. *QueryInterface* es un método que se aplica a interfaces, y no funciona con referencias a objetos. Recuerde que usted puede crear un procedimiento que reciba un parámetro de tipo *TObject*, e interesarle saber, de todos modos, si implementa cierta interfaz. Pero disponemos de un potente método definido en la propia clase *TObject*, llamado *GetInterface*:

```
procedure TObject.GetInterface(const IID: TGUID; out Obj): HRESULT;
```

Su prototipo es muy similar al de *QueryInterface*, porque realmente se utiliza internamente en la clase *TInterfacedObject* para implementar ese método. Tenemos, además, toda una batería de funciones globales en la unidad *SysUtils*, que ofrecen diversas variantes de interrogación:

```
function Supports(const Instance: IInterface;
  const IID: TGUID; out Intf): Boolean; overload;
function Supports(const Instance: TObject;
  const IID: TGUID; out Intf): Boolean; overload;
function Supports(const Instance: IInterface;
  const IID: TGUID): Boolean; overload;
function Supports(const Instance: TObject;
  const IID: TGUID): Boolean; overload;
function Supports(const AClass: TClass;
  const IID: TGUID): Boolean; overload;
```

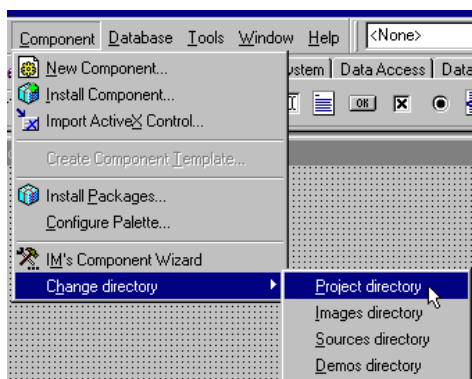
Especialmente útil es la última versión, que acepta una referencia de clase, porque no se necesita un objeto o instancia para realizar la pregunta.

## Interfaces: ejemplos

SENTIDO MUCHO HABER TENIDO QUE LARGAR de golpe toda la teoría del capítulo anterior, sin poner ni un solo ejemplo práctico. Para redimirme, éste contiene un par de ejemplos sencillos de aplicación de las interfaces, con una restricción importante: ninguno de ellos necesita COM. Aprovecharé para explicar, al final del capítulo, cómo Delphi implementa las interfaces, al menos hasta donde he podido averiguar. No es un conocimiento que tenga aplicaciones prácticas directas, pero puede ayudarle a afianzar su comprensión sobre este recurso.

### Extensiones para el Entorno de Desarrollo

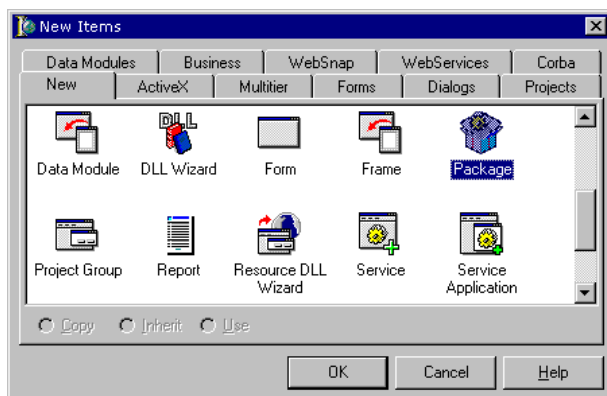
El primer ejemplo que mostraré se aparta de lo habitual: programaremos una *extensión* para el Entorno de Desarrollo de Delphi. Crearemos un *package* o paquete que podrá ser instalado dentro de Delphi; durante la instalación, nuestro código se ocupará de añadir un nuevo comando de menú. Lo que hará ese comando de menú es lo de menos. La imagen que aparece a continuación muestra un ejemplo un poco más sofisticado de extensión del Entorno de Desarrollo.



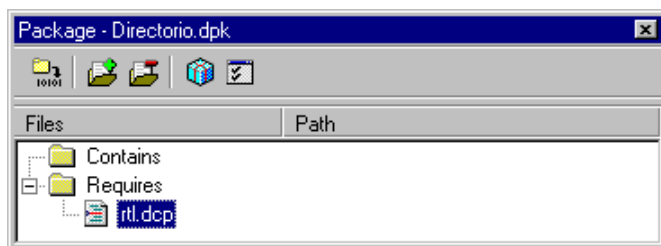
¿Qué tiene que ver todo eso con las interfaces? Pues que para desarrollar este tipo de módulos necesitaremos utilizar el API de Borland conocido con el nombre de *Open Tools API*, que nos permite fisgar en las entrañas de Delphi. Como signo de los tiempos que corren, *Open Tools API* está basado en el uso de interfaces en su casi totali-

dad. Además, para darle más interés al ejemplo, se trata de un API muy poco documentado.

Comencemos por la parte más mecánica del proyecto, ejecutando el comando de menú *File|New|Other* y seleccionando el icono *Package* en la primera página del correspondiente diálogo:



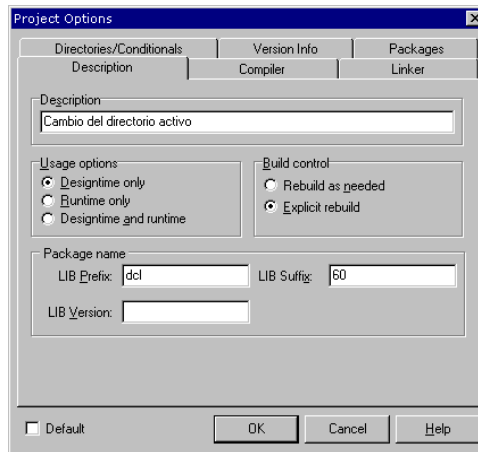
Con ello, crearemos un esqueleto de paquete vacío, del que solamente veremos una ventana flotante con botones y un árbol con los ficheros que forman parte del paquete para su administración.



Estando activa la nueva ventana, ejecute el comando de menú *Project|View source*, para traer al Editor de Código el fichero con las fuentes del paquete. Localice en él la cláusula **requires** y modifíquela de la siguiente manera:

```
requires
  rtl,
  DesignIDE;           // Añada esta unidad
```

En realidad, deberíamos haber podido efectuar el cambio desde la ventana de administración del paquete, pero mi Delphi se ha negado a hacerlo; no sé si es culpa suya o mía, pero no discutiremos por tal minucia. Cuando haya realizado el cambio, regrese a esa ventana, y pulse el botón más a la derecha de la barra de herramientas, para modificar las opciones del proyecto:



En la primera página, debemos seleccionar la opción *Designtime only*, en el cuadro titulado *Usage options*, para indicarle a Delphi que el paquete solamente será utilizado por el Entorno de Desarrollo. En el cuadro *Package name* debemos cambiar el valor de *LIB Prefix* a *dcl*, y el de *LIB Suffix* a *60*, como se muestra en la imagen anterior.

Los dos últimos cambios no son estrictamente necesarios, pero al ser una novedad de Delphi 6, he querido mostrar para qué se utilizan. El prefijo y el sufijo nos servirán como complemento del nombre que le asignemos al paquete. Le sugiero que guarde el proyecto con el nombre de *directorio.dpk*. Así, al compilar, el paquete generado se llamará *dcldirectorio60*: las siglas *dcl* son comunes en los paquetes de tiempo de diseño de Borland, e indican claramente la función del mismo a un programador que solamente tenga en sus manos el fichero compilado. Por otra parte, el número *60* se refiere a la versión de Delphi. Cuando aparezca Delphi 7, solamente tendremos que retocar el sufijo, sin necesidad de cambiar el nombre del proyecto.

Ejecute entonces el comando de menú *File|New|Unit*, para crear una nueva unidad. Guarde el fichero con el nombre de *Experto*, o algo parecido. Y añada, en la sección de interfaz, justo debajo de la palabra **interface**, la siguiente cláusula **uses**:

```
uses Menus, SysUtils, ToolsAPI;
```

## ¿Te llamo o me llamas?

La unidad *ToolsAPI* que acabamos de incluir es la que contiene las declaraciones de tipos de interfaz que vamos a utilizar. Es una unidad *mu*y larga; encontrará su código fuente en el directorio *source\ToolsAPI* de Delphi. No obstante, hay poco código dentro de la misma, pues casi todo el texto corresponde a declaraciones de tipos. No puedo explicar con todo los detalles que quisiera el contenido de *ToolsAPI*, por lo que me limitaré a presentar los procedimientos y tipos que vamos a necesitar.

El primero de ellos es este procedimiento, que será utilizado para registrar nuestra extensión:

```
procedure RegisterPackageWizard(const Wizard: IOTAWizard);
```

Como ve, el tipo del parámetro es un puntero a interfaz. Eso quiere decir que para añadir una extensión a Delphi tenemos que crear una nueva clase e implementar en ella la interfaz *IOTAWizard*, y luego crear una instancia de la clase para pasarla como parámetro a *RegisterPackageWizard*. Veremos en breve dónde debemos llamar a este procedimiento.

Para lograr el mismo efecto sin interfaces, Borland tendría que definir una clase base, y obligarnos a que nuestras extensiones fuesen clases derivadas obligatoriamente de esa clase base. Pero con interfaces tenemos total libertad para escoger nuestra clase base. En este ejemplo, por simplificar, vamos a utilizar el ya manido *TInterfacedObject*; para asistentes más complejos, no obstante, suelo utilizar un módulo de datos. Nuestra única obligación es implementar *IOTAWizard*.

A su vez, la interfaz de *IOTAWizard* es la siguiente:

```
type
  IOTANotifier = interface (IUnknown)
  ['{F17A7BCF-E07D-11D1-AB0B-00C04FB16FB3}']
    procedure AfterSave;
    procedure BeforeSave;
    procedure Destroyed;
    procedure Modified;
  end;

  IOTAWizard = interface (IOTANotifier)
  ['{B75C0CE0-EEA6-11D1-9504-00608CCBF153}']
    function GetIDString: string;
    function GetName: string;
    function GetState: TWizardState;
    procedure Execute;
  end;
```

Cuando nos exigen que implementemos una interfaz ajena, como en este caso, debemos averiguar dos cosas:

- 1 Todos los métodos que tenemos que implementar. Es por este motivo que he incluido también la declaración de *IOTANotifier*, la clase base de *IOTAWizard*. Si sumamos, vemos que hay ocho métodos ansiosos por materializarse.
- 2 Esto es más complicado: hay que saber *qué* se nos exige que haga cada método. Aquí nuestra única guía serán los comentarios que generosamente nos facilite el creador de la interfaz. Afortunadamente, veremos que la mayoría de los métodos anteriores pueden recibir una implementación trivial que no haga nada.

Ya sabemos que tenemos la responsabilidad de implementar *IOTAWizard*. Pero *ToolsAPI* define otras interfaces que podemos utilizar para extraer información del Entorno de Delphi y hacer cambios en su interior. Observe entonces que, aunque la



unidad menciona una larga lista de tipos de interfaz sin más, hay dos papeles que podemos adoptar frente a cada uno de esos tipos:

- 1 Ser el implementador del tipo de interfaz.
- 2 Actuar como *clientes* del tipo de interfaz, para realizar llamadas a métodos a través de sus variables.

De las interfaces que podemos aprovechar como *clientes*, la más importante de todas es la siguiente:

```
type
  IBorlandIDEServices = interface(IUnknown)
    ['{7FD1CE92-E053-11D1-AB0B-00C04FB16FB3}']
  end;

var
  BorlandIDEServices: IBorlandIDEServices;
```

¿Sorprendido? Es cierto que la interfaz anterior es desoladoramente simple, y no nos proporciona pista alguna. Pero lo que sucede es que el Entorno de Delphi, al iniciar su ejecución, crea un objeto de una clase que desconocemos (al menos en teoría) y que implementa como mínimo la interfaz *IBorlandIDEServices*. Podemos entonces utilizar *QueryInterface* para intentar recuperar otros tipos de interfaces a partir de ese puntero “opaco” inicial.

En particular, nuestro ejemplo extraerá de *BorlandIDEServices* punteros a las siguientes interfaces:

```
type
  INTAServices40 = interface(IUnknown)
    ['{3C7F3267-F0BF-11D1-AB1E-00C04FB16FB3}']
    function AddMasked(Image: TBitmap;
      MaskColor: TColor): Integer; overload;
    function GetActionList: TCustomActionList;
    function GetImageList: TCustomImageList;
    function GetMainMenu: TMainMenu;
    function GetToolBar(const ToolBarName: string): TToolBar;

    property ActionList: TCustomActionList read GetActionList;
    property ImageList: TCustomImageList read GetImageList;
    property MainMenu: TMainMenu read GetMainMenu;
    property ToolBar[const ToolBarName: string]: TToolBar
      read GetToolBar;
  end;

  INTAServices = interface(INTAServices40)
    ['{C17B3DF1-DFE5-11D2-A8C7-00C04FA32F53}']
    function AddMasked(Image: TBitmap; MaskColor: TColor;
      const Ident: string): Integer; overload;
  end;

  IOTAModuleServices = interface(IUnknown)
    ['{F17A7BCD-E07D-11D1-AB0B-00C04FB16FB3}']
    function AddFileSystem(FileSystem: IOTAFileSystem): Integer;
```

```

function CloseAll: Boolean;
function CreateModule(const Creator: IOTACreator): IOTAModule;
function CurrentModule: IOTAModule;
function FindFileSystem(const Name: string): IOTAFileSystem;
function FindFormModule(const FormName: string): IOTAModule;
function FindModule(const FileName: string): IOTAModule;
function GetModuleCount: Integer;
function GetModule(Index: Integer): IOTAModule;
procedure GetNewModuleAndClassName(const Prefix: string;
    var UnitIdent, ClassName, FileName: string);
function NewModule: Boolean;
procedure RemoveFileSystem(Index: Integer);
function SaveAll: Boolean;

property ModuleCount: Integer read GetModuleCount;
property Modules[Index: Integer]: IOTAModule read GetModule;
end;

```

Los prefijos *NTA* y *OTA* tienen su explicación: el primero se utiliza con interfaces que dan acceso a objetos “reales” del Entorno de Desarrollo. Por ejemplo, podemos comprobar que *INTAServices* nos permite obtener el puntero al mismísimo menú principal de Delphi. Sí, sonría, por el desastre que podemos provocar al menor descuido. Otras interfaces *NTA* nos devuelven incluso punteros a los componentes que situamos sobre un formulario en tiempo de diseño. A propósito, la *N* de la sigla quiere decir *Native* y las otras dos letras, *Tools API*.

Por el contrario, las interfaces *OTA* son más cautas. En el caso de las estructuras de diseño, en contraste con las *NTA*, nos ofrecen punteros a “representantes” de los objetos. *OTA* quiere decir simplemente *Open Tools API*.

## Implementando la extensión

Ya estamos en condiciones de seguir programando nuestra extensión. En la sección de implementación de la unidad *Experto*, declare la siguiente clase:

```

type
  TimExperto = class(TInterfacedObject, IOTAWizard)
  protected
    FMenuItem: TMenuItem;
    procedure MenuClicked(Sender: TObject);
  protected
    procedure AfterSave;           // Vacío
    procedure BeforeSave;         // Vacío
    procedure Destroyed;          // Vacío
    procedure Modified;           // Vacío
    function GetIDString: string;
    function GetName: string;
    function GetState: TWizardState;
    procedure Execute;             // Vacío
  public
    constructor Create;
    destructor Destroy; override;
  end;

```

Como puede observar, implementaremos con métodos vacíos cinco de los ocho métodos de *IOTAWizard*. Si nos ocuparemos de las tres funciones siguientes, que son utilizadas por el IDE para informarse del propósito del asistente y de su estado:

```
function TimExperto.GetIDString: string;
begin
    Result := 'IntSight.Ejemplo de interfaces';
end;

function TimExperto.GetName: string;
begin
    Result := 'Ejemplo de interfaces-La Cara Oculta de Delphi 6';
end;

function TimExperto.GetState: TWizardState;
begin
    Result := [wsEnabled];
end;
```

Gran parte de la funcionalidad de la extensión se encuentra en el constructor de la clase:

```
constructor TimExperto.Create;
begin
    inherited Create;
    FMenuItem := TMenuItem.Create(nil);
    FMenuItem.Caption := 'Cambiar al directorio del proyecto';
    FMenuItem.OnClick := MenuClicked;
    with (BorlandIDEServices as INTAServices).MainMenu.Items[6] do
        Insert(Count, FMenuItem);
    end;
```

Creamos un componente *TMenuItem* y nos quedamos con la referencia al mismo. Cambiamos su aspecto modificando su propiedad *Caption*; en un ejemplo más completo, utilizaríamos un componente *TAction* asociado al comando de menú, para aprovechar el evento *OnUpdate* de la acción y activar o desactivar el comando, según las circunstancias. Si repasa los métodos de *INTAServices*, verá que también es sencillo añadir una imagen al comando.

Después de modificar el título del comando, asignamos un puntero a evento en su propiedad *OnClick*: la dirección del método *MenuClicked* que hemos definido dentro de nuestra propia clase de extensión, y que examinaremos en breve. Y entonces llega el momento más importante:

- Utilizamos el operador **as** para obtener un puntero a *INTServices* a partir de la variable global *BorlandIDEServices*. Tenemos la seguridad de que la conversión de tipos no fallará; por eso no encerramos la instrucción dentro de un **try/except**.
- Por medio de la propiedad *MainMenu* de la interfaz obtenida, ganamos el acceso al componente del menú principal del Entorno de Delphi. Para insertar nuestro comando de menú, he elegido arbitrariamente el submenú *Components*, que es el séptimo de la barra. Por consiguiente, hacemos referencia al subcomponente de la propiedad *Items* del menú que está asociado al índice 6 (recuerde que se cuenta

con el cero como origen). De acuerdo, acepto el regaño: podía haber utilizado alguna técnica más segura que localizar el menú por su posición. Pero no valía la pena complicar más las cosas.

- Para terminar, insertamos el nuevo ítem de menú. *Insert* y *Count* son un método y una propiedad del *TMenuItem* al que hacemos referencia con la instrucción **with**: el mismo que corresponde *Components*. Quiere decir que insertamos el nuevo comando al final de ese submenú.

Para hacerlo bien, tenemos que eliminar el comando añadido cuando alguien descarga o elimina nuestra extensión del Entorno. En caso contrario, quedaría la entrada en el menú, y se produciría una sonora excepción al intentar ejecutarla, pues su evento *OnClick* estaría apuntando a una zona de memoria liberada. Basta con destruir explícitamente el comando de menú para que desaparezca del submenú donde lo hemos insertado:

```
destructor TimExperto.Destroy;
begin
    FMenuItem.Free;
    inherited Destroy;
end;
```

Si como dije antes, hubiera utilizado un módulo de datos como clase base, podría haber creado el *TMenuItem* pasando el módulo como *Owner* o propietario en el parámetro correspondiente del constructor. Al hacer esto, nos ahorraríamos liberar de manera explícita el objeto creado, porque el módulo se encargaría automáticamente del sucio asunto. También podríamos añadir sobre el módulo, con la ayuda del ratón, componentes para la acción, para los mapas de bits que utilizaríamos en el comando...

Finalmente, tenemos que programar la respuesta al evento *OnClick*, que se disparará cuando el usuario/programador seleccione el nuevo comando de menú:

```
procedure TimExperto.MenuClicked(Sender: TObject);
var
    Module: IOTAModule;
    Project: IOTAProject;
begin
    Module :=
        (BorlandIDEServices as IOTAModuleServices).CurrentModule;
    if Module <> nil then
    begin
        if Module.QueryInterface(IOTAProject, Project) <> S_OK then
            if Module.OwnerCount <> 0 then
                Project := Module.Owners[0];
            if Project <> nil then
            begin
                ChDir(ExtractFilePath(Project.FileName));
                Exit;
            end;
        end;
        raise Exception.Create('No hay proyecto activo');
    end;
```

El método simplemente cambia el directorio activo del sistema para que coincide con el directorio donde reside el proyecto... si es que hay un proyecto activo, claro. Recuerde lo molesto que es en Delphi cambiar de directorio temporalmente y añadir un nuevo fichero al proyecto activo: tenemos que regresar laboriosamente al directorio original.

Para averiguar el nombre del directorio activo, se ejecutan los siguientes pasos:

- Primero se obtiene el *módulo* activo extrayendo la interfaz *INTModuleServices* de la variable global *BorlandIDEServices* y llamando a su método *CurrentModule*. Para Borland, un módulo es una unidad abierta en el Editor de Código, o el formulario en tiempo de diseño, un módulo de datos, etc.
- Como puede que el módulo activo sea el propio fichero del proyecto, llamamos a *QueryInterface* para ver si el *IOTAModule* obtenido se refiere a un objeto que implementa también *IOTAProject*. Si hay suerte, ya tendríamos el proyecto en la mano.
- En caso contrario, buscaríamos el módulo que es propietario del módulo activo, que en ese caso sería un proyecto, con absoluta seguridad. Delphi permite que un fichero forme parte de dos proyectos abiertos diferentes; esto es posible cuando los dos proyectos, a su vez, forman parte de un grupo de proyectos (*bpr*), o cuando hay un proyecto activo simultáneamente con un paquete, y hemos abierto una unidad incluida en ambos. Por eso es que la interfaz *IOTAModule* utiliza *OwnerCount* y una propiedad vectorial, *Owners*, para enumerar todos los posibles propietarios del módulo. Nosotros, claro está, nos quedamos con el primero de los propietarios, si es que hay alguno.
- Si hemos podido localizar un proyecto, averiguamos su nombre mediante la propiedad *FileName* de *IOTAProject*, y con la ayuda de la función *ExtractFilePath*, de la unidad *SysUtils*, aislamos la parte del directorio para llamar, ¡al fin!, al veterano procedimiento *ChDir*, de aquellos buenos tiempos del Turbo Pascal.

## Interfaces y eventos

Hay una relación importante entre la implementación de interfaces y el manejo de eventos, en el sentido que Delphi les da a estos últimos. Recordemos: ¿para qué nos sirven los eventos en lenguajes como Delphi, C++ Builder o Visual Basic? Principalmente, para establecer la comunicación inversa entre el sistema que intentamos controlar y los objetos que creamos.

Sé que eso del “sistema que intentamos controlar” suena pomposo, pero es que necesitamos abstraernos un poco, por el momento, de los detalles concretos. El mencionado “sistema”, por ejemplo, puede ser el propio sistema operativo, o el sistema operativo *más* la capa formada por las clases y funciones suministradas por un *runtime*, como sucede en Delphi. Consideremos al “sistema” como una caja negra: no sabemos lo que hay en su interior. Pero hay varios botones en su superficie, aparte de

algunas bombillas y aparatos de medición, y un buen manual de usuarios que explica claramente qué sucede cuando se pulsa cada botón<sup>3</sup>.

En la programación más tradicional, nuestro cometido consiste en pulsar botones. Bueno, en pulsarlos y mirar con cautela los indicadores luminosos, para comprobar que el efecto es el deseado. Pero la única entidad “viva”, con iniciativa, somos nosotros mismos. Mantenemos una conversión unidireccional con la caja negra. Preste atención al detalle psiquiátrico: estoy identificándome con el programa.

Cuando añadimos la posibilidad de disparar y manejar eventos, es como si la caja negra cobrase vida, y fuera capaz de conversar con nosotros por iniciativa suya. Estamos sentados tan tranquilos en la habitación, cuando el maldito aparato comienza a encender y apagar luces, y a chillar por un altavoz: “¡hay un usuario moviendo el ratón, hay un usuario moviendo el ratón!”. Y se supone que debemos hacer algo para que el artefacto se tranquilice.

Cuando programamos en el entorno RAD de Delphi, el “sistema” nos habla disparando eventos. Si yo fuese un formulario, en mi curriculum vitae habría un párrafo como el siguiente:

*“Dispongo de un evento OnCloseQuery al que podéis llamar cuando vayáis a cerrar el chiringuito, y así sabréis si estoy listo o no. Por favor, avisadme también cuando sepáis que el cierre es definitivo llamando a mi evento OnClose, para recoger mi cepillo dental, las monedas del cambio de la máquina de café y mi calendario Playboy”*

Y así hasta completar los 34 eventos que tiene un formulario en Delphi 6.

Las interfaces nos proporcionan una alternativa. Podríamos definir un tipo como el siguiente:

```
type
  IEventosFormulario = interface
    // Omitiré el GUID
    procedure OnClose(Sender: TObject;
      var Action: TCloseAction);
    procedure OnCloseQuery(Sender: TObject;
      var CanClose: Boolean);
  end;
```

Entonces implementaría esa interfaz, y modificaría la redacción del curriculum:

*“Dispongo de un puntero de interfaz IEventosFormulario. Por favor, si ocurre algo importante, ejecutad el método apropiado”*

---

<sup>3</sup> Sé que lo último es demasiada fantasía.

En realidad, estaríamos pidiendo demasiado: estaríamos exigiendo que el “sistema” sepa de antemano cuál es la especificación de *IEventosFormulario*. Por otra parte, además del formulario, la caja negra tiene que tratar con módulos, botones, cuadros de edición y el resto de la tribu. Si tuviese que complacer a cada uno por separado, nuestro “sistema” reventaría...

Por lo tanto, la verdadera alternativa sería que la caja negra definiese interfaces para las situaciones más frecuentes. Por ejemplo:

```
type
  IEventosRaton = interface
    // Otra vez sin GUID ...
    procedure MouseMove(Sender: TObject; Shift: TShiftState;
      X, Y: Integer);
    procedure MouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure MouseUp(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
  end;
```

Para cada objeto que quiera entrar en el juego, le advertiría en la primera entrevista:

*“Mira, en esta hoja tienes la descripción de IEventosRaton.  
Si quieres estar enterado de los movimientos del ratón, dame un puntero  
a esa interfaz y ya me encargaré de utilizarlo. Y si no quieres enterarte,  
me importa un rábano”*

Resumiendo: las interfaces pueden utilizarse como un mecanismo alternativo a los punteros de métodos que utilizan los eventos de Delphi. De hecho, es el sistema que se utiliza en Java, que no permite punteros explícitos, para el tratamiento de eventos.

¿Cuál sistema es mejor, el de Delphi con los punteros a métodos, o el de Java, con punteros a interfaces? Mientras no sepamos cómo se representan los punteros de interfaz en Delphi, no estaremos en condiciones de poder opinar sobre la eficiencia de las dos técnicas. Pero puedo adelantar un poco de información: el disparo de eventos al estilo Delphi es inconmensurablemente más rápido que el equivalente en Java. En cuanto al espacio ocupado, por el contrario, los punteros a interfaces pueden ocupar menos memoria que el total de eventos equivalentes, si es que se utilizan las interfaces de forma razonable. Sin embargo, Java suele también fastidiar el consumo de memoria al utilizar la técnica de *adaptadores de eventos*.

A pesar de todo, hay que reconocer dos ventajas importantes al tratamiento de eventos mediante interfaces:

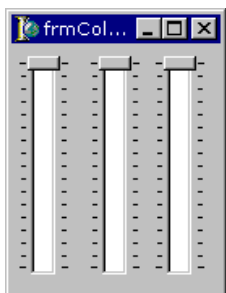
- 1 Los sistemas de disparo de eventos se pueden clasificar en dos grupos: los sistemas *unicast* y *multicast*. En el primero, el productor del evento solamente puede enviar la notificación a un solo cliente o componente; es lo que sucede en Delphi. En un sistema *multicast*, el emisor guarda una lista de componentes interesados y envía la notificación a todos ellos. Las interfaces facilitan la implementa-

ción del *multicasting*, porque el emisor sólo necesita almacenar un puntero de interfaz por cliente.

- 2 Las interfaces están mejor dotadas para el manejo de eventos generados por un productor remoto. Esto lo veremos al estudiar el tratamiento de eventos en COM.

## Marcos e interfaces

Por supuesto, no vamos a lanzarnos ahora a crear nuevos componentes que utilicen interfaces en vez de punteros a métodos, solamente por probar la novedad. Pero Delphi nos ofrece una oportunidad para capturar eventos mediante interfaces cuando diseñamos *marcos*, o *frames*, según como prefiera llamarles.



No tengo deseos de escribir una frase erudita para explicarle qué es un marco; mejor creamos uno. Cree un nuevo proyecto de aplicación y ejecute el comando de menú *File|New|Frame*. Aparecerá una ventana de diseño muy parecida a un formulario; si va al código fuente, sin embargo, comprobará que descende de una clase llamada *TFrame*, en vez de derivarse del conocido *TForm*. Cambie el nombre del marco y llámelo *frmColores*. Añada tres componentes *TTrackBar*, y para cada uno de ellos modifique *Max* a 256, *Orientation* a *trVertical* y *Frequency* a 16. Renombre las barras, y llámelas *Roj*o, *Verde* y *Azul*, empezando si lo desea desde la izquierda. Finalmente mueva los componentes y modifique el tamaño del marco hasta que tenga el aspecto mostrado en la imagen adyacente.



Ahora que ya tenemos un marco, veamos cómo se utiliza. Regrese a la ventana principal de la aplicación, asegúrese de tener seleccionada la primera página de la Paleta de Componentes de Delphi, y haga clic sobre el primer componente de esa página... sí, ese cuya ayuda dice *Frames*. Haga clic una vez más, esta vez sobre la superficie del formulario, en el lugar donde desea poner el marco. Debe aparecer entonces un diálogo para que elija alguno de los marcos definidos dentro del proyecto. Sólo tenemos uno, al menos de momento: selecciónelo y cierre el cuadro de diálogo. Traiga entonces un panel y póngalo a la derecha del marco. Por último, arregle los componentes y el formulario hasta que tenga el aspecto de la imagen anterior.



Al diseñar el marco hemos creado una especie de control visual compuesto, que podemos reutilizar cuantas veces queramos en otros formularios, e incluso dentro de otros marcos. O podríamos crear varias instancias de un mismo marco dentro de un mismo formulario. Examine la declaración de la clase del formulario dentro del Editor de Delphi:

```
type
  TwndPrincipal = class(TForm)
    frmColores1: TfrmColores;
    Panel1: TPanel;
    // ... etcétera ...
  end;
```

Note que en la clase del formulario no aparecen referencias directas a cada uno de los controles del interior del marco, sino una sola referencia a la clase del marco. En definitiva, es el propio marco el encargado de leer las propiedades de esos controles internos para su inicialización, durante la creación del formulario.

Naturalmente, lo que pretendemos es que cambie el color del panel cuando se muevan las barras. El algoritmo necesario se divide en dos partes:

- 1    Calculamos el color a partir de las posiciones de las barras.
- 2    Asignamos el color resultante al panel.

Claro, el primer paso se resuelve con una llamada a la función *RGB* del API de Windows, y el segundo con una asignación en la propiedad *Color* del panel, pero vamos a exagerar deliberadamente la complejidad. El problema que se plantea es dónde implementar ese algoritmo. Es evidente que el primer paso irá mejor dentro del marco, porque necesita consultar propiedades de las barras. Pero el segundo paso quedará mejor dentro de la ventana principal, porque el panel es propiedad suya. La solución salomónica será que el marco realice el cálculo del color, y que *avise* al formulario que lo contiene cuando éste cambie. Recuerde que un marco se puede incluir dentro de varios formularios.

Vaya a la ventana de diseño del marco y salte a su código fuente. Inserte la siguiente definición de tipo entre la palabra reservada **type** y la declaración de la clase:

```
type
  IMuestraColor = interface
    ['{D4EB5760-9679-11D5-BA53-00C026263861}']
    procedure NuevoColor(C: TColor);
    procedure ColorGris;
  end;
```

¿El GUID? Le permito que copie el mío, si se atreve, pero es mejor que pulse la combinación de teclas CTRL+SHIFT+G, y obtenga su propio valor.

El próximo paso es seleccionar simultáneamente las tres barras que están dentro del marco y crear un manejador compartido para el evento *OnChange*.

```

procedure TfrmColores.CambioCompartido(Sender: TObject);
var
    MuestraColor: IMuestraColor;
begin
    if Supports(Owner, IMuestraColor, MuestraColor) then
        begin
            MuestraColor.NuevoColor(
                RGB(Rojo.Position, Verde.Position, Azul.Position));
            if (Rojo.Position = Verde.Position) and
                (Verde.Position = Azul.Position) then
                MuestraColor.ColorGris;
        end;
    end;

```

Es la primera vez que utilizo la función *Supports*, definida en la unidad *SysUtils*. Su propósito es encapsular la llamada a *GetInterface*, y verificar por nosotros que el puntero al objeto no esté vacío. Además, ofrece varias versiones por sobrecarga que permiten pasar indistintamente referencias a objetos, punteros a interfaces e incluso referencias a clases.

El manejador compartido de *OnChange* averigua primero si el *owner*, o propietario, del marco es una clase que implementa la interfaz *IMuestraColor*. Si el objeto propietario no lo hace, nos largamos del método sin más. Pero en caso contrario, calculamos el valor del color resultante y llamamos al método *NuevoColor* del puntero a interfaz obtenido gracias a *Supports*. De paso, he añadido otro “evento” que se dispara cuando las tres barras tienen el mismo valor y generan un tono de gris.

Volvemos a la ventana principal. Tenemos que modificar la declaración de la clase para que implemente *IMuestraColor*:

```

type
    TwndPrincipal = class(TForm, IMuestraColor)
        frmColores1: TfrmColores;
        Panel1: TPanel;
    private
        procedure NuevoColor(C: TColor);
        procedure ColorGris;
    end;

```

El código de los métodos de respuesta será el siguiente:

```

procedure TwndPrincipal.NuevoColor(C: TColor);
begin
    Panel1.Color := C;
end;

procedure TwndPrincipal.ColorGris;
begin
    Beep;
end;

```

Reconozco que estoy matando gorriones a cañonazos en este ejemplo. Pero lo importante es que comprenda cómo se pueden utilizar los tipos de interfaz para lograr la colaboración entre un marco y el formulario que lo contiene.

## Un repaso a las VMTs

Antes de adentrarnos en los detalles de la representación de interfaces, conviene que refresquemos nuestros conocimientos sobre la implementación de objetos y clases en Delphi. Recordemos una de las consecuencias más importantes del polimorfismo:

*“Una variable declarada con el tipo de clase TX puede apuntar, en tiempo de ejecución, a un objeto de la clase TX propiamente hablando, o a un objeto de cualquier clase TY que descienda directa o indirectamente de la clase TX”*

El enunciado anterior es muy básico<sup>4</sup>, pero a efectos prácticos impone una restricción importante en la representación de objetos. Observe la siguiente instrucción, por ejemplo:

```
ShowMessage(objX.AtributoCadena);
```

Supondremos que *objX* es una variable declarada como perteneciente a la clase TX, y que *AtributoCadena* es precisamente un atributo o variable de tipo **string**. De acuerdo al enunciado anterior, la instrucción anterior debe “tener sentido” tanto si el objeto asociado a *objX* pertenece a TX como a cualquiera de sus descendientes. En este contexto, “tener sentido” quiere decir que el código ensamblador generado por el compilador debe funcionar del mismo modo con cualquier objeto al que la variable pueda hacer referencia. La forma de lograrlo más sencilla consiste en obligar a que, en todas las clases descendientes de TX, la variable *AtributoCadena* caiga en la misma posición respecto al inicio del objeto:

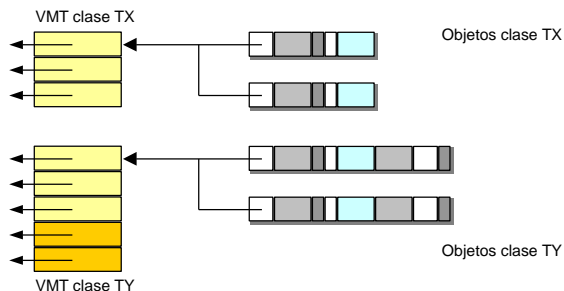


En realidad, como se puede inferir del diagrama anterior, la representación de un objeto de la clase TX siempre estará presente al inicio de la representación de un objeto perteneciente a cualquier clase que descienda de TX.

Estoy hablando de la implementación más sencilla. Una alternativa posible sería utilizar una doble indirección para llegar a la zona de almacenamiento de los atributos; incluso C++ utiliza una variante de esa representación en algunos casos. Evidentemente, es una técnica poco eficiente, aunque muy flexible.

<sup>4</sup> En Delphi no hay excepciones a esta regla, pero sí las hay en C++, en relación con el uso de la herencia “pública” o “privada”.

La principal dificultad consiste en la implementación de las llamadas a métodos virtuales. Voy a mostrar directamente la solución casi universal empleada por los lenguajes con herencia simple, como Delphi y Java:



Todos los objetos reservan los primeros cuatro bytes de su representación para un puntero a una estructura conocida como *Descriptor de Clase*. Este descriptor, a su vez, contiene una subestructura llamada *Tabla de Métodos Virtuales*, o *VMT*, según sus siglas en inglés. He simplificado el diagrama, para que los objetos aparezcan apuntando directamente a sus VMTs, pero es una simplificación sin importancia.

Lo verdaderamente importante es que todos los objetos de una misma clase comparten el mismo Descriptor de Clase y, por consiguiente, la misma VMT. Por lo tanto, puedo saber si dos objetos, de los que desconozco su tipo exacto, pertenecen a la misma clase o no tomando los cuatro primeros bytes de sus cabeceras y comparándolos. Si las direcciones son distintas, pertenecen a clases diferentes, y viceversa.

Otra cosa sería saber si un objeto dado pertenece a una clase derivada de una clase dada. Pero también lo resolveríamos examinando el Descriptor de Clase, porque en éste Delphi incluye un puntero al descriptor de su clase base. Solamente hay que seguir la cadena de punteros.

Cada VMT se representa como un vector (*array*) de punteros a código ejecutable, y cada uno de estos punteros ocupa 4 bytes. En realidad, el puntero a la VMT que mantiene el objeto no apunta al principio de la VMT, sino a un punto cercano al principio. De esta manera, cuando derivamos una clase a partir de *TObject* y definimos el primer método virtual, éste ocupa la primera entrada a partir del punto de referencia. Supongamos que tenemos la siguiente clase:

```
type
  TPrueba = class(TObject)
    procedure Prueba1; virtual;
    procedure Prueba2; virtual;
  end;
```

Entonces el puntero al código de *Prueba1* se encuentra en la primera entrada de la VMT, y *Prueba2* en la segunda. Los métodos virtuales predefinidos por *TObject* se sitúan en entradas “negativas”. Una instrucción como la siguiente:

```
FPrueba.Prueba1;
```

se compila de la siguiente forma, suponiendo que *FPrueba* es una variable local:

```
mov eax, FPrueba      // Leer el puntero al objeto
mov edx, [ax]         // Leer el puntero a la VMT
call dword ptr [edx]  // Llamar a la función de la entrada 0
```

En cambio, una llamada a *Prueba2* se traduciría así:

```
mov eax, FPrueba
mov edx, [ax]
call dword ptr [edx+4] // La entrada siguiente en la VMT
```

Pero el detalle más importante es que, en todas las clases derivadas de *TPrueba*, la entrada de la VMT asociada a *Prueba1* será la primera, y que cuatro bytes más adelante se encuentra el puntero a la versión de *Prueba2* de la clase real a la que pertenece el objeto.

## Representación de interfaces

Veremos ahora que la implementación de llamadas a través de punteros de interfaz se basa en una variante de las VMTs. ¿Qué pasa cuando declaramos variables de interfaz?

```
type
  IAparatoElectrico = interface
    function VoltajeNecesario: Double;
    function PotenciaNominal: Double;
    function GetEncendido: Boolean;
    procedure SetEncendido(Valor: Boolean);
  end;
```

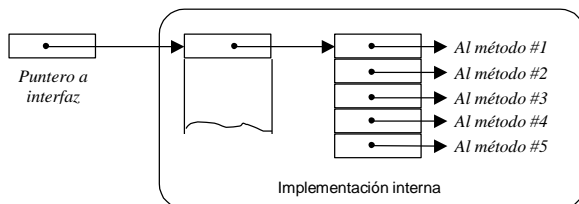
Supongamos que tenemos una declaración como ésta:

```
var
  MiOrdenador: IAparatoElectrico;
```

Una variable de interfaz se representa de forma similar a los punteros tradicionales a objetos: el compilador reserva cuatro bytes para un puntero. Lo mismo que con las variables de tipo clase, esta variable puede estar apuntando a *algo*, o tener el valor especial **nil**, el puntero vacío de Delphi.

La siguiente imagen muestra lo que ocurre cuando un puntero de interfaz apunta a *algo*. Comencemos por el lado izquierdo del gráfico. De la zona de memoria a la que apunta directamente la variable, solamente podemos garantizar el formato y contenido de los cuatro primeros bytes. Entiéndalo literalmente: no estoy diciendo que “ahora no nos interesa el resto”, sino que normalmente no sabremos qué es lo que hay del quinto byte en adelante... si es que hay algo. Más adelante, en este mismo

capítulo y en el siguiente, veremos que esa incertidumbre es la clave para independizarnos del formato físico con que ha sido implementado un objeto.



Esos cuatro bytes garantizados apuntan, a su vez, a otra estructura. No se preocupe, esta vez sabemos qué hay en ella: es un vector con punteros al segmento de código, similar a una VMT. Por supuesto, las rutinas a las que se refiere este vector corresponden a los métodos definidos por la interfaz. Por ejemplo, si nuestro puntero es de tipo *LAparatoElectrico*, la cuarta entrada del vector debe apuntar a “una implementación” de la función *VoltajeNecesario*, la quinta apuntará a *PotenciaNominal*, y así sucesivamente.

Supongamos que se asigna un valor no nulo a *MiOrdenador*, el puntero de interfaz. Podremos entonces ejecutar instrucciones como la siguiente:

```
MiOrdenador.SetEncendido(True);
```

Si conoce algo de lenguaje ensamblador, aquí tiene la forma en que Delphi 6 compila la llamada al método de la interfaz:

```
mov dl, $01
mov eax, MiOrdenador
mov ecx, [eax]
call dword ptr [ecx+$18]
```

El convenio de llamado de *SetEncendido* es **register**, por lo que tanto el parámetro como el puntero de interfaz se pasan dentro de registros. El parámetro, en el registro DL, de 8 bits, y el puntero a interfaz en el registro EAX, de 32 bits. En la tercera instrucción se mueve al ECX los cuatro primeros bytes a los que apunta EAX, es decir, se carga en ECX el inicio de la VMT. A partir de esa dirección se calcula la séptima entrada, o la sexta si se cuenta desde cero, y se realiza la llamada a la dirección contenida en la misma.

Pero no espere encontrar el código de la implementación de *SetEncendido* en la nueva dirección. Esto es lo que se ejecutará:

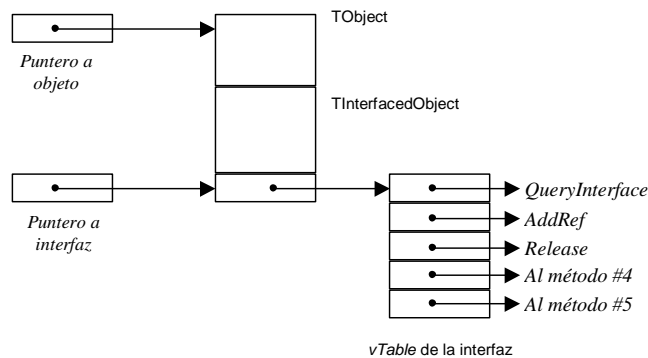
```
add eax, -$10
jmp T AparatoElectrico.SetEncendido
```

Para explicar por qué Delphi resta 16 bytes al puntero de interfaz, debemos precisar cuál es el formato exacto de la clase que ha implementado la interfaz. Para nuestro ejemplo, he utilizado la siguiente declaración:

```

type
  T AparatoElectrico = class(TInterfacedObject, I AparatoElectrico)
  private
    // ...
  protected
    function VoltajeNecesario: Double;
    function PotenciaNominal: Double;
    function GetEncendido: Boolean;
    procedure SetEncendido(Value: Boolean);
  end;

```



Como he intentado mostrar en la imagen anterior, cuando se define una clase y se menciona alguna interfaz en su lista de implementaciones, Delphi añade automáticamente un atributo de tipo puntero dentro del formato de sus instancias. En particular, el atributo correspondiente a *IAparatoElectrico* dentro de los objetos de la clase *TAparatoElectrico* se encuentra 16 bytes a partir del inicio de cada instancia. Nos ha salido un número “redondo”, pero el desplazamiento depende de los atributos e interfaces implementadas por la clase base; en nuestro caso, *TInterfacedObject*. Si utilizáramos alguna otra clase base, es casi seguro que el desplazamiento sería mayor.

Para comprender lo que sucede, echemos un vistazo a la instrucción donde asignamos un valor al puntero de interfaz *MiOrdenador*:

```

MiOrdenador := T AparatoElectrico.Create;

```

Analice su traducción, con algunos retoques insignificantes, a lenguaje ensamblador:

```

mov  dl, $01                // Vamos a reservar memoria
mov  eax, T AparatoElectrico // El descriptor de la clase
call TObject.Create         // ¡Hemos heredado el constructor!
mov  edx, eax               // Copiamos el puntero al nuevo objeto
test edx, edx               // Comprobamos si es nil para ...
jz   +$03                   // ... saltarnos una instrucción
sub  edx, -$10              // ¡¡¡Sumamos 16 bytes!!!

```

```
lea  eax, MiOrdenador      // Copiamos el puntero, actualizando ...
call @IntfCopy             // ... los contadores de referencias
```

En otras palabras, cuando convertimos un puntero a un objeto de tipo *TAparatoElectrico* en un puntero a la interfaz *LAparatoElectrico*, debemos desplazar el puntero original 16 bytes: ¡justo para que apunte al atributo oculto introducido por Delphi! De este modo, cualquier instrucción que llame a un método de la interfaz puede preocuparse acerca de la clase concreta a la que pertenece el objeto del que se ha extraído el puntero a interfaz.

Claro, la implementación de *SetEncendido*, y de cualquier otro método de *TAparatoElectrico*, espera que le pasemos de forma oculta un puntero *al objeto*, no a la interfaz. Por ese motivo es que se ajusta nuevamente el puntero de interfaz al llamar a cualquiera de sus métodos; Delphi vuelve a restarle los 16 bytes específicos de la implementación de *TAparatoElectrico*.

## Implementación de interfaces por delegación

Resumamos lo que hemos aprendido sobre los tipos de interfaz: tenemos un nuevo recurso que enriquece las técnicas ya existentes de la programación orientada a objetos. Su principal objetivo es reducir los requisitos que se le exigen a un componente para que pueda ser “enchufado” dentro de un sistema de módulos complejo. En la POO basada únicamente en clases exigíamos que el componente fuera un descendiente de tal o más cuál clase. Ahora sólo necesitamos que implemente la interfaz que realmente necesitamos.

Sin embargo, algo se nos ha quedado en el camino. En la POO clásica, basada en clases, cada nueva clase que definíamos por derivación de una clase madre, recibía dos regalos de su progenitora:

- 1 Recibía la misma signatura o prototipo público, lo que le permitía ser utilizada en sustitución de su madre. Si creamos en Delphi una clase derivada de *TStrings*, podemos utilizarla sin más en cualquier algoritmo o componente que necesite un objeto de tipo *TStrings*. A esta característica se le llama, en jerga técnica, *herencia de tipos*, y es precisamente el punto fuerte de las interfaces.
- 2 Pero la nueva clase también heredaba el código concreto desarrollado para su madre. Por ejemplo, un hijo de *TStrings* recibe mucho trabajo adelantado, porque ya tiene implementado métodos como *SaveToStream*, *SaveToFile*, *AddStrings* y muchos más. El nombre de esta característica es *herencia de código*, y es uno de los grandes atractivos de la programación orientada a objetos.

Pues bien, las interfaces por sí mismas no nos ofrecen ningún tipo de *herencia de código*. Usted ha definido inicialmente un tipo de interfaz *LAritmetica*. Después ha creado una clase *TAritmetica*, y se ha esmerado en la implementación de los métodos requeridos por *LAritmetica*. Al pasar unos meses, necesita una nueva clase que implemente una calculadora científica que, además de las operaciones básicas soportadas por



*LAritmetica*, implemente otra interfaz, digamos que *ITrigonometria*. ¿Hay alguna forma de reaprovechar todo el código escrito para *TAritmetica*? ¡Claro que sí!, pero a condición de que la nueva clase sea su heredera. Pero, ¿y si ya existe una clase *TTrigonometria*? ¿Debemos renunciar siempre al código de alguna de las clases?

Por supuesto, podríamos elegir arbitrariamente alguna de las dos clases, derivar una nueva a partir de ella, e implementar la segunda interfaz pasando sus llamadas a un objeto interno de la segunda clase. Por ejemplo:

```
type
  TCalculadoraCientifica = class(TAritmetica, ITrigonometria)
  private
    // En este primer ejemplo, un puntero a clase
    FTrigonometria: TTrigonometria;
  protected
    function Seno(X: Extended): Extended;
    // ...
  end;
```

Ahora usamos explícitamente la delegación para implementar los métodos requeridos por la interfaz:

```
function TCalculadoraCientifica.Seno(X: Extended): Extended;
begin
  Result := FTrigonometria.Seno(X);
end;
```

Esta técnica se conoce como *delegación*, o *delegation*. Pero implementarla de esta forma es demasiado laborioso. Afortunadamente, Delphi nos ofrece una solución más conveniente para reutilizar la clase *TTrigonometria*.

```
type
  TCalculadoraCientifica = class(TCalculadora, ITrigonometria)
  private
    // ;Ahora es un puntero a interfaz!
    FTrigonometria: ITrigonometria;
  protected
    property Trigonometria: ITrigonometria read FTrigonometria
      implements ITrigonometria;
  public
    constructor Create;
  end;
```

Estas son las novedades:

- 1 La clase dice implementar determinada interfaz.
- 2 Existe una propiedad que devuelve un puntero a dicha interfaz. Da lo mismo su sección, o si permite o no la escritura.
- 3 Lo más importante: al declarar la propiedad se utiliza la nueva cláusula **implements**, para aclarar que a través de esta propiedad se implementan los métodos de la interfaz correspondiente en la nueva clase.

De lo único que tenemos que preocuparnos es de inicializar el puntero interno a la interfaz:

```
constructor TCalculadoraCientifica.Create;
begin
    FTrigonometria := TTrigonometria.Create;
end;
```

Encontrará el ejemplo completo en el CD, con el nombre de *Calculadoras*.

## Implementaciones dinámicas

Este es otro ejemplo, un poco más práctico y sofisticado, del uso de la delegación para la implementación de interfaces. Queremos desarrollar una aplicación MDI, en la que algunas ventanas podrán imprimir su contenido. Sin embargo, queremos que la impresión se dispare desde un botón situado en una barra de herramientas global, ubicada en la ventana principal.

Comenzamos definiendo una interfaz como la siguiente:

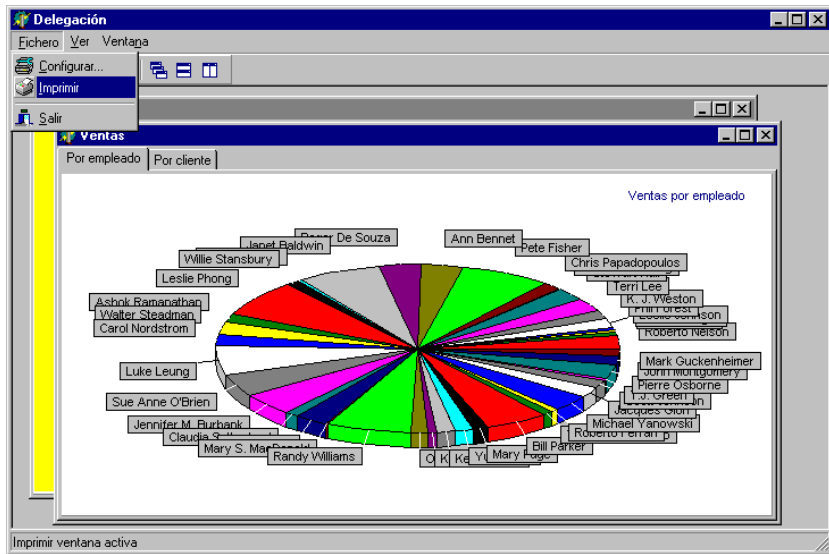
```
type
    IImprimible = interface
        ['{CE78FBA0-F248-11D5-8943-00C026263861}']
        procedure Imprimir;
    end;
```

En la ventana principal nos limitaremos a definir una acción, *FilePrint*. Trataremos sus eventos *OnUpdate* y *OnExecute* del siguiente modo:

```
procedure TwndPrincipal.FilePrintUpdate(Sender: TObject);
begin
    TAction(Sender).Enabled := Supports(ActiveMDIChild, IImprimible);
end;

procedure TwndPrincipal.FilePrintExecute(Sender: TObject);
var
    Imp: IImprimible;
begin
    if Supports(ActiveMDIChild, IImprimible, Imp) then
        Imp.Imprimir;
end;
```

Es decir, el interlocutor del botón es siempre la ventana MDI activa. El botón no rebaja su dignidad para tratar con componentes de menor rango. Sin embargo, esta política puede obligarnos a ser demasiado previsores, o a tener que trabajar demasiado, sobre todo si existen distintos tipos de documentos imprimibles. Vamos a suponer, por ejemplo, que uno de los posibles documentos es un gráfico TeeChart. Sí, ya sé que los gráficos no se tratan sino al final del libro, pero el proyecto del CD se encargará de todos los detalles sucios.



Vamos entonces a añadir un marco o *frame* al proyecto, al que llamaremos *frmGraph*, y hagamos que el marco implemente la interfaz *IImprimible*. Debemos añadir también un componente *DBChart* en su interior:

```
type
  TfrmGraph = class(TFrame, IImprimible)
    DBChart1: TDBChart;
  protected
    procedure Imprimir;
  end;
```

La implementación de imprimir será muy sencilla:

```
procedure TfrmGraph.Imprimir;
begin
  DBChart1.PrintLandscape;
end;
```

Nuestro salto mortal consistirá en implementar un tipo de ventana dividida en páginas, con un marco gráfico en el interior de cada una de ellas. Queremos que al pulsar el botón de imprimir, se imprima solamente el gráfico de la página activa. En tal caso, esta es la declaración de la clase del formulario:

```
type
  TwndVentas = class(TForm, IImprimible)
    // ...
  private
    function GetImprimible: IImprimible;
  protected
    property Imprimible: IImprimible read GetImprimible
      implements IImprimible;
  end;
```

Esta vez hemos implementado la propiedad delegada mediante una función:

```
function TwndVentas.GetImprimible: IImprimible;  
begin  
    if PageControl.ActivePageIndex = 0 then  
        Result := GEmpleados  
    else  
        Result := GClientes;  
end;
```

En dependencia de la página que esté activa, *GetImprimible* devolverá un puntero de interfaz a uno u otro marco. Por lo tanto, el objeto que implementa la interfaz *IImprimible* en el formulario cambiará según la página seleccionada.

#### **NOTA**

Reconozco que estos ejemplos de delegación parecen triviales, de tan sencillos que son. Para reconocer su verdadera utilidad, sin embargo, deberíamos utilizar interfaces “reales”, que suelen contener muchos más métodos, o implementaciones alternativas. Hay pocos ejemplos de esta técnica en el código fuente del propio Delphi... si dejamos aparte a WebSnap, que está basado casi por completo en interfaces. El más sencillo y comprensible se encuentra dentro de la unidad *AxCtrls*: la clase *TConnectionPoints* implementa la interfaz *IConnectionPoints* “por adelantado”. Más adelante, la clase *TActiveXControl* utiliza esta clase auxiliar para implementar la misma interfaz por delegación.

# El Modelo de Objetos Componentes

UNO DE LOS EJEMPLOS MÁS ESPECTACULARES de éxito del concepto de interfaz es el llamado *Modelo de Objetos Componentes*, más conocido por sus siglas en inglés: COM. Casi cualquier área de la programación actual para Windows implica el manejo de clases e interfaces que implementan dicho modelo. De ahí el radical cambio en el orden del contenido de este libro respecto a su anterior edición: necesitaremos dominar esta técnica para trabajar eficientemente con DataSnap, WebSnap, la programación de objetos ASP, el desarrollo de servicios Web ... y estoy convencido de haber dejado algo fuera de la lista.

## Los objetivos del modelo

No voy a contarle otra vez la historia de COM, OLE y otros fósiles. En vez de eso, presentaré el diseño de COM como si hubiera surgido de golpe, a través de un análisis de las necesidades de los programadores. Por supuesto, usted y yo hemos sobrepasado la edad en que se cree en ese tipo de mentirijillas piadosas.

Nuestra meta es simple: queremos desarrollar algún tipo de sistema que nos permita compartir clases y objetos entre diferentes aplicaciones. Dicho así, parece poca cosa: cualquier lenguaje de programación orientado a objetos permite que definamos una clase dentro de un proyecto, y que la reutilicemos en otro, ¿no? Pero note que he hablado de compartir clases... y *objetos*; lo último es más difícil. Delphi permite compartir objetos entre una aplicación y una DLL, pero incluso ha tenido que idear los *packages* o paquetes para que esto no sea un problema. ¿Y compartir memoria entre aplicaciones? Imposible en Windows sin ayuda: el sistema de memoria en modo protegido no lo permite.

Sólo he mencionado el primer objetivo. El segundo: que los objetos puedan ser compartidos aunque los módulos en juego hayan sido desarrollados en lenguajes de programación diferentes. Es verdad que Windows cuenta con las DLLs para compartir código de funciones con independencia del lenguaje, pero de compartir una función a compartir una clase hay una diferencia muy grande.

La principal dificultad está en que los lenguajes deben primero ponerse de acuerdo acerca de qué es una clase. Y no estoy hablando de conceptos filosóficos o simplemente funcionales; me refiero al mismo formato físico. Por ejemplo, en Delphi y Java todos los objetos son accesibles a través de referencia; en cambio, en C++ es posible “incrustar” un objeto directamente en el segmento global de datos, o en la pila, como variable local o parámetro. En C++ existe herencia múltiple, y una de sus variaciones es un engendro conocido como *herencia virtual*, que lamentablemente está muy ligada a la representación física; en Delphi y Java no existe nada remotamente parecido. Observe la siguiente declaración de clases en C++ Builder:

```
class TBase {
public:
    TBase() { prueba(); }
    virtual void prueba() { ShowMessage("Base"); }
};
class TDerivada: public TBase {
public:
    void prueba() { ShowMessage("Derivada"); }
};
```

En la respuesta al evento *OnClick* de un botón programamos lo siguiente:

```
void __fastcall TForm1::Button1Click(TObject *Sender) {
    TBase* derivada = new TDerivada;
    derivada->prueba();
    delete derivada;
}
```

¿Qué mensajes aparecerían en el monitor? Si tiene un C++ a mano podrá verificar que primero aparece *Base* y luego *Derivada*. En contraste, si calcamos este ejemplo en Delphi, veríamos aparecer *Derivada* dos veces.

## Un acuerdo de mínimos

Cuando varias partes en conflicto se sientan en una mesa de negociaciones, saben que la mejor vía para resolver sus problemas es comenzar con un acuerdo de mínimos. ¿Cuál es el modelo mínimo exigible a una clase? Para Microsoft, ese modelo mínimo debe utilizar la semántica que hemos visto asociada al trabajo con tipos de referencias.

En primer lugar, las interfaces solamente definen métodos y propiedades implementadas por entero mediante métodos. Al no utilizar atributos físicos, no se imponen restricciones sobre el formato físico de las instancias de objetos. En segundo lugar, el modelo físico de representación de las interfaces es sumamente sencillo: un puntero a interfaz apunta a otro puntero, que a su vez apunta a una tabla con punteros a funciones. Casi cualquier lenguaje puede definir o manejar una estructura de este tipo. Más adelante, veremos que esta sencillez es crucial para implementar de forma transparente el acceso a objetos remotos.

Pero lo mejor de todo es que esa misma estructura de las interfaces puede servir de “adaptador” para cualquiera de los modelos concretos de clase existentes en los lenguajes de moda.

## Transparencia de la ubicación

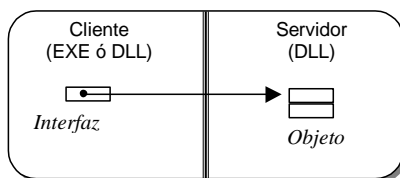
El otro problema grave al que se enfrenta quien decide crear un sistema de compartición de objetos es el de la ubicación de los objetos. Es una dificultad que no se presenta en las DLLs, porque éstas se utilizan sólo para compartir código. Sin embargo, un objeto normalmente tiene datos internos o, como se suele decir, tiene su propio *estado*. Otra cosa es que a ese *estado* no se acceda directamente, sino a través de métodos.

Usted desarrolla una clase, para compartirla generosamente con medio mundo, pero ¿en qué tipo de fichero empaqueta la clase? Lo más sencillo sería utilizar una DLL. El código que hace uso del objeto y el propio objeto residen dentro del mismo espacio de proceso. Pues bien, COM permite este tipo de arquitectura.

Pero quedarnos en este punto nos limitaría demasiado. COM va más allá, y ofrece la posibilidad de que una aplicación utilice objetos COM que residan en *otra* aplicación. Y para rematar, no importa si esa aplicación se está ejecutando en el mismo ordenador, o incluso en un ordenador diferente. Lo mejor de todo: COM logra que el código que trabaja con un objeto remoto sea exactamente igual que el que accede a un objeto situado en una DLL. De este modo, es posible decidir dinámicamente dónde buscar la mejor implementación de una clase.

¿Cómo fue posible esta hazaña? Sabemos que en Windows, y en cualquier otro sistema operativo decente, por motivos de seguridad no se permite que una aplicación pueda referirse directamente al espacio de memoria de otro proceso.

Para entender lo que sucede, veamos primeramente cómo se comunica una aplicación con un objeto COM, cuando este último reside en una DLL:

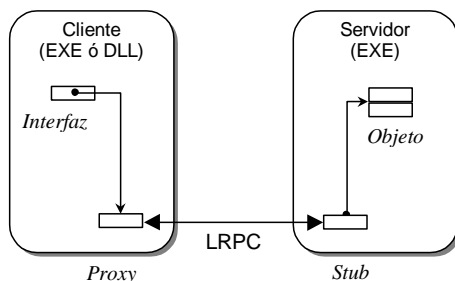


### TERMINOLOGIA

El módulo de software que alberga la definición e implementación de una clase COM recibe el nombre de *servidor* COM. Cuando una aplicación o DLL utiliza un objeto COM, se dice que está actuando como *cliente* COM. Está claro que un servidor puede, en determinado momento, actuar también como cliente.

He simplificado a propósito la estructura del objeto COM en el diagrama. En el lado cliente tenemos un puntero de interfaz, y sabemos que ese puntero hace referencia a un puntero a una tabla de métodos virtuales o *v-table*. Por lo tanto, el cliente trabaja de forma idéntica a como lo haría con un objeto definido por el propio Delphi, con la única restricción que el único tipo de referencias permitido son los punteros de tipo interfaz.

Ahora veamos lo que sucede cuando el objeto reside en un proceso diferente:



En ese caso, Windows se ocupa de crear dos objetos ocultos: uno en el cliente y otro en el servidor. El objeto oculto de la parte cliente recibe el nombre de *proxy*, y tiene la misma estructura “superficial” que el objeto COM, para engañar al cliente y hacerle pensar que está trabajando con el objeto real. Por ejemplo, si el objeto COM implementa una interfaz con cinco métodos, el *proxy* tiene la estructura de una *v-table* también con cinco métodos.

La diferencia consiste en que el *proxy* no duplica la memoria de atributos del objeto simulado, y en la implementación de sus métodos. Cuando el cliente ejecuta uno de esos métodos, pensando que está trabajando con el objeto COM, el *proxy* envía un mensaje al proceso servidor. Dentro de ese mensaje van convenientemente empaquetados la identificación del objeto, del método y de sus parámetros. El protocolo utilizado se llama *Lightweight Remote Procedure Call*, y corresponde a las siglas *LRPC* del diagrama anterior. Pero a un nivel más bajo, el protocolo se implementa sobre el sistema de envíos de mensajes de Windows.

¿Recuerda que también hay un objeto oculto en el lado servidor? Su función es imitar al cliente, y es conocido por el nombre de *stub*. El *stub* está siempre pendiente de los mensajes que el bucle de mensajes (¡sí, el mismo bucle de mensajes que se ocupa del movimiento del ratón y de los quejidos del teclado!) tiene a bien transmitirle. Cuando recibe una petición, desempaqueta los datos asociados y ejecuta el método apropiado en el verdadero objeto COM. Naturalmente, hay que esperar a que el objeto termine de ejecutar el método, para notificarlo al cliente junto con cualquier parámetro de salida, y siguiendo el camino inverso.



**MAS TERMINOLOGIA**

El proceso de empaquetar los datos asociados a la ejecución de un método se conoce como *marshaling*. Este vocablo se utiliza tanto en la literatura sobre COM que siempre que pueda, lo dejaré sin traducir.

Como supondrá, el mismo mecanismo del *proxy* y el *stub* puede utilizarse cuando ambos procesos se ejecutan en máquinas distintas. Sin embargo, eso no era posible en las primeras versiones de COM, y cuando se implementó finalmente recibió el nombre de *Distributed COM*, o DCOM. A nivel físico, la diferencia está en que se utiliza el protocolo *RPC* (*Remote Procedure Calls*) sin el diminutivo *Lightweight* delante. Pero desde el punto de vista del programador, la mayor dificultad reside en las verificaciones de acceso que realiza el sistema operativo.

**PLAN DE TRABAJO**

En este capítulo nos limitaremos a los clientes COM. Explicaré cómo crear objetos a partir de una clase COM, y cómo trabajar con esos objetos. Los servidores serán estudiados en el próximo capítulo.

## Clases, objetos e interfaces

Existen tres conceptos en COM que debemos distinguir con claridad:

### 1 Interfaces

No hay diferencia alguna respecto al tipo de interfaz que Delphi pone a nuestra disposición. Eso sí, toda interfaz utilizada en COM debe ir asociada a un identificador global único.

### 2 Clases

En la Programación Orientada a Objetos, las clases tienen dos responsabilidades: se utilizan para crear objetos, mediante la ejecución de constructores, y sirven para declarar punteros a objetos. En COM, sin embargo, sirven solamente para crear objetos, y no se permiten punteros a objetos, excepto los punteros a interfaces. Cada clase COM también debe ir asociada a un identificador global único. La infame palabreja *co-class* se utiliza a veces para distinguir las clases COM de las demás.

### 3 Objetos

Como acabo de decir, en COM no se permite el acceso a un objeto, excepto a través de sus interfaces.

Ahora puedo explicarle el ciclo de vida de un objeto COM, que difiere bastante del ciclo de un objeto creado internamente por una aplicación escrita en Delphi:

- El objeto se crea siempre a partir de un identificador de clase. Se puede utilizar una función del API, como *CoCreateInstance*, pero es más sencillo recurrir a *CreateCOMObject*, una función de más alto nivel definida por Delphi.

- Estas funciones devuelven, desde el primer momento, solamente un puntero de interfaz de tipo *IUnknown*. Eso significa que *nunca* se nos permite el acceso al “objeto real” que se esconde tras la interfaz.
- Naturalmente, podemos interrogar al objeto para saber si implementa tal o más cual interfaz, y obtener el puntero correspondiente, cuando la respuesta es afirmativa. Para este propósito se utiliza el método *QueryInterface* que implementan todas las interfaces, pero es también posible usar el operador **as** de Delphi.
- Una vez que tenemos en nuestras manos una interfaz del objeto, podemos ejecutar los métodos de la misma, al igual que hemos hecho en los capítulos anteriores.
- Del mismo modo, el objeto muere cuando ya no hay referencias a él.

¿Qué tal si vemos un ejemplo sencillo? Voy a mostrar cómo crear accesos directos (*links*) a aplicaciones o documentos. Es un ejemplo que aparece en casi todos los libros sobre COM, pero hay que reconocer que es muy bueno, y a la vez sencillo. No explicaré ahora todos los detalles, sin embargo. ¡Ah!, probablemente el servidor se encuentra ubicado dentro de una DLL del sistema; de lo que sí puede estar seguro es que se trata de un servidor local:

```

procedure CreateLink(const APath: string; const AFile: WideString);
var
    SL: IShellLink;
    PF: IPersistFile;
begin
    // Crear el objeto, y obtener la interfaz IShellLink
    SL := CreateCOMObject(CLSID_SHELLLINK) as IShellLink;
    OleCheck(SL.SetPath(PChar(APath)));
    // Obtener la interfaz IPersistFile
    PF := SL as IPersistFile;
    OleCheck(PF.Save(PWideChar(AFile), False));
end;

```

Los identificadores *IShellLink* e *IPersistFile* son tipos de interfaces. El primero de ellos ha sido declarado dentro de la unidad *ShlObj*, y el segundo, dentro de *ActiveX*. Tenemos que añadir esas dos unidades, más la unidad *ComObj*, a alguna de las cláusulas **uses** de la unidad donde definamos el procedimiento. En *ComObj* es donde se define la función *CreateCOMObject*, que sirve para crear objetos COM a partir de un identificador de clase. Precisamente, *CLSID\_SHELLLINK* es un identificador global único declarado dentro de *ShlObj* del siguiente modo:

```

const
    CLSID_ShellLink: TGUID = (D1:$00021401; D2:$0000; D3:$0000;
        D4:($C0,$00,$00,$00,$00,$00,$00,$46));

```

*CreateCOMObject* devuelve un puntero de interfaz de tipo *IUnknown*, y lo primero que hacemos es interrogar al objeto que se esconde tras ese anodino puntero para saber si implementa la interfaz *IShellLink*, por medio del operador **as**. En realidad, el compilador de Delphi transforma la operación en una llamada a *QueryInterface*.

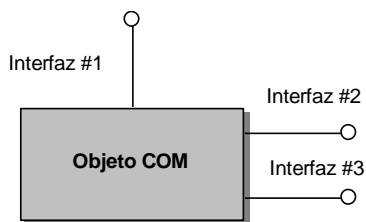
Una vez obtenido el puntero de tipo *IShellLink*, cambiamos algunas variables internas del objeto asociado mediante el método *SetPath* de la interfaz y, acto seguido, volvemos a preguntar por las capacidades del objeto. Esta vez necesitamos un puntero de tipo *IPersistFile*; una vez que lo tenemos, llamamos al método *Save* para que el objeto se grabe en un fichero. Ambas llamadas, se encuentran “envueltas” por una llamada al método *OleCheck*, pero el único papel de éste es vigilar el valor de retorno de *SetPath* y *Save*, para disparar una excepción en caso de error. Por último, observe que no destruimos el objeto de forma explícita. Recuerde que las variables de interfaz utilizan un contador de referencias para destruir automáticamente los objetos inaccesibles.

### ADVERTENCIA

Para poder programar un procedimiento como el anterior, tuve que enterarme de algún modo que los objetos de la clase CLSID\_SHELLLINK implementan las dos interfaces *IShellLink* e *IPersistFile*. Más adelante veremos cómo se puede obtener tal información. Pero quiero disuadirle de buscar alguna función “mágica” del tipo “dime qué interfaces implementa el objeto que tengo en mi mano derecha”. Simplemente, no existe tal.

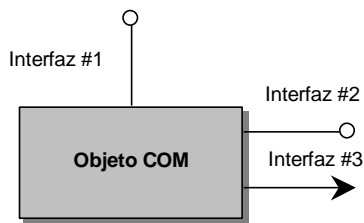
## Convenios gráficos

En la mayoría de los artículos y libros sobre COM, se utiliza un diagrama parecido al siguiente para la representación de los objetos de este modelo:



Creo que la simbología es clara: un objeto COM es una caja negra, a la cual tenemos acceso únicamente a través de un conjunto de interfaces, representadas mediante las líneas terminadas con un círculo. El círculo hueco significa que los clientes pueden “enchufarse” a estas interfaces para llamar los métodos ofrecidos por ellas.

La presencia de los “huecos” no es trivial ni redundante; más adelante veremos cómo los objetos COM pueden implementar interfaces “de salida” (*outgoing interfaces*), que nos servirán para recibir eventos disparados desde otros objetos COM. En tales casos, la interfaz de salida se representa mediante una flecha convencional:



Por el momento, mantendré en secreto el motivo por el que una de las “antenas” de nuestro insecto apunta a las alturas.

## Cómo obtener un objeto

Veamos con más detalle las funciones que permiten crear objetos COM. En el ejemplo anterior utilizamos *CreateComObject*, que Delphi define dentro de la unidad *Com-Obj* del siguiente modo:

```
function CreateComObject(const ClassID: TGUID): IUnknown;
begin
  OleCheck(CoCreateInstance(ClassID, nil,
    CLSCTX_INPROC_SERVER or CLSCTX_LOCAL_SERVER, IUnknown,
    Result));
end;
```

Aquí tenemos dos funciones diferentes: *OleCheck*, que también pertenece a Delphi y *CoCreateInstance* que sí es parte del API de COM, y se encuentra en la unidad *ActiveX*. *OleCheck* convierte el código de retorno de las funciones COM en una excepción, si fuese necesario; más adelante veremos más detalles. Y éste es el prototipo de *CoCreateInstance*:

```
function CoCreateInstance(
  const clsid: TCLSID; unkOuter: IUnknown; dwClsContext: Longint;
  const iid: TIID; out pv): HRESULT; stdcall;
```

Esto es lo que significa cada parámetro:

- *clsid*: Aquí pasamos el identificador de la clase del objeto que deseamos crear.
- *pUnkOuter*: COM ofrece soporte especial para la *agregación* de objetos. Un objeto “externo” pueda administrar el tiempo de vida de uno o más objetos “internos”, a la vez que permite al cliente trabajar directamente con punteros a interfaces de los subobjetos.
- *dwClsContext*: Indica qué tipo de servidor deseamos utilizar. Como he explicado, un servidor puede implementarse mediante un DLL, una aplicación local o una remota.
- *riid*: Una vez creado el objeto, se busca determinada interfaz en su interior, para devolver el puntero a la misma. Este parámetro sirve para identificar dicha interfaz. En la mayoría de los casos se utiliza *IID\_IUnknown*.

- *ppv*: Es un puntero a un puntero a una interfaz que sirve para depositar el puntero a la interfaz localizada por *CoCreateInstance*.

Por lo tanto, analizando la implementación de *CreateComObject* vemos que intenta crear un objeto de la clase especificada en su único parámetro, que espera que la implementación de la clase sea local, no importa si en un ejecutable o en una DLL, y que pide una interfaz inicial del tipo más general, *IUnknown*. Delphi define otra función, muy parecida, para la creación de objetos:

```
function CreateRemoteComObject(const MachineName: WideString;  
    const ClassID: TGUID): IUnknown;
```

Esta vez COM busca la información de la clase indicada en el registro de otra máquina, crea el objeto basado en la clase en el espacio de direcciones de ese ordenador, y devuelve nuevamente un “puntero” al objeto creado.

Existen otras dos funciones importantes para la creación de objetos COM: *CoCreateInstanceEx*, que permite obtener varias interfaces a la vez durante la creación, y *CoGetObject*. Esta última es útil cuando se quieren crear varias instancias de la misma clase, pero tendremos que esperar al capítulo siguiente para explicar su funcionamiento.

## Detección de errores

Casi todas las funciones del API de COM devuelven como resultado un valor perteneciente al tipo numérico *HResult*, que en Delphi se define así:

```
type  
    HResult = type LongInt;
```

El convenio que sigue COM para los valores de retornos no es tan simple como decir que 0 está bien y el resto es un desastre. Para saber si determinado valor representa o no un error, tendríamos que comprobar el estado del bit más significativo del mismo. Dicho con menos rodeos: si el valor es negativo, hubo problemas, en caso contrario, no hay motivos para quejarse. Pero en vez de comparar directamente el código de retorno, es preferible utilizar estas dos funciones lógicas, definidas dentro de la unidad *ActiveX*:

```
function Succeeded(Res: HResult): Boolean;  
function Failed(Res: HResult): Boolean;
```

En realidad, estas funciones se implementan realmente como macros en C/C++.

Gracias a la existencia de un criterio tan uniforme para la notificación de errores, a Delphi le ha sido posible programar *OleCheck*, un procedimiento que recibe el código de retorno de una función del API de COM, y lanza una excepción si ese valor indica un fallo:

```

procedure OleError(ErrorCode: HRESULT);
begin
    raise EOleSysError.Create('', ErrorCode, 0);
end;

procedure OleCheck(Result: HRESULT);
begin
    if not Succeeded(Result) then OleError(Result);
end;

```

Si vamos a ejecutar por cuenta nuestra cualquier función del API de COM, debemos realizar la comprobación de su valor de retorno mediante *OleCheck*. No obstante, más adelante veremos que para una amplia categoría de interfaces COM, Delphi genera automáticamente el código de lanzamiento de excepciones.

## COM y el Registro de Windows

Para que podamos sacar provecho de los servidores COM, estos deben anunciarse como tales en algún lugar. El tablón de anuncios de Windows es el Registro de Configuraciones. Los datos acerca de servidores se guardan, por regla general, bajo la clave *HKEY\_CLASSES\_ROOT*. En particular, bajo la clave subordinada *CLSID* encontramos todos los identificadores únicos de clases registradas en un ordenador.

Hagamos un experimento: busquemos dentro la unidad *ShlObj* la declaración de la constante *CLSID\_SHELLLINK* que hemos utilizado como ejemplo. Encontraremos lo siguiente:

```

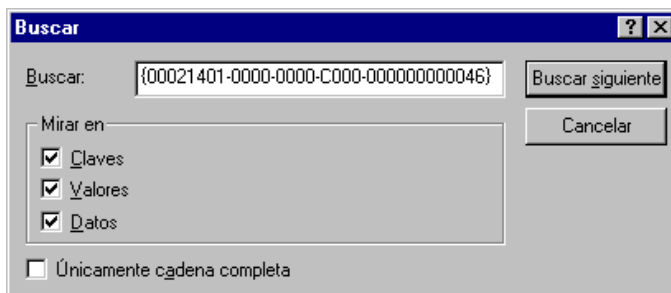
const
    CLSID_ShellLink: TGUID = (D1:$00021401; D2:$0000; D3:$0000;
        D4:($C0,$00,$00,$00,$00,$00,$00,$46));

```

Pero si concatenamos todos los valores hexadecimales y adcentamos un poco el resultado, obtendremos la siguiente cadena de caracteres:

```
{00021401-0000-0000-C000-000000000046}
```

En el Editor del Registro (*regedit.exe*), busque la clave *HKEY\_CLASSES\_ROOT*, localice en su interior la clave *CLSID* y ejecute el comando de menú *Edición* | *Buscar*:



Cuando el programa encuentre la clave que estamos buscando, ejecute el comando de menú de exportación de claves, y asegúrese de que solamente se utilizará la rama seleccionada en el árbol. Este es el resultado, un poco maquillado por culpa de la excesiva longitud de algunas líneas:

```
[HKEY_CLASSES_ROOT\CLSID\{00021401-0000-0000-C000-000000000046}]
@="Acceso directo"
[HKEY_CLASSES_ROOT\CLSID\
{00021401-0000-0000-C000-000000000046}\InProcServer32]
@="shell32.dll"
"ThreadingModel"="Apartment"
[HKEY_CLASSES_ROOT\CLSID\
{00021401-0000-0000-C000-000000000046}\shellex]
[HKEY_CLASSES_ROOT\CLSID\
{00021401-0000-0000-C000-000000000046}\shellex\MayChangeDefaultMenu]
@=""
[HKEY_CLASSES_ROOT\CLSID\
{00021401-0000-0000-C000-000000000046}\ProgID]
@="lnkfile"
```

Bien, tiene cinco segundos para maldecir la hora en que eligió la Informática para ganarse el sustento. ¿Ya está más tranquilo? Le diré lo que nos importa ahora del galimatías anterior explicando qué es lo que hace *CoCreateInstance* para crear un objeto perteneciente a una clase COM:

- El identificador de clase se busca en la clave *CLSID* de la raíz de las clases del registro: *HKEY\_CLASSES\_ROOT*.
- Una vez encontrada la clave, se busca una clave subordinada dentro de la misma rama que nos indique de qué tipo de servidor se trata. Los tipos posibles son: *InprocServer32*, para servidores dentro del proceso, y *LocalServer32*, para servidores locales implementados como ejecutables.
- La clave encontrada contiene como valor predeterminado la ruta a la DLL o al ejecutable que actuará como servidor. Aquí nos detendremos, de momento.

Aquí nos detendremos, de momento, pero reanudaremos la explicación en breve. Lo que me importaba ahora mismo es que viese cómo Windows busca en el registro la información sobre el módulo binario donde se implementa determinada clase COM.

## Identificadores de programa

La mayoría de los desarrolladores que se inician en el mundo de la programación COM perciben como una tragedia el tener que trabajar con identificadores globales de clase. No veo por qué. Cuando usted tenga que crear un objeto de una clase, sencillamente tendrá a su disposición una unidad que contendrá las declaraciones necesarias, incluyendo los identificadores de clase y de interfaz. En caso contrario, tendrá todas las facilidades para generar dicho fichero; se lo prometo, pero tendrá que esperar un poco, hasta ver las bibliotecas de tipo, para comprender el motivo.

No obstante, reconozco que en ocasiones puede ser más cómodo disponer de una representación “legible” del nombre de una clase, que se traduzca unívocamente a un CLSID. En la jerga de COM, a estos nombres se les conoce como *identificadores de programas*, ó *programmatic identifiers*. Las siguientes funciones del API convierten un identificador de programa en un identificador de clase, y viceversa:

```
function ProgIDFromCLSID(const clsid: TCLSID;
    out pszProgID: PChar): HRESULT; stdcall;
function CLSIDFromProgID(pszProgID: PChar;
    out clsid: TCLSID): HRESULT; stdcall;
// ¿Sabéis una cosa? ¡Odio la notación húngara!
```

Pero es más sencillo utilizar estas funciones auxiliares, definidas dentro de *ComObj*:

```
function ProgIDToClassID(const ProgID: string): TGUID;
function ClassIDToProgID(const ClassID: TGUID): string;
```

La asociación entre un identificador de clase y su identificador de programa se almacena también en el registro. En el ejemplo de los accesos directos hemos visto que bajo la clave correspondiente a *CLSID\_SHELLLINK* se almacena el siguiente valor:

```
[HKEY_CLASSES_ROOT\CLSID\{00021401-0000-0000-C000-000000000046}\ProgID]
@="lnkfile"
```

Esta entrada es aprovechada por *ProgIDFromCLSID*, pero si queremos que la conversión en sentido contrario sea eficiente, debemos almacenar entradas redundantes. Directamente dentro de *HKEY\_CLASSES\_ROOT* se almacena una clave con el identificador de programa. Dentro de dicha clave, existe una clave subordinada *CLSID*, que naturalmente contiene el identificador de clase correspondiente. Para comprobarlo, busque la rama que corresponde a *lnkfile*.

## ADVERTENCIA

La clase que estoy utilizando como ejemplo tiene sus majaderías personales. Por ejemplo, su identificador de programa es un simple *lnkfile*, mientras que la mayoría tienen la sintaxis *NombreServidor.NombreClase*, como *Word.Application*. Además, hay más información almacenada en el registro para esta clase que lo habitual. Por último, quiero advertir que la estructura de los identificadores de programas se complica cuando se añaden los números de versión al potaje. Si tiene un Word instalado, puede hurgar un poco en las entrañas del Editor del Registro.

En la aplicación de ejemplo del CD-ROM encontrará que he añadido un botón para obtener el identificador de programa asociado a *CLSID\_SHELLLINK*. La implementación del evento *OnClick* es elemental:

```
procedure TwndPrincipal.bnVerifyClick(Sender: TObject);
begin
    ShowMessage(Format('%s = %s',
        [GUIDToString(CLSID_SHELLLINK),
        ClassIDToProgID(CLSID_SHELLLINK)]));
end;
```



Ante tanta alegre conversión, ¿no sería útil tener una versión de *CreateComObject* que admitiese un identificador de programa legible, en vez de exigir un identificador de clase numérico? Bueno, ya existe la siguiente función dentro de *ComObj*:

```
function CreateOleObject(const ClassName: string): IDispatch;
```

Pero todavía no he explicado qué demonios es un *IDispatch*. Y el *Ole* del nombre de la función parece indicar que no vale para cualquier clase; y así es, efectivamente. Por lo tanto, no nos queda otra alternativa que definir nuestra propia rutina:

```
function CreateObject(const ClassName: string): IDispatch;
var
    ClassID: TCLSID;
begin
    ClassID := ProgIDToClassID(ClassName);
    OleCheck(CoCreateInstance(ClassID, nil,
        CLSCTX_INPROC_SERVER or CLSCTX_LOCAL_SERVER, IUnknown,
        Result));
end;
```

## Fábricas de clases

Antes de retomar la explicación sobre cómo se construyen las instancias de una clase COM, debe responder una pregunta muy importante: ¿qué fue primero, el huevo o la gallina? Cuando se trata de crear objetos, se nos presenta un problema similar. Pongamos como hipótesis que estamos creando un lenguaje de programación. Como somos unos puristas radicales de la programación orientada a objetos, haremos que todo el código que se programe en ese lenguaje deba estar situado dentro de un método de una clase; no queremos funciones o procedimientos “globales”, como sucede en los lenguajes híbridos. Si llevásemos nuestro purismo hasta ese extremo, nos veríamos en un grave problema: ¿cómo podríamos crear un objeto, digamos que de la clase *X*? Mediante métodos de la propia clase *X* sería imposible, porque todo método requiere de un objeto para su ejecución. La única salida que tendríamos sería que los objetos de la clase *X* fuesen creados desde un método de otra clase, digamos que de la clase *Y*. Pero, ¿quién crearía entonces los objetos de la clase *Y*?

En la verdadera Programación Orientada a Objetos no se presenta ese problema porque existen métodos especiales, que conocemos bajo el nombre de *constructores*. Aunque un constructor parece un método, no lo es en sentido estricto, porque no hace falta un objeto para poder ejecutarlo. Esto se ve de forma inmediata en la sintaxis de construcción de instancias del propio Delphi:

```
Form1 := TForm1.Create(Application);
```

Dejemos tranquilo a Delphi y regresemos a COM: ¿qué hacemos, definimos entonces un tipo especial de métodos dentro de las interfaces? Aunque sería posible, no fue la solución que Microsoft le dio al dilema; creo que el motivo fue que al tener dos tipos de métodos, se iba a complicar bastante un modelo que buscaba la mayor sen-

cillez posible. Y aunque parezca absurdo, la solución elegida fue que para crear un objeto... hacía falta otro objeto: la *fábrica de clases* (*class factory*) de nuestra clase COM.

¿Dónde está el truco, dónde se rompe la circularidad? La respuesta es que el mecanismo de construcción de un objeto que actúa como fábrica de clases es diferente. Volvamos a la explicación sobre el funcionamiento de *CoCreateInstance*. Nos habíamos quedado en el momento en que localizábamos el nombre del programa o DLL que contenía la clase que queríamos instanciar. Para que la explicación sea más sencilla, digamos inicialmente que el servidor es una DLL.

Entonces COM carga el código de la DLL en memoria. Pero una de las responsabilidades del desarrollador de la DLL es implementar y exportar una función llamada *DllGetClassObject*, que debe devolver a quien la ejecute un puntero de interfaz de un tipo predefinido por COM. Este es el prototipo de la función:

```
function DllGetClassObject(const CLSID, IID: TGUID;
    var Obj): HRESULT; stdcall;
```

El primer parámetro debe contener el identificador de la clase, y en el segundo debemos pasar el identificador de la interfaz del objeto que queremos obtener. Esto es necesario porque el servidor puede implementar varias clases COM. El programador tiene libertad para decidir si utiliza la misma fábrica de clases para todas las clases que publica, o si prefiere utilizar fábricas distintas.

Cuando el código de *DllGetClassObject* se ejecuta, el desarrollador examina el CLSID que le han pasado, y si corresponde a una de las clases que implementa, debe crear entonces una instancia de la clase que va a actuar como fábrica de clases para *esa* clase en particular. Debe, además, devolver en el parámetro de salida *Obj* un puntero a interfaz perteneciente a uno de los siguientes tipos:

```
type
  IClassFactory = interface(IUnknown)
  ['{00000001-0000-0000-C000-000000000046}']
  function CreateInstance(const unkOuter: IUnknown;
    const iid: TIID; out obj): HRESULT; stdcall;
  function LockServer(fLock: BOOL): HRESULT; stdcall;
end;

  IClassFactory2 = interface(IClassFactory)
  ['{B196B28F-BAB4-101A-B69C-00AA00341D07}']
  function GetLicInfo(var licInfo: TLicInfo): HRESULT; stdcall;
  function RequestLicKey(dwResrved: Longint;
    out bstrKey: WideString): HRESULT; stdcall;
  function CreateInstanceLic(const unkOuter: IUnknown;
    const unkReserved: IUnknown; const iid: TIID;
    const bstrKey: WideString; out vObject): HRESULT; stdcall;
end;
```

La segunda de las interfaces, *IClassFactory2*, se utiliza solamente en los controles ActiveX con control de licencia, por lo que nos ocuparemos sólo de *IClassFactory*.

Es evidente que, una vez que *CoCreateInstance* tiene un puntero *IClassFactory* en su poder, debe ejecutar el método *CreateInstance* para crear entonces el objeto que habíamos solicitado inicialmente.

¿Qué pasa entonces con el objeto que actúa como fábrica de clases? Es destruido automáticamente, porque la referencia a él se almacena en una variable interna de la función *CoCreateInstance*. Puede parecer una estupidez montar toda esta tramoya para obtener un objeto que nunca llegamos a ver. Si lo pensamos, podríamos llegar a la conclusión que el inventor de COM debería haber exigido a la DLL que se encargase directamente de la creación de objetos. Por ejemplo, en vez de pedir que implementase *DllGetClassObject*, podía haber estipulado que la DLL implementase una función como la siguiente:

```
// ;;;Esta función no existe!!!
function ConstructorCOM(const CLSID, IID: TGUID;
    var Obj): HRESULT; stdcall;
```

Pero no le he revelado un pequeño detalle: existe una función en el API de COM que sirve para obtener directamente un puntero a la fábrica de clases:

```
function CoGetClassObject(const clsid: TCLSID;
    dwClsContext: Longint; pvReserved: Pointer;
    const iid: TIID; out pv): HRESULT; stdcall;
```

¡Un momento! ¿Es que no podemos llamar nosotros mismos a *DllGetClassObject*? Pues no, no es recomendable: recuerde que la explicación anterior se basaba en que ya habíamos localizado el servidor donde reside la clase, y que casualmente era una DLL. *CoGetClassObject* nos ahorra la búsqueda en el registro, la carga del servidor, sea éste un ejecutable o una DLL, y finalmente cumple su papel de obtener la fábrica de clase con independencia del formato físico del servidor.

¿Qué sentido tiene obtener un puntero a una fábrica de clases? Muy sencillo: *CoCreateInstance* crea para nosotros un único objeto de la clase que estamos instanciando. Si necesitásemos cincuenta objetos de esa clase, parte de la ejecución de las 49 últimas ejecuciones de la función serían redundantes e innecesarias. Por lo tanto, ahorraríamos mucho tiempo y recursos si obtuviésemos directamente la fábrica de clases, y a partir de ese momento creásemos tantos objetos como quisiéramos, llamando directamente al método *CreateInstance* del puntero a interfaz.



## Servidores dentro del proceso

**N**OS VAMOS DE EXCURSIÓN A LA OTRA orilla: ahora que ya sabemos utilizar clases e interfaces COM, nos interesa aprender a programar los servidores que las implementan. Aunque los detalles de bajo nivel son muchos, aprovecharemos la ayuda que Delphi nos ofrece: el informalmente llamado DAX, un conjunto de clases y procedimientos que facilitan el uso y desarrollo de objetos ActiveX.

### Tipos de servidores

Hemos visto que existen tres modelos diferentes para ubicar la implementación de una clase COM:

- 1 La clase, y sus objetos, reside en una DLL.
- 2 La clase reside en una aplicación local, dentro del mismo ordenador.
- 3 La clase reside en una aplicación remota, en otro ordenador de la red.

Los servidores de objetos implementados como DLLs reciben el nombre en inglés de *in-process servers*, o servidores dentro del proceso. Los controles ActiveX se crean obligatoriamente mediante servidores de este estilo. Son los más eficientes, porque el cliente puede actuar directamente sobre las instancias de la clase, al encontrarse estas en el mismo espacio de memoria del proceso.

#### ADVERTENCIA

La última afirmación no es del todo cierta, porque puede surgir la necesidad de utilizar *proxies* si los modelos de concurrencia del cliente y del servidor no son idénticos. Estos problemas los estudiaremos más adelante.

Como puede imaginar, cuando el servidor es una aplicación ejecutable recibe el nombre simétrico de *out-of-process server*. Lo interesante es que las técnicas para programar un servidor ejecutable local son las mismas que para desarrollar uno remoto. El sistema operativo es el encargado de suministrar la infraestructura para que el acto de magia sea posible. De todos modos, DCOM plantea una serie de interrogantes sobre seguridad y permisos de acceso que encuentran su reflejo en el desarrollo de clases remotas.

¿Es posible ejecutar remotamente una clase que se ha implementado dentro de una DLL? Inicialmente, DCOM solamente consideraba la posibilidad de activación remota para servidores ejecutables. Pero es perfectamente posible utilizar una clase ubicada dentro de una DLL desde otro puesto... siempre que encontremos alguna aplicación que quiera echarnos una mano y cargar la DLL por nosotros. El entorno de COM+, en Windows 2000 y XP, es un ejemplo adecuado. Pero cuando estudiemos los servidores de capa intermedia de DataSnap veremos que existen alternativas más sencillas; eso sí, con algunas limitaciones.

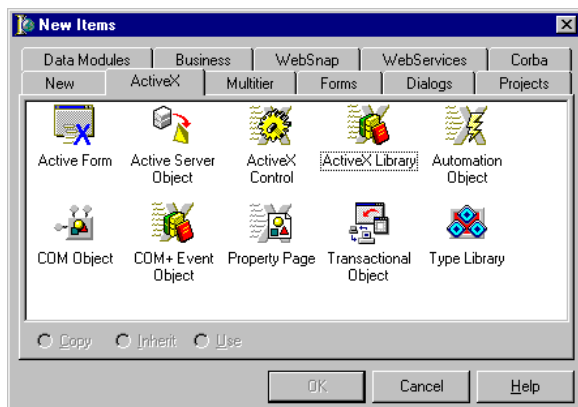
Independientemente del modelo físico concreto utilizado por un servidor COM, hay una lista bien definida de tareas que este servidor debe cumplir:

- 1 Debe poder añadir al registro la información necesaria para que los clientes puedan utilizarlo.
- 2 Naturalmente, debe también ser capaz de eliminar dicha información.
- 3 Debe poner a disposición del sistema, de una forma u otra, los objetos que jugarán el papel de fábricas de clases.
- 4 Finalmente, un servidor COM debe saber en qué momento “sobra” su presencia, es decir, cuándo ya no es necesario que esté cargado en memoria, para suicidarse discretamente.

A lo largo de este capítulo veremos ejemplos de creación de cada uno de los modelos de servidores mencionados.

## Servidores dentro del proceso

Para crear un servidor dentro del proceso, primero necesitamos un proyecto de tipo DLL. Pero no nos vale cualquier DLL, a no ser que deseemos implementar todos los oscuros detalles que son necesarios para el servidor. Lo mejor es llamar al Almacén de Objetos, y en la segunda página (*ActiveX*) hacer doble clic sobre el icono *ActiveX Library*:



Delphi no nos pedirá información alguna, pero cerrará el proyecto que podamos tener abierto en esos momentos y generará un fichero *dpr* con el siguiente contenido:

```
library Project1;

uses
  ComServ;

exports
  DllGetClassObject,
  DllCanUnloadNow,
  DllRegisterServer,
  DllUnregisterServer;

{$R *.RES}

begin
end.
```

La DLL ya exporta cuatro funciones, que al parecer se encuentran definidas e implementadas dentro de la unidad *ComServer*, y que corresponden a las cuatro tareas que mencioné antes que deben ser implementadas por un servidor COM. Sus prototipos son los siguientes:

```
function DllGetClassObject(const CLSID, IID: TGUID;
  var Obj): HRESULT; stdcall;
function DllCanUnloadNow: HRESULT; stdcall;
function DllRegisterServer: HRESULT; stdcall;
function DllUnregisterServer: HRESULT; stdcall;
```

En concreto:

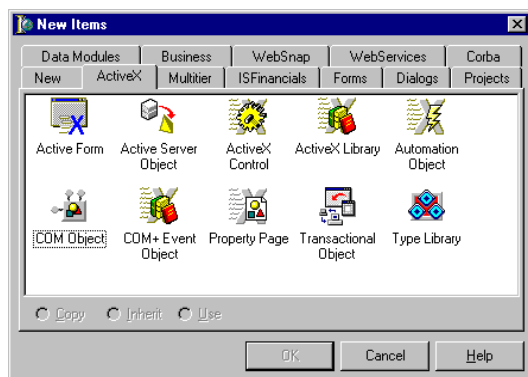
- 1 *DllRegisterServer* y *DllUnregisterServer* son funciones que deben existir en todo servidor DLL para registrarlo o eliminar sus datos del registro.
- 2 *DllGetClassObject* es el mecanismo estándar para que un cliente, o el propio COM, pueda solicitar al servidor la fábrica de clases correspondiente a determinada clase.
- 3 Por último, *DllCanUnloadNow* es llamada de cuando en cuando por COM para saber si puede descargar la DLL de memoria.

Antes de continuar, quiero que guarde el proyecto y le de un nombre que recuerde más adelante; he utilizado *ISFinancials*. Ese nombre es importante porque formará parte del identificador de programa que se le asignará más adelante a las clases COM que definamos dentro de la DLL...

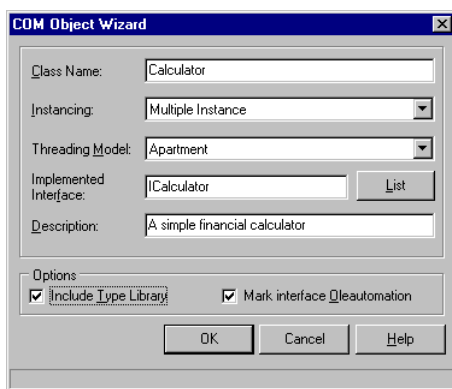
...porque hasta el momento, nuestro proyecto es un cascarón vacío, sin clases COM en su interior. Vamos entonces a crear nuestra primera clase COM. Tenemos dos posibilidades para elegir el objetivo de este ejemplo: crear una clase que implemente alguna interfaz ya definida, sea por el propio Windows o por otro fabricante de software, o inventarnos nuestra propia interfaz. Para intentar que el ejemplo sea lo más simple posible, seguiremos la segunda vía. Vamos a definir una interfaz que nos per-

mita realizar cálculos simples de intereses financieros, y simultáneamente crearemos una clase que implemente dicha interfaz.

Busque nuevamente el Almacén de Objetos, y seleccione también la segunda página:



Podríamos elegir ahora un *Automation Object* o un *COM Object*; en ambos casos crearíamos clases COM, con la misma facilidad. Pero internamente el *Automation Object* tiene una estructura más compleja que el *COM Object* mundo y lirondo. Su ventaja, en cambio, es que puede ser utilizado desde cualquier lenguaje moderno de *script*. Como estudiaremos los objetos de automatización en un capítulo posterior, vamos a hacer doble clic sobre *COM Object*. Este es el asistente que aparece, en respuesta a nuestras plegarias:



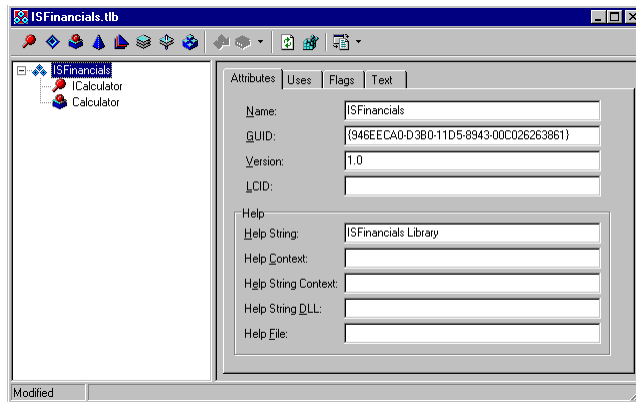
En el cuadro de edición *Class Name* debemos teclear *Calculator*. De esta manera, el identificador de programa para esa clase será *ISFinancials.Calculator*, que es la concatenación del nombre del proyecto con el nombre que hemos dado a la clase. Observe que, por omisión, el propio asistente rellena el valor de *Implemented Interface*, dándole el nombre de *ICalculator* a la nueva interfaz que vamos también a definir. Y una breve descripción, en el control *Description*, no vendría mal.



Puede dejar el resto de los parámetros del asistente con su valor por omisión. Más adelante veremos el significado de cada uno de ellos. Por el momento, ahí va un resumen:

- 1 *Instancing* no tiene sentido para un servidor DLL, sino para los ejecutables.
- 2 En cambio, *Threading Model* es muy importante para una DLL. Al elegir *Apartment* estamos informando a COM, y de paso a los potenciales clientes de nuestra clase, que cada objeto de esta clase debe ser llamado solamente desde el mismo hilo en que ha sido creado.
- 3 Debemos dejar activa la opción *Include Type Library* para que el proyecto incluya información persistente sobre los tipos de datos de la clase. Esto es lo próximo que veremos.
- 4 Por último, *Mark Interface Oleautomation* informa a COM de que solamente utilizaremos cierto conjunto restringido de tipos de datos en los métodos de nuestra clase. Veremos enseguida que esta disciplina nos ahorrará un montón de trabajo.

En cuanto pulse *OK*, verá que aparece ante usted la siguiente ventana:



Se trata del *Editor de Bibliotecas de Tipos*, o *Type Library Editor*. Sin embargo, para explicar cómo funciona, debo antes contarle qué demonios es una *biblioteca de tipos*.

## Bibliotecas de tipos

Suponga que llega a sus manos una DLL desconocida; le dicen que implementa la comunicación con un sofisticado dispositivo de hardware y que, esté de acuerdo o no, tiene que implementar una aplicación alrededor de esa DLL. ¿Qué posibilidades tiene de triunfar, si sólo cuenta con el fichero binario? Pues muy pocas. Hay herramientas que permiten averiguar el nombre de las funciones exportadas por la DLL, pero no hay forma de obtener los parámetros de cada una de esas funciones, como no sea siguiendo paso a paso la ejecución con mucho cuidado, en lenguaje ensamblador. Hay quien pensará que es muy bueno mantener el secreto de una implementación. Puede que eso sea cierto para algún tipo especial de aplicaciones, pero recono-

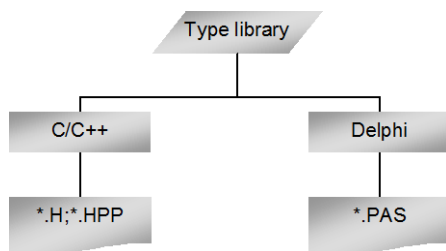
camos que la paranoia alrededor de interfaces que se suponen públicas no ayudan al ideal de reutilización de código.

Por suerte, Microsoft pensó en este problema al diseñar COM, e ideó un mecanismo para que los servidores COM llevasen consigo la documentación necesaria para poder usarlos: las *bibliotecas de tipos*, o *type libraries*. Una biblioteca de tipos es un recurso binario que puede almacenarse indistintamente dentro de un servidor, ya sea en un fichero ejecutable o en una DLL, o en algún fichero especial distribuido junto con el servidor. En el caso de que se encuentre dentro de un fichero independiente, las extensiones más frecuentes del fichero son *tlb* y, para servidores más antiguos, *olb*. Dentro de una biblioteca se almacena información sobre clases, interfaces y tipos especiales que son definidos o implementados por el servidor al que está asociada. No es obligatorio que un servidor se distribuya con una biblioteca de tipos.

En particular, en el ejemplo que estamos desarrollando, la biblioteca se almacena en un fichero llamado *ISFinancials.tlb*. Abra nuevamente el código fuente del proyecto, y verá que ha aparecido una nueva directiva de compilación, más bien de enlace, en su interior, que sirve para incluir la *tlb* dentro de la DLL binaria:

```
{ $R *.TLB }
```

Si el creador de un lenguaje desea que su criatura pueda utilizar clases COM, debe desarrollar una herramienta que reciba como entrada una biblioteca de tipos, y que a partir de ella pueda generar un fichero con las declaraciones de las clases, los tipos de interfaz implementados, los identificadores únicos utilizados, etc. Si el lenguaje es C o C++, el resultado debe ser un fichero de cabecera, de extensión *h* o *hpp*. En el caso de Delphi, debe generarse una unidad, con las declaraciones de tipos y constantes en la sección **interface**:



Para Delphi, en concreto, la herramienta de *importación* de bibliotecas de tipos se ejecuta por medio del comando de menú *Project|Import type library*, que veremos en otro momento. Pero dentro del propio proyecto donde definimos una biblioteca de tipos, no es necesario que tomemos medidas especiales para contar con una unidad Pascal equivalente; Delphi se encarga del asunto. En nuestro ejemplo, la unidad se llama *ISFinancials\_TLB*, y muestra la siguiente advertencia en su cabecera:

```
// *****
// WARNING
// -----
```

```
// The types declared in this file were generated from data read
// from a Type Library. If this type library is explicitly or
// indirectly (via another type library referring to this type
// library) re-imported, or the 'Refresh' command of the
// Type Library Editor activated while editing the Type Library,
// the contents of this file will be regenerated and all
// manual modifications will be lost.
// *****
```

¿Mi consejo? Hágale caso a Borland y no modifique directamente el contenido de esa unidad.

Los servidores COM que definen una biblioteca de tipos gozan de una ventaja adicional: siempre que los tipos de datos utilizados en los parámetros de los métodos pertenezcan a cierto conjunto definido por COM, el sistema operativo podrá organizar el *marshaling* de las llamadas a métodos sin ayuda por nuestra parte. Recuerde que el *marshaling* es la operación mediante la cual el *proxy* y el *stub* simulan la ejecución de los métodos de una interfaz cuando no hay acceso directo desde el cliente hacia el servidor. De no ser por este mecanismo de transmisión automático, tendríamos que echarle una mano al sistema implementado la interfaz *IMarshal* dentro las clases COM que desarrollemos. Por esta razón es que hemos pedido al asistente que ha generado la biblioteca de tipos que la marque como *Oleautomation*, para que COM sepa que nos vamos a adherir al convenio sobre tipos de datos.

## El Lenguaje de Descripción de Interfaces

Puede que una imagen valga por mil palabras, pero un par de palabras bien escritas permiten una precisión imposible en una imagen ordinaria. Quiero decir, que podemos utilizar el editor gráfico de Borland para definir la interfaz *ICalculator*, representada por una piruleta roja, y para asignar atributos e interfaces a la clase COM *Calculator*, que es la tarta azul con una guinda gigante. Pero que podemos hacer lo mismo escribiendo una declaración en modo textual, utilizando un lenguaje especial diseñado para este único propósito. Su nombre es *Interface Description Language*, o lenguaje de descripción de interfaces, aunque es más conocido por sus siglas *IDL*.

IDL es un lenguaje con fuerte sabor a C, aunque no es lo es. Con IDL podemos describir una biblioteca de tipos definiendo interfaces, algunos tipos auxiliares que podamos necesitar, como enumerativos y registros, y clases COM. Para que se haga una idea sobre IDL, incluyo a continuación la declaración de la interfaz *ICalculator* que vamos a utilizar en nuestro ejemplo:

```
[ uuid(946EECA1-D3B0-11D5-8943-00C026263861),
  version(1.0),
  helpstring("Interface for Calculator Object"),
  oleautomation ]
interface ICalculator: IUnknown
{
    [propget, id(0x00000001)]
    HRESULT _stdcall Financiado([out, retval] CURRENCY * Valor );
    [propput, id(0x00000001)]
```

```

HRESULT _stdcall Financiado([in] CURRENCY Valor );
[ propget, id(0x00000002) ]
HRESULT _stdcall Interes(
    [out, retval] double * Valor );
[propput, id(0x00000002)]
HRESULT _stdcall Interes([in] double Valor );
[propget, id(0x00000003)]
HRESULT _stdcall Plazo(
    [out, retval] long * Valor );
[propput, id(0x00000003)]
HRESULT _stdcall Plazo([in] long Valor );
[propget, id(0x00000004)]
HRESULT _stdcall PrimeraCuota(
    [out, retval] CURRENCY * Valor );
[propget, id(0x00000005)]
HRESULT _stdcall CuotaRegular(
    [out, retval] CURRENCY * Valor );
[propget, id(0x00000006)]
HRESULT _stdcall TotalIntereses(
    [out, retval] CURRENCY * Valor );
[id(0x00000007)]
HRESULT _stdcall GetSaldo([in] long Mes,
    [out, retval] CURRENCY * Valor );
[id(0x00000009)]
HRESULT _stdcall GetPrincipal([in] long Mes,
    [out, retval] CURRENCY * Valor );
[id(0x0000000A)]
HRESULT _stdcall GetIntereses([in] long Mes,
    [out, retval] CURRENCY * Valor );
[id(0x0000000B)]
HRESULT _stdcall GetCuota([in] long Mes,
    [out, retval] CURRENCY * Valor );
};

```

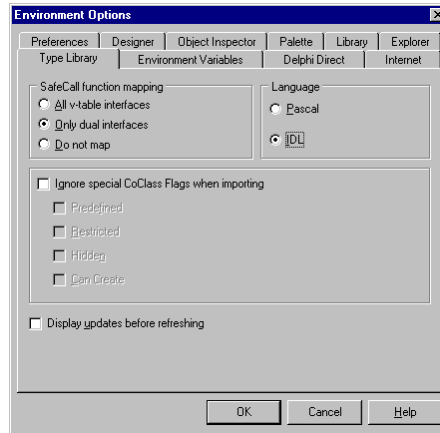
Si carga el ejemplo terminado que se encuentra en el CD-ROM, podrá aislar esta declaración seleccionando en el árbol de la izquierda el elemento correspondiente a la interfaz, y activando luego la página *Text* a la derecha. Seleccione después el nodo de la clase COM, y busque en la página *Text* correspondiente la siguiente declaración:

```

[
    uuid(946EECA3-D3B0-11D5-8943-00C026263861),
    version(1.0),
    helpstring("Calculator")
]
coclass Calculator
{
    [default] interface ICalculator;
};

```

Si usted detesta tanto como yo todo lo que huele a C, le gustará saber que no tendrá que aguantar la respiración para trabajar con bibliotecas de tipos en modo texto. Como, en definitiva, lo importante es la descripción “binaria” de la misma, Delphi ofrece la posibilidad de mostrar el contenido de una biblioteca en un lenguaje equivalente más al aire de Pascal. La opción se encuentra dentro del diálogo *Environment options*, en la página *Type Library*:



El grupo de opciones *Language* es el que nos interesa ahora. Voy a dejar activa, sin embargo, la opción *IDL*, porque se trata de un estándar *de facto*. De paso, eche un vistazo al grupo *SafeCall function mapping*. Por el momento, vamos a dejar marcada la opción *Only dual interfaces*, pero más adelante, para este mismo ejemplo que estamos desarrollando, cambiaremos a *All v-table interfaces*.

Vamos a centrarnos en el siguiente par de métodos, extraídos de la declaración de la interfaz *Calculator*:

```
[propget, id(0x00000001)]
HRESULT _stdcall Financiado([out, retval] CURRENCY * Valor );
[propput, id(0x00000001)]
HRESULT _stdcall Financiado([in] CURRENCY Valor );
```

Observe, en primer lugar, que tenemos dos métodos con el mismo nombre, aunque con diferentes prototipos. Sin embargo, está esa cláusula extraña antes del prototipo; sirve para asociar atributos semánticos a los métodos que no son soportados por C ó C++ (jun par de lenguajes muy primitivos!). Comprobará que uno de los métodos lleva el atributo **propget**. Eso significa que debe utilizarse para implementar el acceso en modo lectura de una propiedad llamada *Financiado*. El segundo método utiliza **propput**, y su atributo **id** tiene el mismo valor del método anterior. Así indicamos que es la pareja del método de acceso anterior.

Note también que los parámetros de los métodos llevan indicaciones detalladas sobre su uso. Vemos, por ejemplo que existen atributos **in** y **out** que, aunque no se muestra en el ejemplo, se pueden combinar. Hay también un atributo **retval**, para “simular” el valor de retorno de una función...

Hay algo, sin embargo, que sigue teniendo color de hormiga, que no encaja. Si busca la declaración de la interfaz *ICalculator* en Delphi, esta vez dentro de la unidad *ISFinancials\_TLB*, encontrará que los métodos han sido importados con los siguientes prototipos:

```
function Get_Financiado(out Valor: Currency): HRESULT; stdcall;
function Set_Financiado(Valor: Currency): HRESULT; stdcall;
```

¿Son apropiadas estas declaraciones para crear una propiedad a partir de ellas? ¿Qué pinta el omnipresente tipo *HRESULT* como valor de retorno de *todas* las funciones de la interfaz?

## La directiva safecall

Comencemos por *HRESULT*: ya sabemos que es un tipo entero que utilizan casi todas las funciones del API de COM para informar del éxito o el fracaso a quien ejecuta una de ellas. Eso mismo es lo que pretendemos que hagan los métodos de *ICalculator*. Todos ellos serán programados de modo que devuelvan *S\_OK* cuando terminen correctamente, y algún valor negativo (con el bit más significativo activo) cuando se produzca una excepción. Es muy importante que comprenda este hecho:

*Nuestra clase puede potencialmente ser utilizada desde cualquier lenguaje de protección, incluso desde lenguajes que no utilizan excepciones.*

La premisa anterior tiene su consecuencia:

*A diferencia de lo que es conveniente en una aplicación GUI, no debemos dejar que una excepción “flote”, sin ser atendida, más allá de alguno de nuestros métodos.*

Por lo tanto, y en principio, deberíamos implementar todos los métodos de nuestras clases COM con un patrón similar al siguiente:

```
function TLoQueSea.MetodoAbsurdo(
  ParametrosTontos: TipoRaro): HRESULT; _stdcall;
begin
  try
    IntentarHacerAlgo;
    Result := S_OK;
  except
    Result := S_FAIL;           // Al menos, esta precaución
  end;
end;
```

Las aplicaciones que usen la clase deben ser conscientes de la posibilidad de fallo. En nuestro caso, en que implementaremos esas aplicaciones con el mismo Delphi, deberemos “traducir” de vuelta el código de error en una excepción, si no queremos escribir un programa endeble. Por ejemplo, en el programa de prueba del CDROM, la asignación de un valor a la “propiedad” financiado se escribe de esta absurda manera:

```
OleCheck(Calc.Set_Financiado(StrToCurr(edImporte.Text)));
```

¿Hay algo que podamos hacer? Sí, pedirle ayuda a Delphi. Cierre el proyecto del servidor con el comando de menú *File|Close all*. Ejecute el comando *Tools|Environment options*, y seleccione la opción *All v-table interfaces*, de nuevo en la página *Type Li-*

*brary*. Delphi le advertirá que el cambio se aplicará solamente a partir del siguiente proyecto que abra.

### ADVERTENCIA

Para que usted pueda comparar las dos formas de trabajo, hay dos servidores diferentes en el CDROM, con dos clientes independientes, pero funcionalmente equivalentes. Sin embargo, para simplificar haré como que modificamos el código fuente del mismo proyecto.

Si ahora carga el proyecto del servidor, selecciona la biblioteca de tipos mediante el comando *View|Type Library* y pulsa el botón de guardar la misma, verá que el fichero *ISFinancials\_TLB* genera una declaración en Delphi para la interfaz *ICalculator* muy diferente a la anterior:

```
type
  ICalculator2 = interface(IUnknown)
    ['{164DF662-D4A8-11D5-8943-00C026263861}']
    function Get_Financiado: Currency; safecall;
    procedure Set_Financiado(Valor: Currency); safecall;
    function Get_Interes: Double; safecall;
    procedure Set_Interes(Valor: Double); safecall;
    function Get_Plazo: Integer; safecall;
    procedure Set_Plazo(Valor: Integer); safecall;
    function Get_PrimerCuota: Currency; safecall;
    function Get_CuotaRegular: Currency; safecall;
    function Get_TotalIntereses: Currency; safecall;
    function GetSaldo(Mes: Integer): Currency; safecall;
    function GetPrincipal(Mes: Integer): Currency; safecall;
    function GetIntereses(Mes: Integer): Currency; safecall;
    function GetCuota(Mes: Integer): Currency; safecall;
    property Financiado: Currency read Get_Financiado
      write Set_Financiado;
    property Interes: Double read Get_Interes write Set_Interes;
    property Plazo: Integer read Get_Plazo write Set_Plazo;
    property PrimeraCuota: Currency read Get_PrimerCuota;
    property CuotaRegular: Currency read Get_CuotaRegular;
    property TotalIntereses: Currency read Get_TotalIntereses;
  end;
```

Los métodos han sufrido una transformación espectacular: el valor de retorno *HResult* ha desaparecido de todos los prototipos, y el convenio de llamado ha dejado de ser **stdcall** para convertirse en **safecall**. Compare, por ejemplo, las traducciones de *Set\_Financiado*, antes y después de activar la opción:

```
// ANTES
function Set_Financiado(Valor: Currency): HResult; stdcall;
// DESPUES
procedure Set_Financiado(Valor: Currency); safecall;
```

Aún más espectacular es el cambio sufrido por *Get\_Financiado*:

```
// ANTES
function Get_Financiado(out Valor: Currency): HResult; stdcall;
// DESPUES
function Get_Financiado: Currency; safecall;
```

Con los nuevos prototipos, ya es posible definir una propiedad correctamente:

```
property Financiado: Currency
    read Get_Financiado write Set_Financiado;
```

Y, como comprobará en el ejemplo de uso de la clase, es mucho más sencillo asignar el valor financiado a la clase:

```
Calc.Financiado := StrToCurr(edImporte.Text);
```

¿Qué es lo que ha hecho Delphi cuando hemos activado la opción mencionada? En primer lugar, debe darse cuenta de que la descripción de la Biblioteca de Tipos no cambia en lo más mínimo. En segundo lugar, Delphi al editor de la Biblioteca de Tipos que genere todos los métodos en Delphi marcados con la directiva **safecall**. Cuando Delphi compila un método **safecall**, añade automáticamente código para capturar todas las excepciones que floten fuera del método, y las transforma en códigos de error de COM.

Por otra parte, cuando llamamos en Delphi a un método declarado con **safecall**, el compilador también añade las instrucciones necesarias para convertir un posible código de error en una excepción.

## Implementación de interfaces para COM

Confieso que la explicación anterior sobre las bibliotecas de tipos ha consumido más espacio del que esperaba. Y hemos perdido de vista el segundo fichero que se genera al ejecutar el asistente para la creación de *COM Objects*: la unidad con la declaración de la clase. A diferencia de los ficheros relacionados con la biblioteca de tipos, que reciben un nombre fijo, tenemos total libertad para nombrar esta unidad. En el ejemplo del CD-ROM, la he bautizado como *Calculator*. Esta es la declaración de la clase correspondiente al proyecto en que he forzado el uso de **safecall**:

```
type
    TCalculator = class(TTypedComObject, ICalculator)
    private           // Esta sección es nuestra
        FCalculator: TisCalculator;
    public           // Esta sección también es nuestra
        procedure Initialize; override;
        destructor Destroy; override;
    protected      // Creada por Delphi
        function Get_Financiado: Currency; safecall;
        function Get_Interes: Double; safecall;
        function Get_Plazo: Integer; safecall;
        function Get_PrimerCuota: Currency; safecall;
        function Get_CuotaRegular: Currency; safecall;
        function Get_TotalIntereses: Currency; safecall;
        function GetCuota(Mes: Integer): Currency; safecall;
        function GetIntereses(Mes: Integer): Currency; safecall;
        function GetPrincipal(Mes: Integer): Currency; safecall;
        function GetSaldo(Mes: Integer): Currency; safecall;
        procedure Set_Financiado(Valor: Currency); safecall;
        procedure Set_Interes(Valor: Double); safecall;
```



```

procedure Set_Plazo(Valor: Integer); safecall;
end;

```

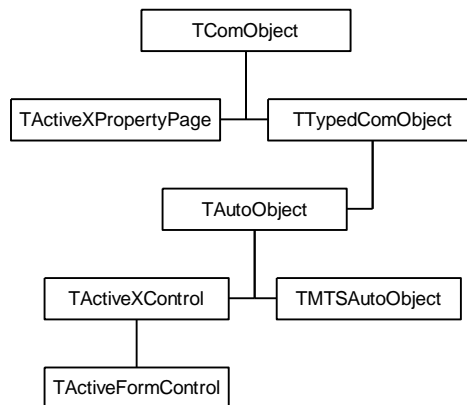
A estas alturas, puede que esté confundido por la profusión de nombres y entidades creados por el asistente. Ahí va un resumen:

Identificador	Creador	Uso
<i>Calculator</i>	Nosotros	El nombre que le hemos dado a la clase COM.
<i>ICalculator</i>	Delphi	La interfaz por omisión, propuesta por Delphi.
<i>TCalculator</i>	Delphi	La clase Delphi que implementa la clase COM.
<i>Calculator</i>	Nosotros	El nombre que damos a la unidad de <i>TCalculator</i> .
<i>CoCalculator</i>	Delphi	Una clase auxiliar que utilizarán los clientes.

De todos ellos, el más extraño es *CoCalculator*, porque todavía no lo he presentado en sociedad. Lo veremos dentro de poco, cuando explique cómo podemos utilizar la clase COM desde otra aplicación escrita en Delphi.

Sigamos entonces con la clase *TCalculator*. Lo primero que observamos es que implementa la interfaz *ICalculator*, como era de esperar, y que descende de una clase predefinida por Delphi, llamada *TTypedComObject*. Esta clase, declarada en *ComObj*, ya implementa las interfaces *IUnknown* e *ISupportErrorInfo*; la segunda interfaz se utiliza para la propagación de excepciones, cuando los métodos de la clase utilizan la directiva **safecall**.

Los diversos asistentes que crean clases COM eligen automáticamente la clase base para la implementación, pero es bueno conocer un poco más sobre ellas. El siguiente diagrama muestra las relaciones entre las clases de implementación básicas de Delphi:



Están, además, las clases que se utilizan para los módulos de datos remotos, en los servidores de capa intermedia de DataSnap. Debe también saber que *TComponent* está igualmente preparada para la implementación de interfaces COM.

Si va al final de la unidad, encontrará la siguiente instrucción:

```

initialization
  TTypedComObjectFactory.Create(ComServer, TCalculator2,
    Class_Calculator2, ciMultiInstance, tmApartment);
end.

```

La clase *TTypedComObjectFactory* es la fábrica de clases que corresponde en Delphi a las clases COM implementadas a partir de *TTypedComObject*. La instrucción que he mostrado cumple dos objetivos: en primer lugar, crea el objeto que COM utilizará más adelante para crear las instancias de *TCalculator* a petición de un cliente. Pero también inserta el objeto creado en una lista perteneciente a una estructura global que puede obtenerse mediante la siguiente función de *ComObj*:

```

function ComClassManager: TComClassManager;

```

Precisamente, la implementación predefinida de *DllGetClassObject* realiza un recorrido dentro de esa estructura para localizar una fábrica de clases en beneficio de *CoGetClassFactory*.

### IMPORTANTE

¿Recuerda que, al ejecutar el asistente para la creación de un *COM Object*, tuvo que especificar un modelo de hilos, *threading model*, y de creación de instancias, o *instanting*? Si quiere modificar los valores que eligió inicialmente, debe retocar los parámetros del constructor de la fábrica de clase, en la sección de inicialización de la unidad.

Queda muy poco por ver en este ejemplo. Para no aturdirlo con los detalles de implementación de los cálculos de intereses, cuotas y tablas de amortización, he creado una unidad, *isFCalc*, en donde declaro e implemento una clase “normal” de Delphi, *TisCalculator*, para que se ocupe de esos detalles. Internamente, *TCalculator* crea una instancia de *TisCalculator*, que deberá destruir durante su propia destrucción:

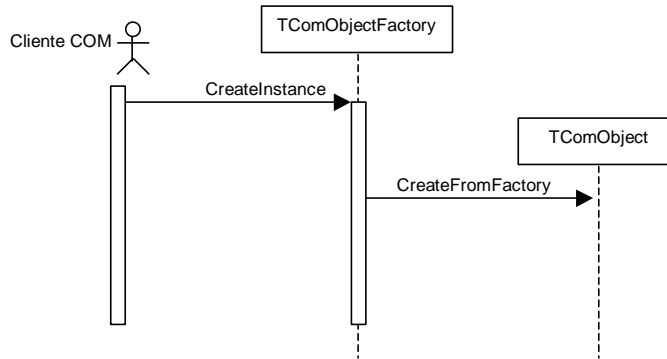
```

procedure TCalculator2.Initialize;
begin
  inherited Initialize;
  FCalculator := TisCalculator.Create;
end;

destructor TCalculator2.Destroy;
begin
  FCalculator.Free;
  inherited Destroy;
end;

```

¿Ha notado que he realizado la inicialización en un método llamado *Initialize*, en vez de añadirla al constructor? El problema es que los constructores de los objetos descendientes de *TComObject* no son virtuales. El siguiente diagrama de secuencias muestra lo que sucede cuando se crea un objeto COM:



He utilizado los nombres de clases bases, como *TComObject*, en vez de *TTypeComObject*, pero este detalle no cambia la explicación. Para crear un objeto, la fábrica de clases utiliza el constructor *CreateFromFactory*, que no es virtual. Por lo tanto, si añadimos redefinimos este constructor en una clase derivada y añadimos instrucciones, éstas nunca serán ejecutadas por la versión estática de *CreateFromFactory* que es ejecutada por la fábrica de clases. Sin embargo, la implementación de este constructor *sí* realiza una llamada al método virtual *Initialize* para resolver el problema. Por lo tanto, cuando estemos implementando clases COM en Delphi, y la clase base utilizada descienda de *TComObject*, directa o indirectamente, debemos alojar nuestras instrucciones de inicialización en una nueva versión del método *Initialize*.

La clase *TCalculator*, finalmente, se limita a delegar la implementación de sus métodos sobre los de *TisCalculator*; el código fuente completo, por supuesto, se encuentra en el CD-ROM que acompaña a este libro.

## Registrando el servidor

Antes de poner a prueba el servidor, debemos *registrarlo* en nuestro ordenador. La manera más sencilla de hacerlo es utilizando el propio Entorno de Desarrollo. El comando de menú *Run | Register ActiveX Server* se encarga de esta tarea. Note que también existe *Unregister ActiveX Server*, para deshacer el registro si fuese necesario.

Pero, ¿qué hacen estos comandos? Ambos cargan la DLL dentro del proceso activo, y buscan en su interior una de las funciones *DllRegisterServer* o *DllUnregisterServer*, para ejecutarla. En realidad, la propia unidad *ComObj* ofrece una función llamada *RegisterComServer*, por si necesitamos registrar manualmente un servidor *in-process* desde una aplicación. Esta es su implementación:

```

procedure RegisterComServer(const DLLName: string);
type
    TRegProc = function: HRESULT; stdcall;
const
    RegProcName = 'DllRegisterServer';
var
    Handle: THandle;
  
```

```

    RegProc: TRegProc;
begin
    Handle := SafeLoadLibrary(DLLName);
    if Handle <= HINSTANCE_ERROR then
        raise Exception.CreateFmt('%s: %s',
            [SysErrorMessage(GetLastError), DLLName]);
    try
        RegProc := GetProcAddress(Handle, RegProcName);
        if Assigned(RegProc) then
            OleCheck(RegProc)
        else
            RaiseLastOSError;
    finally
        FreeLibrary(Handle);
    end;
end;

```

Para deshacer el registro, nos bastaría crear una segunda función idéntica, pero sustituyendo el valor de la constante *RegProcName* por *DllUnregisterServer*.

Si necesitásemos registrar una DLL desde la línea de comandos del sistema operativo, podríamos también recurrir a una sencilla aplicación distribuida por Borland: *tregsvr.exe*, que se encuentra en el directorio *bin* de Delphi. Se puede ejecutar en una de estas dos formas:

```

rem Para registrar:
tregsvr nombreDll.dll

rem Para deshacer el registro
tregsvr -u nombreDll.dll

```

De todos modos, si vamos a distribuir un servidor COM dentro del proceso como parte de una aplicación, y utilizamos InstallShield para generar la instalación, es muy sencillo indicarle a esta herramienta que registre automáticamente la DLL una vez que haya terminado de copiar los ficheros en el disco duro.

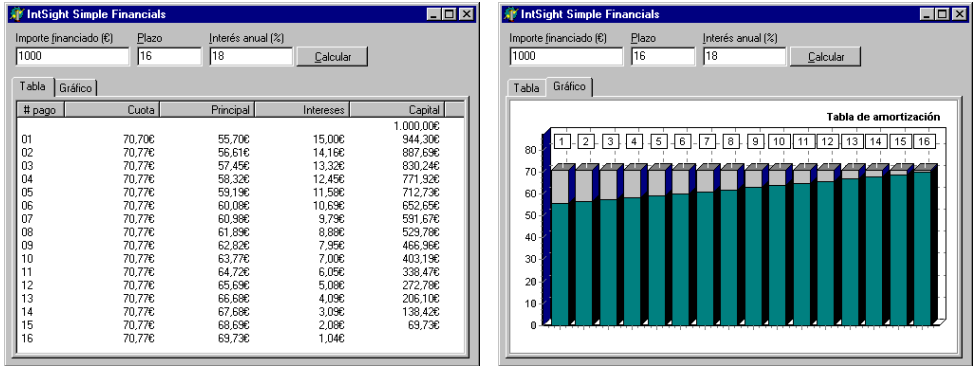
## Un cliente para la calculadora

Y ya estamos en condiciones de crear un cliente para la clase de la calculadora. No voy a mostrar todos los detalles sucios, sino solamente la forma en que el cliente crea una instancia de la clase COM. Una vez que hayamos creado un nuevo proyecto de aplicación, el siguiente paso será... ¡incluir el fichero *ISFinancials\_TLB.pas*, que fue generado automáticamente al programar el servidor! En definitiva, es ahí donde se encuentra toda la información necesaria para que el cliente pueda trabajar con la clase:

- Los GUIDs, o identificadores únicos, de la clase y la interfaz.
- La declaración del tipo de interfaz *ICalculator*: sus métodos y propiedades.

Pero, ¿qué pasaría si recibiéramos solamente la DLL con el servidor? Muy fácil: tendríamos que ejecutar el comando *Project | Import Type Library* para crear la unidad a

partir del recurso binario correspondiente a la biblioteca de tipo incluida dentro de la DLL. Voy a posponer la explicación de esta operación para cuando estudiemos la automatización OLE.



Cuando el nuevo proyecto tenga acceso a la unidad con las declaraciones del servidor, podremos obtener punteros a la interfaz *ICalculator* mediante la clase auxiliar *CoCalculator* definida dentro de *ISFinancials\_TLB.pas*:

```
type
  CoCalculator = class
    class function Create: ICalculator;
    class function CreateRemote(
      const MachineName: string): ICalculator;
  end;
```

Observe que *CoCalculator* solamente contiene dos métodos de clase, lo que significa que nunca necesitaremos crear instancias reales de *CoCalculator* para poder utilizar dichos métodos. Esta clase es, en realidad, una pequeña ayuda que nos ofrece Delphi para la creación de instancias de la clase COM. La implementación de los métodos *Create* y *CreateRemote* encapsula llamadas a un par de viejas conocidas: *CreateComObject* y *CreateRemoteComObject*.

```
class function CoCalculator.Create: ICalculator;
begin
  Result := CreateComObject(CLASS_Calculator) as ICalculator;
end;

class function CoCalculator.CreateRemote(
  const MachineName: string): ICalculator;
begin
  Result := CreateRemoteComObject(MachineName, CLASS_Calculator)
    as ICalculator;
end;
```

La aplicación cliente puede entonces obtener, y utilizar posteriormente, un puntero de tipo *ICalculator* del siguiente modo:

```
procedure TwndMain.bnCalcularClick(Sender: TObject);
const
```

```

    EUROFMT = '0,.00€;;#';
var
    I: Integer;
    Calc: ICalculator;
begin
    Calc := CoCalculator.Create;
    Calc.Financiado := StrToCurr(edImporte.Text);
    Calc.Plazo := StrToInt(edPlazo.Text);
    Calc.Interes := StrToFloat(edInteres.Text);
    Series1.Clear;
    Series2.Clear;
    Tabla.Items.BeginUpdate;
    try
        Tabla.Items.Clear;
        for I := 0 to Calc.Plazo do
            with Tabla.Items.Add do
                begin
                    Caption := FormatFloat('00;;#', I);
                    SubItems.Add(FormatCurr(EUROFMT, Calc.GetCuota(I)));
                    SubItems.Add(FormatCurr(EUROFMT, Calc.GetPrincipal(I)));
                    SubItems.Add(FormatCurr(EUROFMT, Calc.GetIntereses(I)));
                    SubItems.Add(FormatCurr(EUROFMT, Calc.GetSaldo(I)));
                end;
            end;
        finally
            Tabla.Items.EndUpdate;
        end;
    end;
end;

```

Como puede apreciar, no hay que tomar medidas especiales para el control de errores. Si se produjese alguna excepción dentro de la clase “remota”, las instrucciones añadidas en el servidor por Delphi detendría su propagación y produciría un valor de retorno con la información requerida. Dentro del cliente, Delphi volvería a generar de forma automática la excepción original. Note también que no es necesario destruir el objeto COM. La única referencia al mismo es la variable local *Calc*; como sabemos, el objeto se destruirá automáticamente al terminar la ejecución del procedimiento anterior.

## Servidores fuera del proceso

**P**ASEMOS PÁGINA PARA OCUPARNOS DE LAS clases implementadas dentro de servidores ejecutables. Desde el punto de vista de sus clientes, estas clases son idénticas a las implementadas dentro de una DLL. A fin de cuentas, la transparencia de la ubicación es uno de los objetivos declarados del Modelo de Objetos Componentes.

### Diferencias técnicas

Pero, como era de esperar, los detalles de implementación de este tipo de servidor varían mucho respecto a los servidores dentro del proceso:

- La forma de registrar y de eliminar un servidor del registro son diferentes.
- Cambia también la técnica para publicar las fábricas de clases.
- Es el propio servidor quien decide detener su ejecución, cuando no hay ningún cliente en su espacio de proceso. Recuerde que los servidores DLL deben implementar la función *DllCanUnloadNow* para indicar al sistema operativo que pueden ser descargados.
- Para ayudar a la implementación del *stub*, hay que implementar un bucle de mensajes... y tener mucho cuidado en cómo lo hacemos.
- Finalmente, las consideraciones sobre concurrencia son diferentes que en los servidores dentro del proceso.

¿Qué motivos existen para querer que una clase COM se implemente dentro de un ejecutable? El primero de ellos es de tipo histórico: en los inicios de DCOM, era la única forma directa de utilizar una clase remota. Sólo con la aparición de MTS, que más tarde se transformó en COM+, surgió la posibilidad de ejecutar una DLL remota, siempre que se registrase dentro del entorno MTS.

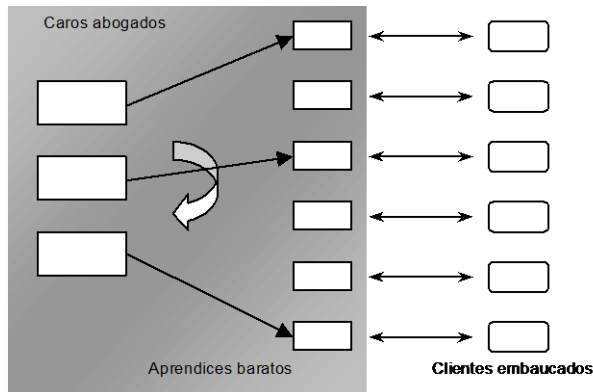
Otra razón es la posibilidad de compartir datos entre procesos. Cuando una clase de un servidor *in-process* es instanciada por un proceso, su espacio de memoria pertenece al proceso cliente, y tiene las mismas dificultades que la aplicación que la aloja para comunicarse con otros procesos. Si quisiéramos mantener estructuras de datos comunes a todas las instancias de esa clase tendríamos que recurrir a técnicas más o

menos complejas, como el uso de memoria global mediante ficheros asociados a memoria (*memory mapped files*).

En cambio, es posible implementar servidores fuera del proceso que creen y ejecuten todos sus objetos dentro de una misma instancia o proceso. Los objetos, como consecuencia, pueden acceder a la memoria global del proceso, siempre que tomen las debidas precauciones relacionadas con el acceso concurrente.

Los servidores fuera del proceso también permiten implementar con mayor facilidad objetos accesibles globalmente. Abra bien los ojos: he escrito *objetos*, no *clases*. COM implementa una estructura a nivel de estación de trabajo llamada en inglés *Running Object Table (ROT)*, traducible más o menos como la tabla de objetos en ejecución. Cuando creamos un objeto tenemos la opción de registrarlo dentro de esa estructura de datos. Un cliente puede solicitar entonces un puntero a *esa* instancia, en vez de crear una nueva. Es una técnica muy efectiva, sobre todo para clases que trabajan con la interfaz de usuarios.

Por último, la flexibilidad del modelo de creación de instancias dentro de un servidor ejecutable es la base de una técnica conocida como *object pooling*; la palabra *pooling*, en esta acepción, debe entenderse en el sentido de un *depósito* o *fondo común* de objetos. Imagine que se reúnen tres abogados y montan una conspiración: contratan a cincuenta aprendices y les enseñan a comportarse como abogados; la lección número uno consiste en robarle el caramelo a un niño. Cuando el adoctrinamiento ha concluido, los tres espíritus malignos abren la jaula y lanzan sus criaturas al mundo, a buscar víctimas potenciales, quiero decir, clientes.



Un cliente contrata a un aprendiz pensando que está tratando directamente con un abogado, porque la interfaz de ambos objetos es idéntica. Sin embargo, el aprendiz implementa sus métodos delegándolos en uno de los tres compinches originales. Esto implica, naturalmente, que en ocasiones los farsantes aprendices tengan que ponerse en cola. En el improbable caso de que los abogados tuvieran un mínimo de conciencia, la técnica explicada permitiría abaratar el precio de sus servicios.



## Modelos de creación de instancias

Una de las diferencias entre los servidores dentro y fuera del proceso es la forma en que se activa el módulo anfitrión. En el caso de un servidor *in-process*, la DLL se carga sin más en el espacio del proceso del cliente. Pero en el otro caso, hay que cargar y ejecutar una aplicación, y aquí tenemos dos posibilidades:

- 1 Al ejecutar el asistente de creación de la clase, especificamos que *Instanting* sea *Single Instance*. Eso quiere decir que si queremos tener diez objetos de esta clase, el sistema debe ejecutar diez instancias de la aplicación que la contiene. Tenga mucho cuidado con la palabra *instancia*, porque están en juego las instancias de la aplicación y las instancias de la clase, y se trata de entidades diferentes. En este contexto, *single instance* significa en realidad que sólo se permite una instancia *de la clase* dentro de un proceso dado.
- 2 Por el contrario, la opción *Multiple Instance* es la indicada cuando queremos que un mismo proceso pueda alojar múltiples instancias de la clase. Es decir, que permitiremos múltiples instancias *de la clase* dentro de un mismo proceso.

Cada alternativa tiene sus pros y sus contras. Cuando existen múltiples instancias dentro de un proceso, se consumen menos recursos que si tuviéramos que lanzar varios procesos con el mismo objetivo. Además, los objetos podrían comunicarse entre sí con mayor facilidad, si fuese necesario, porque pueden acceder directamente a la memoria global del proceso.

No obstante, existe también mayor riesgo de fallos, porque un objeto utilizado incorrectamente puede destrozar la memoria común del proceso. Esto puede ocurrir independientemente de que programemos un hilo separado para cada objeto. Es usual, por ejemplo, que parte de la construcción de las instancias por parte de la fábrica de clases ocurra dentro del hilo principal del proceso. Mi experiencia con aplicaciones de este tipo me dice que pueden ocurrir “cuelgues” si hay algún problema durante la instanciación.

La valoración para servidores de una sola instancia es la inversa: consumen más recursos, pero cada objeto disfruta de un espacio de memoria independiente, a salvo de ataques maliciosos por parte de sus hermanos. Además, es más sencillo abortar la ejecución de una instancia que haya perdido el control. En particular, se recomienda el uso de servidores de una instancia cuando la clase debe manejar elementos de la interfaz de usuario.

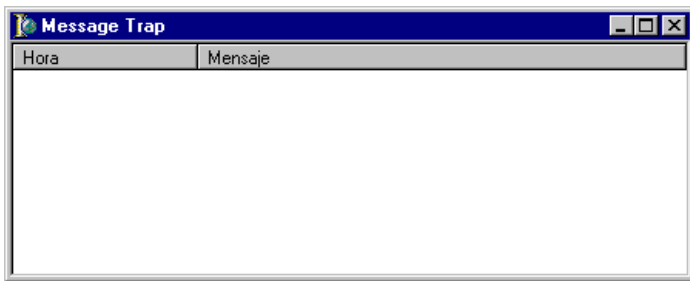
### RECOMENDACIÓN

De todos modos, el modelo generalmente preferido de creación de instancias es el de múltiples instancias dentro de un mismo servidor; siempre que conozcamos en profundidad las consecuencias de esta decisión para la ejecución concurrente. Veremos ahora que no es tarea trivial.

## La Tabla de Objetos Activos

¿Qué tal si mostramos un ejemplo sencillo de servidor ejecutable? El que voy a explicar no es precisamente el tipo de servidor que veremos con mayor frecuencia en la práctica, pero nos servirá para ilustrar algunos conceptos importantes. En concreto, vamos a programar una “trampa de mensajes”, que funcionará como un objeto global, a nivel de estación de trabajo, y en el que los clientes depositarán mensajes con el formato de cadenas de caracteres.

Comenzamos entonces con una aplicación nueva; guardaremos la unidad principal bajo el nombre de *Monitor*, y el proyecto completo como *MsgTrap*. Cambiaremos el nombre de la ventana principal a *wndMonitor*. En su interior añadiremos un control de tipo *TListView*, asignaremos *vsReport* en su propiedad *ViemStyle* y crearemos dos columnas para que tenga este aspecto:

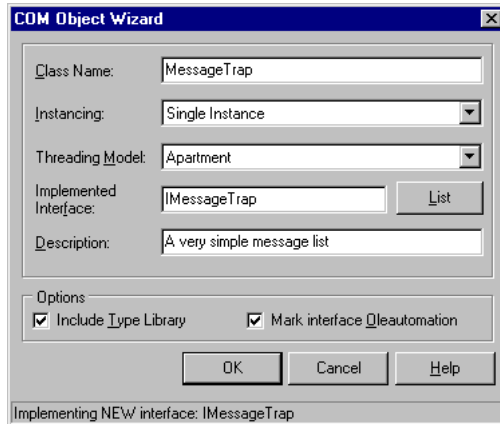


En la sección **public** del formulario crearemos el siguiente método:

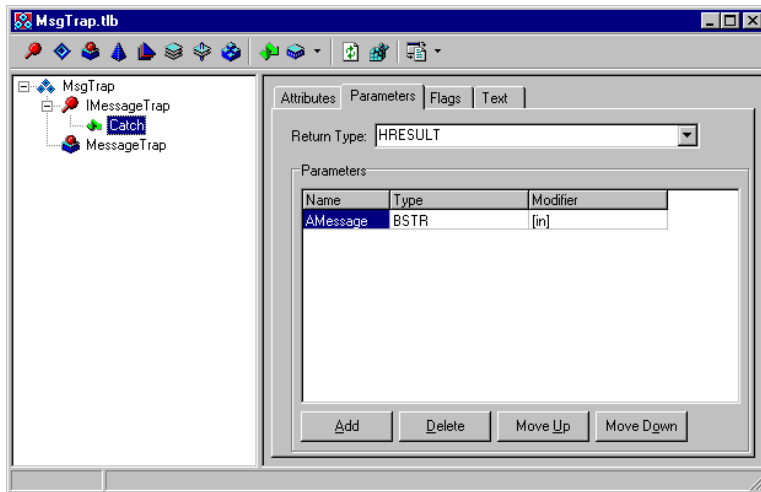
```
procedure TwndMonitor.Add(const AMsg: string);
begin
    with ListView1.Items.Add do
        begin
            Caption := FormatDateTime('dd/mm/yy hh:nn:ss', Now);
            SubItems.Add (AMsg);
        end;
    end;
```

Es el momento de ejecutar el comando *File|New*, ir a la segunda página del Almacén de Objetos y hacer doble clic sobre el icono *COM Object*. Voy a asumir que hemos dejado activa la opción de utilizar **safecall** para todas las interfaces, en las opciones del Entorno de Desarrollo.

En el asistente que aparece en respuesta al doble clic, indicamos que el nombre de la clase será *MessageTrap*, y que su modelo de instanciación será *Single Instance*. Dejamos el resto de las opciones tal como están, pulsamos *OK* y guardamos el nuevo fichero generado con el nombre de *MessageTrap*.



Tenemos que definir los métodos de la interfaz *IMessageTrap* utilizando el editor de la biblioteca de tipos. Crearemos un solo método, que llamaremos *Catch*. Seleccionamos el nodo apropiado en el árbol de la izquierda y definimos los parámetros de *Catch* en la página correspondiente. Note que, a pesar de haber marcado los métodos de *IMessageTrap* con la directiva **safecall**, el editor de la biblioteca de tipos sigue mostrando el tipo de retorno *HResult*. Esto no tiene mayor trascendencia; sólo tiene que recordar que **safecall** es un recurso ideado por Delphi.



Necesitamos que *Catch* tenga un único parámetro de entrada. Puede darle un nombre arbitrario, pero debe utilizar el tipo *BSTR* en su declaración. En realidad, *BSTR* es una macro de IDL que corresponde al tipo *WideString* en Delphi, que soporta el formato Unicode. Siempre que utilicemos cadenas de caracteres en un método de una interfaz COM, debemos utilizar *WideString* como tipo de datos, si queremos descargar el *marshaling* sobre los hombros del sistema operativo. La declaración final de la interfaz en IDL es la siguiente:

```
[ uuid(5ABCEE66-D5F6-11D5-8943-00C026263861), version(1.0),
  helpstring("MessageTrap Interface"), oleautomation ]
interface IMessageTrap: IUnknown
{
  [id(0x00000001)]
  HRESULT _stdcall Catch([in] BSTR AMessage );
};
```

Y ésta será la declaración equivalente en Delphi:

```
type
  IMessageTrap = interface(IUnknown)
    ['{5ABCEE66-D5F6-11D5-8943-00C026263861}']
    procedure Catch(const AMessage: WideString); safecall;
  end;
```

Nos vamos entonces a la unidad *MessageTrap*, para darle una implementación al método *Catch*:

```
procedure TMessageTrap.Catch(const AMessage: WideString);
begin
  wndMonitor.Add(AMessage);
end;
```

Por supuesto, para acceder a la variable global *wndMonitor* tenemos que añadir el nombre de la unidad *Monitor* a la cláusula **uses** de *MessageTrap*.

¿No le entran escalofríos al ver cómo manipulo objetos globales sin las debidas precauciones? Quiero decir, ¿no se supone que tendría que utilizar algún objeto de sincronización, como una sección crítica o un *mutex*, antes de emprender estas aventuras? No se preocupe, más adelante explicaré por qué no es necesario *en este caso* especial.

Hasta aquí, los acontecimientos eran predecibles, pero es el momento de las novedades. Declare los siguientes métodos la sección **public** de la clase *TMessageTrap*:

```
type
  TMessageTrap = class(TTypedComObject, IMessageTrap)
  protected
    FCookie: Integer;
    procedure Catch(const AMessage: WideString); safecall;
  public
    procedure Initialize; override;
    destructor Destroy; override;
  end;
```

E impleméntelos de la siguiente manera:

```
procedure TMessageTrap.Initialize;
begin
  inherited Initialize;
  OleCheck(RegisterActiveObject(Self, Class_MessageTrap,
    ACTIVEOBJECT_WEAK, FCookie));
end;
```

```

destructor TMessageTrap.Destroy;
begin
    OleCheck(RevokeActiveObject(FCookie, nil));
    inherited Destroy;
end;

```

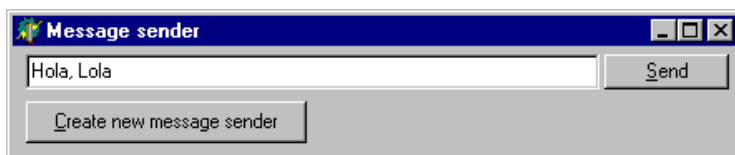
El misterio tiene relación con la *Tabla de Objetos Activos*, o ROT, cuya existencia mencioné hace poco. Durante la creación de cada objeto de la clase *TMessageTrap* ejecutamos la función *RegisterActiveObject* del API de COM. Esta función inserta una referencia a la interfaz *IUnknown* del objeto recién creado en la ROT global, advirtiendo, eso sí, a qué clase pertenece: en el segundo parámetro hemos pasado la constante *Class\_MessageTrap*. En el tercer parámetro indicamos a la función que la referencia al objeto desde la tabla ROT *no cuenta* para el mecanismo de destrucción automática. Esto se conoce como *registro débil*; si utilizáramos el *registro fuerte*, este nuevo objeto ya tendría dos referencias, y cuando el cliente terminara de trabajar con él, seguiría existiendo, por causa de la ROT. Finalmente, el último parámetro nos devuelve una especie de *resguardo* para cuando queramos eliminar el objeto de la ROT.

Precisamente, ese es el papel del destructor *Destroy*, donde se ejecuta la función inversa a la de registrar objetos: *RevokeActiveObject*.

## Crear o reciclar

¿Quiere decir entonces que podemos crear un solo objeto de la clase *MessageTrap*? Pues no, no se trata de eso. Si llamamos a *CreateComObject* dos veces, incluso dentro del mismo ejecutable, seguiremos obteniendo dos instancias independientes; eso sí, en dos procesos diferentes. El truco hay que complementarlo en las aplicaciones clientes, modificando la forma en que se crean las instancias de la aplicación.

Cree una nueva aplicación, que vamos a utilizar como cliente. Prepare el formulario principal para que tenga la siguiente apariencia:



El botón que dice *Create new message sender* tiene la función de lanzar otra instancia del cliente, para que podamos probarlo con mayor facilidad. Su respuesta al evento *OnClick* debe ser:

```

procedure TwndMain.bnLaunchClick(Sender: TObject);
begin
    WinExec(PChar(Application.ExeName), SW_SHOWNORMAL);
end;

```

Tenemos que añadir la unidad *MsgTrap\_TLB* al proyecto, e incluirla en la cláusula *uses* de la sección de interfaz, para que podamos declarar la siguiente variable en la parte **private** de la clase del formulario principal:

```
type
  TwndMain = class(TForm)
    // ...
  private
    FMsgTrap: IMessageTrap;
    // ...
  end;
```

También es muy sencilla la respuesta al *OnClick* del botón *Send*:

```
procedure TwndMain.bnSendClick(Sender: TObject);
begin
  FMsgTrap.Catch(edMessage.Text);
end;
```

La parte interesante se ejecuta durante el evento *OnCreate* del formulario:

```
procedure TwndMain.FormCreate(Sender: TObject);
var
  Unk: IUnknown;
begin
  if Succeeded(GetActiveObject(Class_MessageTrap, nil, Unk)) and
    (Unk <> nil) then
    FMsgTrap := Unk as IMessageTrap
  else
    FMsgTrap := CoMessageTrap.Create;
end;
```

La función *GetActiveObject* recupera el puntero de interfaz que hemos almacenado antes con *Register.ActiveObject*. En realidad, se recupera un puntero “equivalente”, no el original; recuerde que el objeto se encuentra en el espacio de memoria de otro proceso. Si *GetActiveObject* encuentra el objeto, lo almacena para utilizarlo más adelante, pero en caso contrario, hay que crear un objeto nuevo.

En esta variante del ejemplo, el último que sale apaga las luces. Quiero decir, que al haber registrado el objeto en la ROT con *ACTIVEOBJECT\_WEAK*, éste se destruye cuando no hay clientes que hagan referencia a él. Por supuesto, este comportamiento puede modificarse.

Con pequeñas variaciones sin importancia, el algoritmo anterior es el que he encontrado en todos los libros y artículos que hablan sobre la ROT. Sin embargo, la técnica deja mucho que desear: entre la pregunta (¿hay ya un objeto registrado?) y la acción (¡lo creo yo mismo!) hay un intervalo de tiempo. Es muy difícil que ocurra, pero la planificación de tareas puede hacer que entre esas dos llamadas, otro proceso interroge la ROT, cree una instancia de la clase y la registre. En circunstancia reales, es extremadamente improbable que alguien tenga tan mala suerte... pero *shit happens*, como solía decir Murphy antes de que lo atropellara el elefante. Si le preocupa este problema, puede recurrir a un *mutex* o a un semáforo para “agrupar” en una sección

protegida las operaciones de interrogación y creación. Por supuesto, no nos valen las secciones críticas de Windows, porque la sincronización debe tener lugar a nivel de la máquina, y estos objetos sólo son válidos dentro de un mismo proceso.

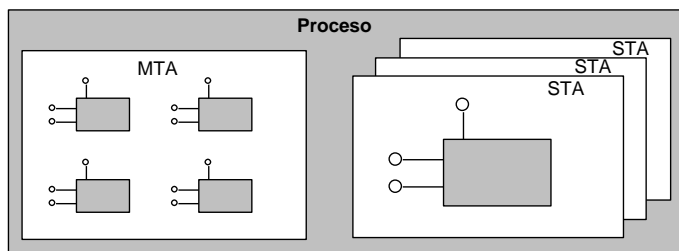
## Apartamentos

Ahora me toca explicar la parte peor explicada de COM: los modelos de concurrencia. Prometo, sin embargo, que si comenzamos la explicación por los apartamentos, no le dolerá la cabeza cuando terminemos. Para empezar, ¿qué es un apartamento? La mejor definición posible es una de tipo funcional, aunque sea poco ortodoxa, mostrando unas pocas reglas:

- 1 Un apartamento es un contexto dentro del cual se ejecuta un hilo.
- 2 Un hilo “entra” dentro de un apartamento cuando llama a una de las funciones que inicializa el sistema COM: *CoInitialize*, *CoInitializeEx* o *OleInitialize*.
- 3 Un hilo “abandona” el apartamento cuando se desconecta del sistema COM, llamando a *CoUninitialize* o a *OleUninitialize*.
- 4 Un hilo sólo puede pertenecer a un apartamento. Pero si el hilo no utiliza COM, entonces no pertenece a ninguno.

Existen tres tipos de apartamentos, aunque para simplificar inicialmente la explicación me centraré en dos de esos tipos: el *Single Threaded Apartment*, o apartamento de un solo hilo, y el *Multiple Threaded Apartment*... que ya imaginará lo que significa. Es costumbre referirse a ellos con las siglas STA y MTA.

Dentro de un elitista STA se admite un único hilo; si un hilo pide su admisión dentro de un STA, con toda garantía se fabrica uno a su medida. En cambio, un MTA más que un apartamento parece un campo de refugiados: pueden entrar cuantos hilos lo pidan, siempre que pertenezcan todos al mismo proceso. Y sólo puede haber un MTA por proceso.



Como es de sospechar, el tipo de apartamento que solicita un hilo viene determinado por parámetros que se pasan a las funciones de inicialización. Veamos, por ejemplo, el prototipo de *CoInitializeEx*:

```
function CoInitializeEx(pvReserved: Pointer;
    coInit: Longint): HRESULT; stdcall;
```

El primer parámetro no sólo está reservado, sino que además es inútil; al menos de momento. Es el segundo parámetro el que nos interesa, porque podemos pasarle una combinación de las siguientes constantes:

```
const
    COINIT_MULTITHREADED      = 0;
    COINIT_APARTMENTTHREADED = 2;
    COINIT_DISABLE_OLE1DDE   = 4;
    COINIT_SPEED_OVER_MEMORY  = 8;
```

A su vez, sólo nos interesan por ahora las dos primeras constantes, y creo que su significado es evidente. ¿Y qué pasa con *CoInitialize* y *OleInitialize*, que sólo tienen el primer parámetro? Estas funciones existen por compatibilidad con el pasado, con el maldito pasado; en COM, quiere decir que ambas meten al hilo dentro de un STA.

¿Cuándo es que se llama a *CoInitializeEx*, o a alguna de sus hermanas? Habrá notado que no hemos tenido que hacerlo explícitamente, al menos en los ejemplos que hemos visto hasta ahora. En una aplicación GUI, por ejemplo, primero se efectúa la inicialización de todas las unidades que tengan código de inicialización, y lo usual es que la primera línea de código que se ejecute después sea nuestra vieja conocida, la llamada al método *Initialize* del objeto global *Application*:

```
begin
    Application.Initialize;           // iii A Q U I !!!
    Application.CreateForm(TForm1, Form1);
    Application.Run;
end.
```

Pero la implementación de *Initialize* es decepcionantemente simple: comprueba el contenido de una variable global, *InitProc*, que puede contener un puntero a un procedimiento, y a ejecutar ese procedimiento si el puntero no es nulo.

```
procedure TApplication.Initialize;
begin
    if InitProc <> nil then TProcedure(InitProc);
end;
```

Si nuestro proyecto menciona, en alguna de las cláusulas **uses** de alguna de sus unidades, la unidad *ComObj*, entonces se ejecuta el código de inicialización de *ComObj* antes de llegar a la ejecución de *Initialize*:

```
initialization
begin
    // ... unas cuantas tonterías preliminares ...
    if not IsLibrary then begin
        SaveInitProc := InitProc;
        InitProc := @InitComObj;
    end;
end;
```

Y, ¡finalmente!, la llamada a *CoInitializeEx* está oculta dentro del procedimiento *InitComObj*. Todo este juego del gato y del ratón existe para que usted, el erudito y



avisado programador que sabe que no todos los apartamentos se pueden medir en metros cuadrados, pueda modificar una variable global llamada *CoInitFlags* antes de que el proyecto ejecute el método *Initialize* de la aplicación. La variable en cuestión se encuentra dentro de *ComObj*:

```
begin
    CoInitFlags := COINIT_MULTITHREADED;      // ... por ejemplo
    Application.Initialize;
    Application.CreateForm(TForm1, Form1);
    Application.Run;
end.
```

Resumiendo: los servidores locales y remotos pueden indicar el modelo de concurrencia de sus objetos controlando las llamadas a *CoInitializeEx*. Lo mismo sucede con las aplicaciones ejecutables que actúan como clientes. Pero tenemos un servidor dentro de una DLL, encontraremos que sus clientes tienen que haber llamado a *CoInitializeEx* antes para que pueda funcionar COM. Por lo tanto, los servidores dentro del proceso no tienen la libertad de elegir el apartamento en que van a residir. Al menos, no lo pueden hacer mediante las funciones de inicialización...

En compensación, a los servidores dentro del proceso se les permite indicar el tipo de apartamento que prefieren en el Registro de Windows, en la clave *Threading Model* que se ubica bajo su identificador de clase. Ahora veremos qué se gana al expresar esa preferencia.

## Compatibilidad entre apartamentos

Quiero darle las gracias por su paciencia por llegar hasta aquí, porque es cierto que toda la explicación anterior sobre los apartamentos no aclara gran cosa sobre cómo funciona la concurrencia dentro de una aplicación COM. Pero ahora estamos en condiciones de aclarar este importante punto. Le he contado antes que cuando un objeto reside en una DLL, el cliente puede acceder a sus interfaces directamente. También le he dicho que, cuando el servidor es un ejecutable con su propio espacio de memoria, la comunicación tiene lugar a través de un intermediario. Ya podemos quitarle el velo a la regla que COM realmente utiliza:

*“La comunicación sin intermediarios es posible solamente cuando el objeto COM y su cliente residen en un mismo apartamento”*

Si el objeto reside en un servidor ejecutable, es evidente entonces que todos sus clientes residirán en un apartamento completamente diferente. Por lo tanto, era completamente cierto que siempre se utiliza un *proxy* para servidores fuera del proceso.

¿Y si el objeto ha sido implementado dentro de una DLL? En ese caso, si no adoptásemos medidas especiales, el objeto tendría que ejecutarse *siempre* dentro del mismo hilo del cliente, y pertenecer por lo tanto al mismo apartamento. ¿Es eso bueno o

malo? Sería bueno desde el punto de vista de la eficiencia, porque el cliente podría trabajar directamente con el objeto. Pero podría ser malo si el programador no hubiese tomado medidas especiales respecto a la concurrencia. Por ejemplo, si el cliente decidiese entrar en el MTA del proceso, podría pasarle el puntero de interfaz a otros hilos del mismo MTA, y podrían surgir conflictos con el acceso concurrente a los datos internos del objeto. Por supuesto, este problema tendría fácil solución si el programador lo hubiera tenido en cuenta, y si no importase los recursos adicionales que consumirían las llamadas a funciones de sincronización.

Resumamos: no todos los objetos COM están preparados para vivir dentro de un apartamento MTA. Por este motivo, los servidores en DLLs deben marcar su modelo de concurrencia preferido en el Registro de Windows. Cuando un cliente pide a Windows que cree una instancia de una clase COM dentro del proceso, el sistema operativo comprueba si el modelo marcado para la clase coincide con el tipo de apartamento que está ejecutando el cliente. Si el cliente se ejecuta dentro de un STA y la clase lo acepta, COM construye el objeto, pide la interfaz por omisión y la pasa sin más al cliente. Si el modelo es incompatible con el apartamento del cliente, entonces COM debe aislar el objeto dentro de un apartamento creado al vuelo. En tal caso, se construye un *proxy* y se pasa la referencia al mismo al cliente. Todo esto sucede de manera automática:

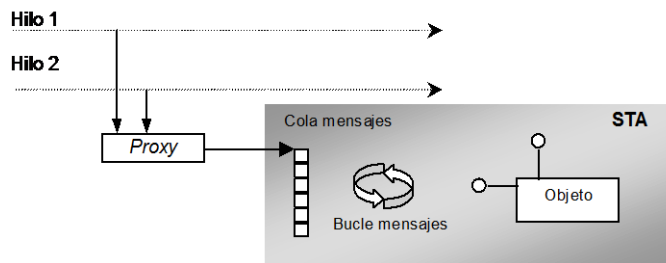
Modelo	Significado
<i>Single</i>	El programador de la clase no ha querido preocuparse, en modo alguno, de los problemas de concurrencia, y pide al sistema operativo que sea bueno con él, y que llame a sus métodos de uno en uno... ¡a nivel de toda la aplicación! Es decir, no existe concurrencia entre instancias paralelas de la clase COM.
<i>Apartment</i>	El objeto debe vivir en un STA, para que sólo puedan realizarse llamadas al mismo desde un mismo hilo. Esta condición garantiza al programador que no necesitará preocuparse por el acceso concurrente a los datos internos del objeto. En cambio, si queremos acceder a variables globales, tenemos que utilizar funciones de sincronización.
<i>Free</i>	El programador de la clase es un paranoico que se ha encargado de sincronizar el acceso tanto a las variables globales como a las de instancia. De este modo, varios hilos clientes pueden trabajar simultáneamente con el mismo servidor.
<i>Both</i>	El objeto puede ejecutarse en un STA o en un MTA. Sólo posible con una condición adicional: que ninguno de sus métodos admita un puntero de interfaz a un objeto implementado en el lado cliente.
<i>Neutral</i>	Esta es una novedad de COM+. Si un objeto dentro del proceso especifica esta modelo, para la ejecución de sus métodos se utiliza un apartamento especial creado por el sistema, conocido como NTA. Su ventaja principal es que no es necesario cambiar el hilo activo cuando el modelo del cliente no es soportado directamente por el servidor.

## Bombas de mensajes

Para no divagar, pongamos un ejemplo muy concreto: un cliente que se ejecuta en un MTA carga un servidor dentro del proceso que contiene una clase programada por alguien que no está muy al día con la programación multihilos. ¡Ojo!, que la frase anterior la digo sin retintín: no abundan precisamente los libros sobre estos temas. El programador, sin embargo, ha tenido cuidado de no tocar variables globales desde los métodos de la clase. Revisando la ayuda en línea, o la tabla de la sección anterior, llega a la conclusión de que su clase debe ejecutarse en el modelo *Apartment*, dentro de un STA.

Como decíamos, en este caso particular el cliente habita dentro de un MTA; esto significa que, si el cliente quisiera, podría pasar el puntero de interfaz al objeto a otro de los hilos de la aplicación, que pueden estar ejecutándose dentro del MTA. El peligro consiste entonces en que dos de esos hilos ejecuten simultáneamente métodos del mismo objeto. Pero COM detecta la incompatibilidad entre apartamentos, y crea un STA separado, esconde el objeto tras un *proxy* y se entrega un puntero a éste al cliente. Gato por liebre.

La pregunta es: ¿cómo se las arregla entonces el STA para evitar llamadas concurrentes sobre ese objeto? Y la respuesta es descaradamente simple: Windows crea una ventana oculta dentro del apartamento, y hace que las llamadas a métodos de los clientes se traduzcan en *mensajes* de Windows. Como sabemos, los mensajes suelen almacenarse en una cola, desde la cuál son extraídos y procesados *de uno en uno* por medio de un bucle de mensajes.



Evidentemente, cuando hablamos de un servidor dentro del proceso es el propio Windows quien se encarga de crear un hilo y alojarlo en el nuevo apartamento, de preparar la cola de mensajes y de implementar el bucle que bombea los mensajes fuera de la cola. Pero, ¿qué pasa con los servidores que se implementan como ejecutables? Para simplificar, supongamos que el servidor aloja una sola clase COM. Y comencemos suponiendo que queremos que las instancias de dicha clase se ejecuten dentro de un MTA.

En primer lugar, somos nosotros los responsables de que el hilo principal del servidor pertenezca al MTA, modificando las opciones de la llamada a *CoInitializeEx*.

Como recordará, esta función se llama dentro del método *Initialize* de la clase *Application*, y debemos asignar las opciones de inicialización en la variable global *CoInitFlags*, definida en la unidad *ComObj*:

```
begin
  CoInitFlags := COINIT_MULTITHREADED;      // ... por ejemplo
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

A partir de ese momento, no tenemos que preocuparnos, porque COM se encarga del resto. Cuando el servidor reciba peticiones concurrentes provenientes de distintos clientes o distintos hilos, el propio COM será quien prepare hilos “al vuelo”, implementados por el servicio RPC, y los asignará dinámicamente a las peticiones que vayan apareciendo. Los hilos son gestionados por una caché: cuando termina una petición, el hilo no se destruye inmediatamente, pues eso sería un desperdicio estúpido. El hilo se aparca en la caché, y si aparece otra petición puede volver a utilizarse; si pasa mucho tiempo hasta la próxima petición, el hilo caduca y muere.

#### ADVERTENCIA

Sin embargo, lo que quise decir antes es que no tenemos que preocuparnos *mucho*. Hay un problema, y es que *no conozco* la forma de controlar el tamaño de esa caché de hilos. Hasta donde he podido indagar, el número máximo de hilos paralelos por servidor es una cantidad fija, y al parecer no muy grande que digamos.

## Experimentos con la concurrencia

¿Y qué pasa si el servidor entra por su propio pie dentro de un STA? Prepárese para una desagradable sorpresa: si no tomamos medidas especiales, COM utilizará la cola de mensajes de la aplicación que actúa como servidor para que ejecutemos secuencialmente los métodos que disparen los clientes. De esa manera, nunca se ejecutarán en paralelo dos métodos del mismo objeto. Es más, ¡nunca se ejecutarán en paralelo dos métodos, aunque pertenezcan a dos objetos diferentes y hayan sido solicitados por dos clientes independientes! Todas las peticiones de los potencialmente numerosos clientes de la clase COM serán atendidas rigurosamente por orden de llegada. Vaya cuello de botella, ¿verdad?

Puedo comprender que no se fie de mi palabra, y he preparado un experimento para demostrarlo. En el CD-ROM viene un ejemplo de un servidor fuera del proceso que implementa una clase a la que he llamado *DiskFiller*: el “rellena discos”. La clase implementa una interfaz llamada *IDiskFiller* que contiene un solo método:

```
[
  uuid(AFD34FC1-BE9A-11D5-8943-00C026263861),
  version(1.0),
  helpstring("Interface for DiskFiller Object"),
  oleautomation
]
```

```

interface IDiskFiller: IUnknown
{
    [id(0x00000001)]
    HRESULT _stdcall CreateFile([in] long Timeout );
};

```

El objetivo de *CreateFile* es escribir tonterías en un fichero durante un tiempo determinado, que expresaremos en milisegundos. En concreto, escribe líneas de 80 caracteres de longitud con caracteres aleatorios, pero añadiendo en el principio de línea la hora exacta en que se escribe. El nombre del fichero se genera aleatoriamente. Pero será mejor que explique el funcionamiento de *CreateFile* mostrando su código fuente:

```

function TDiskFiller.CreateFile(Timeout: Integer): HRESULT;
var
    T0: Cardinal;
    S: string;
    I, J: Integer;
    F: TextFile;
    Aux: Char;
begin
    try
        AssignFile(F, 'MULTI' +
            FormatFloat('000', Random(1000)) + '.TXT');
        Rewrite(F);
        try
            T0 := GetTickCount;
            // Vapulear el disco durante Timeout milisegundos
            while GetTickCount - T0 < Timeout do begin
                // Crear una cadena aleatoria
                SetLength(S, 80);
                for I := 1 to Length(S) do
                    S[I] := Chr(Ord(' ') + Random(127 - 32));
                // ;Sí, es el bubblesort!
                for I := 1 to Length(S) - 1 do
                    for J := I + 1 to Length(S) do
                        if S[I] > S[J] then begin
                            Aux := S[I]; S[I] := S[J];
                            S[J] := Aux;
                        end;
                // Añadir la hora a la cadena
                S := FormatDateTime('hh:nn:ss.zzz ', Now) + S;
                // Escribir la cadena en una línea
                WriteLn(F, S);
            end;
        finally
            CloseFile(F);
        end;
        Result := S_OK;
    except
        Result := S_FALSE; // No dejar escapar excepciones
    end;
end;

```

Antes de empezar a hablar pestes sobre mi estilo de programación, quiero que sepa que el objetivo de *CreateFile* es perder la mayor cantidad de tiempo posible. Mientras más rápida sea nuestra implementación, peor para nosotros, pues el método grabará más líneas en el disco, los ficheros resultantes serán más grandes y correremos el

riesgo de quedarnos sin espacio por culpa de una prueba tonta. Por eso, cada vez que se genera una línea me lanzo a ejecutar el *bubblesort*, un algoritmo muy ineficiente de ordenación, para arañar algunos milisegundos de procesamiento. No he querido ejecutar la función *Sleep*, porque se distorsionarían los resultados.

Una vez que tengamos listo el servidor, pasamos a implementar un cliente sencillo a partir de una nueva aplicación. Añadimos la unidad con las declaraciones de la biblioteca de tipos al nuevo proyecto, para poder declarar una variable de interfaz en la sección **private** de la clase del formulario principal:

```
private
    DiskFiller: IDiskFiller;
```

Asignamos un objeto recién creado en el evento *OnCreate* del formulario:

```
procedure TwndMain.FormCreate(Sender: TObject);
begin
    DiskFiller := CoDiskFiller.Create;
end;
```

Y, por último, añadimos un botón al formulario para que llame al único método del objeto remoto:

```
procedure TwndMain.bnCallServerClick(Sender: TObject);
begin
    bnCallServer.Enabled := False;
    try
        bnCallServer.Update;
        DiskFiller.CreateFile(5000)
    finally
        bnCallServer.Enabled := True;
    end;
end;
```

¿En qué consiste el experimento? Con la ayuda del Explorador de Windows, lance dos veces la aplicación cliente. Como el servidor permite múltiples instancias de la clase COM dentro de un mismo proceso, se crea una sola instancia de la aplicación servidora. A continuación, pulse los botones de ambos formularios, con la mayor rapidez que pueda. Esto provocará dos llamadas casi paralelas al método *CreateFile* de dos objetos COM diferentes. Como *CreateFile* va a tardar 5 segundos en ejecutarse, puede ocurrir una de estas dos situaciones alternativas:

- 1 Las dos ejecuciones transcurren en paralelo.
- 2 El sistema hace esperar a la segunda llamada, hasta que termine la primera.

Por lo tanto, hay que buscar la respuesta en los ficheros generados. A la izquierda incluyo un fragmento del fichero generado por la primera instancia, y a la derecha, un trozo del segundo fichero:

21:21:45.740	!!"#\$%&(+-.//00	21:21:50.740	!#&)*///02447899
21:21:45.740	"\$%' +, --/013456	21:21:50.740	!"#\$%&&' (, -.

```

21:21:45.740  $&' ((+002234477      21:21:50.790  ""&'***, 0147788
                ...
21:21:50.740  !""$%&() **+, , .      21:21:55.730  !""""%")+..22233
21:21:50.740  !""""###'() *+-. /0    21:21:55.840  !"#&'(( (.../0

```

¿Se da cuenta que los intervalos de ejecución no se solapan? El servidor que ha creado el asistente de Delphi sólo es capaz de ejecutar un método a la vez. ¡Vaya plasta de servidor!

## La salvación es una fábrica de clases

Por supuesto, existen soluciones para el problema anterior... que realmente no es un problema. Lo que pasa es que COM espera que sea el desarrollador quien se ocupe de lanzar hilos adicionales cuando lo crea necesario. Pero la falta de una buena documentación sobre estos asuntos cogió a muchas personas de sorpresa. Los servidores Midas que se creaban con Delphi 3, por ejemplo, no se ocupaban de la creación de hilos... y no se advertía al programador del peligro. Incluso en Delphi 4, la solución que voy a presentar aparecía en uno de los ejemplos; no como parte de las clases predefinidas, como ocurre en Delphi 5 y 6.

La idea consiste en utilizar una clase diferente como fábrica de clases. Como recordará, en dependencia de la clase utilizada como ancestro para implementar la clase COM, Delphi elige una clase auxiliar diferente para crear las instancias a petición del cliente. Para el ejemplo que estamos viendo, el asistente utiliza *TTypedComObject* como clase base, y la fábrica de clases que le corresponde es *TTypedComObjectFactory*. En el siguiente capítulo estudiaremos un tipo muy común de servidores para el que Delphi ofrece un asistente especializado. Ese asistente utiliza *TAutoObject* como clase base, y *TAutoObjectFactory* como fábrica de clases.

Las dos clases mencionadas como fábricas de clases tienen una implementación muy sencilla. Cuando COM necesita una instancia de la clase que gestionan, ejecuta el método común *CreateInstance* dentro del hilo principal del servidor y, dentro de esa misma llamada, se crea la instancia, se extrae la interfaz solicitada y se devuelve a COM., para que éste a su vez entregue el correspondiente *proxy* al cliente.

Pero es posible diseñar un algoritmo de creación más sofisticado. Al ejecutarse *CreateInstance*, la fábrica de clases podría lanzar un nuevo hilo y delegar en él la creación del objeto COM. Como estamos hablando de un servidor basados en STAs, el hilo en que se ejecuta *CreateInstance* y el nuevo son diferentes, por lo que es necesario transformar el puntero de interfaz al objeto creado mediante el *marshaling*, para poder entregarlo a *CreateInstance*. No voy a entrar en los detalles sucios de la técnica: la clase que necesitamos se llama *TComponentFactory* y está definida en la unidad *VclCom*. Sólo que tenemos un problema: *TComponentFactory* sólo funciona cuando la clase COM utiliza *TComponent* como clase base.

La clase mencionada se utiliza para crear módulos de datos remotos, que son las clases COM utilizadas por Midas (está bien, por DataSnap). Se dará cuenta de que *TComponentFactory* es un parche de última hora para una dificultad con la que inicialmente no se contaba.

Por todo lo dicho, si queremos aplicar la solución anterior a nuestro ejemplo, vamos a tener que retocar un poco las declaraciones dentro de la unidad que implementa la clase *DiskFiller*. El siguiente fragmento de código muestra cómo utilizar las directivas de compilación opcional para mantener dos alternativas de implementación del servidor:

```
type
{$IFDEF BIEN}
    TBase = class(TComponent);
    TFactory = class(TComponentFactory);
{$ELSE}
    TBase = class(TTypedComObject);
    TFactory = class(TTypedComObjectFactory);
{$ENDIF}

TDiskFiller = class(TBase, IDiskFiller)
protected
    function CreateFile(Timeout: Integer): HRESULT; stdcall;
end;
```

Por supuesto, si queremos utilizar hilos paralelos debemos incluir al principio de la unidad una directiva como la siguiente:

```
{ $DEFINE BIEN }
```

También tendremos que modificar la cláusula de inicialización de la unidad, para registrar la nueva fábrica de clases:

```
initialization
    Randomize;
    TFactory.Create(ComServer, TDiskFiller, Class_DiskFiller,
        ciMultiInstance, tmApartment);
end.
```

Estos son los resultados del nuevo experimento:

22:08:43.690	!#\$%+,//1559::;	22:08:44.070	"##%'()*++,-/00
22:08:43.690	!#\$%+,//1569::;	22:08:44.070	!\$&'()*+,-//02
22:08:43.690	#%'',-12227899	22:08:44.070	"#\$' ' ( ) , , . 01
	...		...
22:08:48.690	!"\$ ( ) + - . / 1125	22:08:49.070	"###' ' ' -- . 012234
22:08:48.690	!\$%&&& ( ) * + , --- .	22:08:49.070	!"#\$%\$ ( ( + - . / 03

Como verá, una de las llamadas comenzó en el segundo 43 y terminó en el 48, mientras que la otra comenzó en el 44 y terminó en el 49. Es decir, ambas ejecuciones tuvieron lugar simultáneamente. Ya nuestro servidor puede atender de forma concurrente peticiones procedentes de diferentes clientes. De todos modos, si un



mismo objeto recibe dos peticiones simultáneas seguirá poniéndolas en cola, y atenderá la segunda sólo cuando haya acabado con la primera.

## El modelo libre

Claro, también podríamos cambiar el modelo de concurrencia del servidor: en vez de *Apartment*, usaríamos *Free*. Voy a mostrarle cómo hacer los cambios apropiados sobre la aplicación existente.

En este caso, podríamos dejar la fábrica de clases original, *TTypedComObjectFactory*. Solamente necesitaremos retocar esta vez la cláusula de inicialización de la unidad:

```
initialization
  Randomize;
  CoInitFlags := COINIT_MULTITHREADED;
  TFactory.Create(ComServer, TDiskFiller, Class_DiskFiller,
    ciMultiInstance, tmFree);
end.
```

Hay dos modificaciones:

- 1 Se asigna la constante *COINIT\_MULTITHREAD* a la variable global *CoInitFlags*, para que el servidor entre inicialmente en un MTA. Para ser honestos, esta asignación no es necesaria. El constructor de la fábrica de clases se encargará de hacerlo, en dependencia del valor que pasemos en su quinto parámetro.
- 2 Se modifica el último parámetro del constructor de la fábrica de clases.

Compile y ejecute la nueva versión del servidor, y comprobará que, al menos cuando hay sólo dos clientes, los métodos pueden ejecutarse en paralelo sin intervención nuestra.

### ADVERTENCIA

A pesar de todo, me preocupa el que no sea posible controlar el comportamiento de la caché de hilos del servicio RPC. Por eso no presenté esta solución en primer lugar.

Es importante que comprenda además, que cada objeto *DiskFiller* está siendo utilizado solamente desde un hilo de su cliente. En caso contrario, tendríamos que añadir código para sincronizar el acceso a las variables internas del objeto. Da la casualidad, sin embargo, que la clase de Delphi *TDiskFiller* no define otras variables dentro de la clase que las que hereda, y estas ya se encuentran protegidas por la propia VCL. Por otra parte, *CreateFile* sólo utiliza parámetros y variables locales, que no necesitamos sincronizar.



# Automatización OLE

LA CHAPUZA MÁS ESPANTOSA DE LA PROGRAMACIÓN en Windows se denomina *Automatización OLE*, y es una técnica que permite manejar métodos de objetos mediante una interfaz de macros. ¿Su objetivo? Permitir que los lenguajes interpretados, a los que son tan adictos en Microsoft, puedan aprovechar una mínima parte de la potencia de COM. Veremos cómo esta técnica se amplía y mejora mediante el uso de *interfaces duales*. El plan de trabajo es sencillo: en la primera parte del capítulo actuaremos como clientes, o como se dice en la jerga, *controladores* de automatización. Y terminaremos desarrollando servidores OLE.

## ¿Por qué existe la Automatización OLE?

La *automatización OLE*, *OLE Automation* en su idioma original, está basada en un tipo de interfaz predefinido por COM, llamado *IDispatch*. Esta interfaz permite que sus clientes ejecuten macros complejas implementadas dentro de objetos COM; de hecho, la automatización OLE sustituye al viejo mecanismo de ejecución de macros de DDE. Pero, ¿no es obsoleto y peligroso recurrir a macros para el control de objetos? Por supuesto que sí: creo que a estas alturas a nadie se le ocurriría defender la falta de comprobación estática de tipos, excepto a los adictos a Visual Basic. Y fue principalmente por culpa de Visual Basic que Microsoft diseñó *IDispatch*. Cuando VB se actualizó para generar programas en 32 bits, Microsoft sustituyó el anterior modelo de componentes VBX por los recién estrenados controles OCX (todavía los publicistas no habían inventado la palabreja *ActiveX*), que se manejaban a través de la obligatoria interfaz *IDispatch*.

En algún momento de su evolución, Visual Basic introdujo cambios que le permitieron trabajar directamente con las *v-tables* de COM... pero otros lenguajes recogieron el testigo y siguieron dependiendo de *IDispatch*. En primer lugar, las mutaciones de Visual Basic que utilizan las aplicaciones de Office. Pero también llegó VBScript, ampliamente utilizado para la creación de páginas ASP, y JScript, la versión de JavaScript de Microsoft. Todos estos son lenguajes interpretados, en los que sería demasiado complicado implementar el acceso directo a las interfaces *v-tables* de COM.

He mencionado aplicaciones que actúan como clientes de automatización OLE; servidores que soportan esta forma de trabajo hay muchos. Están todas las aplicacio-

nes de Office; muchos de los ejemplos de automatización OLE que aparecen en los libros de Delphi utilizan estos servidores. ADO es otra interfaz muy importante que puede ser utilizada a través de interfaces *IDispatch*. Algunas características de la administración de SQL Server pueden ser controladas también de esta forma; y el propio Transact SQL, permite utilizar objetos de automatización desde sus *scripts*.

... y, aunque le parezca increíble, Delphi es un cliente habitual de la interfaz *IDispatch*. Cuando programamos servidores de capa intermedia, utilizando DataSnap, estamos creando clases COM que implementan interfaces que desciende de *LAppServer*, una interfaz definida por Borland y que desciende a su vez de *IDispatch*. Siempre que es posible, las aplicaciones que trabajan con servidores de capa intermedia deben intentar acceder a los mismos mediante el tipo de interfaz correcto, para mayor seguridad y rapidez. Pero, como veremos al estudiar DataSnap, hay ocasiones en que esto no es posible: el caso típico ocurre cuando la aplicación cliente debe conectarse a través de Internet con su servidor. En esos casos, Borland suministra mecanismos que realizan una especie de *marshaling* limitado, que sólo soporta la interfaz *IDispatch*.

## La interfaz *IDispatch*

No es que tenga inclinaciones sádicas, pero es mi obligación mostrarle la declaración de *IDispatch* en IDL:

```
[object, uuid(00020400-0000-0000-C000-000000000046)]
interface IDispatch : IUnknown
{
    HRESULT GetTypeInfoCount([out] UINT *pctinfo);
    HRESULT GetTypeInfo(
        [in] UINT iTInfo,
        [in] LCID lcid,
        [out] ITypeInfo **ppTInfo);
    HRESULT GetIDsOfNames(
        [in] REFIID riid,
        [in, size_is(cNames)] LPOLESTR *rgszNames,
        [in] UINT cNames,
        [in] LCID lcid,
        [out, size_is(cNames)] DISPID *rgid);
    HRESULT Invoke(
        [in] DISPID id,
        [in] REFIID riid,
        [in] LCID lcid,
        [in] WORD wFlags,
        [in, out] DISPPARAMS *pDispParams,
        [out] VARIANT *pVarResult,
        [out] EXCEPINFO *pExcepInfo,
        [out] UINT *puArgErr);
}
```

No se preocupe, yo mismo soy incapaz de recordar todos los parámetros de estas funciones. En realidad, nunca he utilizado directamente la interfaz *IDispatch*. Dada su complejidad, la mayoría de los lenguajes ofrece algún tipo de encapsulamiento para realizar llamadas a la misma. En Delphi y en Visual Basic, ese encapsulamiento nos puede llegar a engañar, de modo que no nos demos cuenta de que realmente estamos

utilizando macros. En otros lenguajes, como C++, existen clases especiales para esta tarea.

El núcleo de *IDispatch* es el método *Invoke*, que sirve para ejecutar una macro en el servidor. Para tal propósito, *Invoke* permite la especificación de parámetros de entrada y salida, de valores de retorno para funciones y la propagación de excepciones desde el servidor al cliente. Ahora bien, *Invoke* no utiliza cadenas de caracteres para especificar el método a ejecutar. Por el contrario, el método se identifica mediante un valor numérico, que se pasa en el parámetro *id*, de tipo *DISPID*. Lo mismo sucede con los nombres de parámetros. *Invoke* admite pasar parámetros por nombre a los métodos que activa, lo cual evita que tengamos que recordar siempre la posición exacta de un parámetro cuando la macro tiene muchos.

La función *GetIDsOfNames* es la encargada de traducir los nombres de métodos y de parámetros en códigos numéricos, para pasarlos posteriormente a *Invoke*. ¿Por qué la ejecución de una macro se realiza en estos dos pasos, traducción a identificador numérico y posterior ejecución? La razón es evidente: la lucha por la eficiencia. Si vamos a ejecutar un método mediante *Invoke* varias veces consecutivas, quizás dentro de un bucle, no queremos que en cada ejecución el objeto tenga que efectuar una larga serie de comparaciones de cadenas de caracteres; más rápido es comparar dos enteros.

¿Cómo se asignan estos valores enteros a los métodos de automatización? En el lenguaje IDL se utiliza un tipo especial de declaración: las interfaces de envío, o *dispatch interfaces*:

```
[ uuid(20D56981-3BA7-11D2-837B-0000E8D7F7B2) ]
dispinterface IGenericReport {
methods:
    [id(1)] void Print([in] BOOL DoSetup);
    [id(2)] void Preview();
}
```

Esta no es una interfaz, en el sentido normal de la palabra, sino que es una tabla para el consumo interno de una clase, que describe los métodos implementados a través del método *Invoke* de una “verdadera” interfaz *IDispatch*, y los identificadores de envío asociados. Podríamos entender la **dispinterface** como la parte del contrato que asegura la funcionalidad implementada dentro de una clase de automatización OLE. Y es necesaria porque no podemos deducir que una clase determinada da soporte a estos métodos observando solamente la declaración de *IDispatch*.

Las interfaces de envío de IDL tienen un tipo de datos equivalente en Delphi. Por ejemplo, la interfaz *IGenericReport* se traduciría del siguiente modo:

```
type
    IGenericReport = dispinterface
    ['{20D56981-3BA7-11D2-837B-0000E8D7F7B2}']
    procedure Print(DoSetup: WordBool); dispid 1;
    procedure Preview; dispid 2;
```

```
end;
```

A diferencia de los tipos de interfaz verdaderos, las interfaces de envío de Delphi no pueden mencionarse en la cabecera de una clase para ser implementadas. Sin embargo, más adelante veremos que sí pueden ser utilizadas por un cliente de automatización para acelerar la ejecución de métodos bajo ciertas condiciones.

## Interfaces duales

Inevitablemente, las llamadas a métodos por medio de *Invoke* son lentas e inseguras. La práctica ha demostrado, por otra parte, que todos los lenguajes que comienzan su existencia dependiendo de un intérprete, terminan al final incorporando un compilador más o menos feliz... o muriendo en el intento. La única excepción son quizás los lenguajes de *scripting* para Internet, pero habrá que cederle la palabra al Tiempo.

Por este motivo, la mayoría de los servidores de automatización utilizan una técnica conocida como *interfaces duales*. ¿Qué se le exige como mínimo a un servidor de automatización? Basta con que implemente *IDispatch*, y que su verdadera funcionalidad se implemente por medio de macros que se ejecutan a través de *Invoke*. Las interfaces duales plantean una exigencia adicional: que esos métodos macros puedan también ser ejecutados a través de una interfaz “normal”, derivada a partir de *IDispatch*.

Supongamos que tenemos que implementar una clase de automatización con la funcionalidad de un aparato eléctrico. Si nos atenemos al mínimo exigible, la clase podría implementar solamente la interfaz *IDispatch*, y dar soporte a través del código que asociemos a *Invoke* a los métodos descritos por esta interfaz de envío:

```
[ uuid(744839E0-DAF9-11D5-8943-00C026263861) ]
dispinterface IAparatoElectricoDisp {
methods:
    [id(1)] void Enchufar;
    [id(2)] void Apagar;
}
```

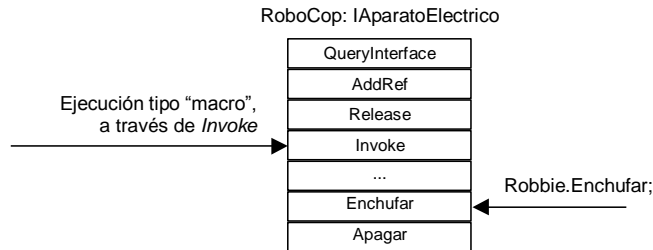
En cambio, si nos exigieran que la clase estuviera basada en una interfaz dual, tendríamos que derivar una nueva interfaz a partir de *IDispatch*, y añadirle los dos métodos antes mencionados. En IDL quedaría más o menos así:

```
[ uuid(744839E2-DAF9-11D5-8943-00C026263861) ,
    dual, oleautomation ]
interface IAparatoElectrico: IDispatch
{
    [id(0x00000001)]
    HRESULT _stdcall Enchufar( void );
    [id(0x00000002)]
    HRESULT _stdcall Apagar( void );
};
```

Note que la interfaz, además de descender de *IDispatch*, está marcada con el nuevo atributo **dual**. La información que antes contenía la interfaz de envío está ahora

presente dentro de la interfaz más tradicional. Delphi, por ejemplo, generaría también una declaración **dispinterface** a partir de la declaración anterior.

El siguiente esquema representa la *v-table* de un objeto que implementa la interfaz dual anterior:



Si el objeto es utilizado desde un lenguaje interpretado como VBScript, el código asociado a *Enchufar* se ejecutaría según la vía de acceso mostrada a la izquierda, como si se tratase de una macro, a través de *Invoke*. Pero desde Delphi sería preferible llamar a *Enchufar* a través de un puntero de tipo *IAparatoElectrico*, como se muestra en la vía de acceso de la derecha.

## Controladores de automatización con variantes

¿Puede Delphi utilizar el mecanismo de macros de *IDispatch*? La respuesta es afirmativa, pues esta funcionalidad se añadió en su segunda versión, cuando aún no existían los tipos de interfaz. La clave está en utilizar variables de tipo *Variant*. Como sabemos, estas variables pueden contener valores pertenecientes a los más diversos tipos y, ¡qué casualidad!, uno de los tipos admitidos es *IDispatch*. Ambos tipos son compatibles para la asignación en los dos sentidos:

```
var
  D: IDispatch;
  V: Variant;
begin
  V := D; D := V;
end;
```

Podemos obtener un puntero a la interfaz *IDispatch* de un objeto de automatización y almacenarlo en una variable de tipo *Variant* mediante una llamada a la función global *CreateOleObject*, declarada en la unidad *ComObj*:

```
function CreateOleObject(const ClassName: string): IDispatch;
```

Esta función utiliza el identificador de programa de la clase, en vez del identificador de clase. Su implementación no contiene misterio alguno:

```
function CreateOleObject(const ClassName: string): IDispatch;
var
```

```

    ClassID: TCLSID;
begin
    ClassID := ProgIDToClassID(ClassName);
    OleCheck(CoCreateInstance(ClassID, nil, CLSCTX_INPROC_SERVER or
        CLSCTX_LOCAL_SERVER, IDispatch, Result));
end;

```

Las diferencias respecto a *CreateComObject* son dos: primero hay que traducir el nombre de la clase, por medio del Registro, para obtener el identificador único de clase asociado, y luego se especifica la interfaz *IDispatch* en la llamada a *CoCreateInstance* para obtener un puntero a dicha interfaz. Me detengo en este código para aclarar que los identificadores alfabéticos de programas no están limitados a las clases que implementan automatización OLE, como he leído en algún lugar hace poco.

Una vez que tenemos la interfaz *IDispatch* bien oculta dentro del variante, podemos ejecutar los métodos de la clase aplicándolos sobre la variable de tipo *Variant*, como si ésta fuese un puntero de interfaz:

```

procedure TForm1.Button1Click(Sender: TObject);
var
    I: Integer;
    WordSvr: Variant;
begin
    WordSvr := CreateOleObject('Word.Basic');
    WordSvr.AppShow;
    WordSvr.FileNewDefault;
    for I := 0 to Memo1.Lines.Count - 1 do
        WordSvr.Insert(Memo1.Lines[I] + #13);
    WordSvr.FileSaveAs(ChangeFileExt(Application.ExeName, '.doc'));
end;

```

El ejemplo anterior es un clásico de los libros de Delphi. Hemos obtenido un puntero a un objeto de la clase *Word.Basic*, implementada dentro de Microsoft Word. Los métodos que se llaman a continuación son métodos exportados por esa clase. El primero, *AppShow*, hace visible a Word, el segundo, *FileNewDefault*, crea un fichero con propiedades por omisión, *Insert* inserta una línea y *FileSaveAs* guarda el fichero en el directorio de la aplicación.

Al terminar el método, no hay que destruir explícitamente al objeto creado, pues el compilador se encarga de hacerlo, llamando al destructor de *Variant* cuando desaparece la variable local *WordApp*. Si quisiéramos destruir el objeto explícitamente, tendríamos que asignar el valor especial *Unassigned* a la variable:

```
WordApp := Unassigned;
```

Es muy fácil cometer errores cuando se ejecutan métodos sobre una variable de tipo *Variant*, porque el compilador no realiza tipo alguno de verificación estática. Por ejemplo, podríamos añadir con toda tranquilidad la siguiente instrucción dentro del método mostrado:

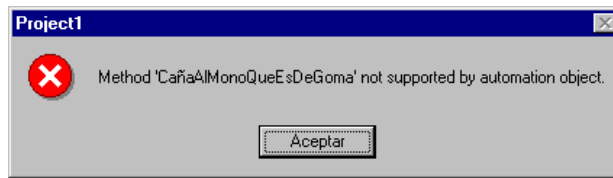


```

procedure TForm1.Button1Click(Sender: TObject);
var
    WordSvr: Variant;
begin
    WordSvr := CreateOleObject('Word.Basic');
    WordSvr.CañaAlMonoQueEsDeGoma(50, True);
end;

```

Compile esa instrucción y verá que Delphi no protesta por el método, que obviamente no existe, por la *N* que le hemos colado, ni tampoco por los parámetros que he inventado. Claro, cuando intentamos ejecutar el presunto método, saltan los fusibles:



Para que se haga una idea de lo que realmente hace Delphi con los falsos métodos aplicados a un *Variant*, es aconsejable que vea cómo se programa el mismo ejemplo en C++ Builder:

```

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Variant word = Variant::CreateObject("Word.Basic");
    word.Exec(Procedure("AppShow"));
    word.Exec(Procedure("FileNewDefault"));
    for (int i = 0; i < Mem1->Lines->Count; i++)
        word.Exec(Procedure("Insert") <<
            (Mem1->Lines->Strings[i] + "\n"));
    word.Exec(Procedure("FileSaveAs") <<
        ChangeFileExt(Application->ExeName, ".doc");
}

```

Note que *AppShow*, *FileNewDefault* y el resto de la tripulación son ahora pasados como parámetros a métodos definidos en la clase *Variant* de C++ Builder. Por supuesto, la técnica utilizada por Delphi es mucho más fácil de utilizar, pero puede conducir a malas interpretaciones sobre lo que realmente está sucediendo.

## Propiedades OLE y parámetros por nombre

Hay muchas más posibilidades sintácticas cuando se ejecutan métodos de automatización por medio de variantes. Mostraré un par de ejemplos utilizando el conjunto de clases que Microsoft recomienda utilizar a partir de Office 97; la clase que vimos en la sección anterior, *Word.Basic*, ha sido declarada obsoleta desde entonces. En la nueva jerarquía, en vez de definir un sinnúmero de métodos en una monstruosa clase solitaria, se reparten los métodos entre varias clases: un objeto raíz de clase *Application* contiene un puntero a una colección *Documents*, que contiene objetos de tipo

*Document*, etcétera, etcétera. El siguiente ejemplo es equivalente al de la sección anterior:

```
procedure TForm1.Button2Click(Sender: TObject);
var
    WApp, WDoc: Variant;
begin
    WApp := CreateOleObject('Word.Application');
    WApp.Visible := True;
    WApp.Documents.Add;
    WDoc := WApp.ActiveDocument;
    WDoc.Content.InsertAfter('Mira lo que hago con Word'#13);
    WDoc.SaveAs(
        FileName := ChangeFileExt(Application.ExeName, '.doc'));
end;
```

La primera novedad es el soporte para propiedades, que internamente se implementan mediante métodos de lectura y escritura. Los tipos variantes de Delphi nos permite trabajar con ellas directamente:

```
WApp.Visible := True;
WApp.Documents.Add;
```

La última instrucción es un ejemplo de cómo utilizar un parámetro con nombre:

```
WDoc.SaveAs(
    FileName := ChangeFileExt(Application.ExeName, '.doc'));
```

Resulta que el método *SaveAs*, de la clase *Document* de Word, tiene nada más y nada menos que once parámetros. De todos ellos, solamente nos importa el primero, pues los restantes van a utilizar valores por omisión. Podíamos haber escrito esa instrucción de esta otra forma, aprovechando que el parámetro que suministramos es el primero:

```
WDoc.SaveAs(ChangeFileExt(Application.ExeName, '.doc'));
```

Recuerde que esta rocambolesca sintaxis se traduce, en definitiva, a llamadas al método *GetIDsOfNames* y en operaciones sobre los parámetros de *Invoke*.

### ADVERTENCIA

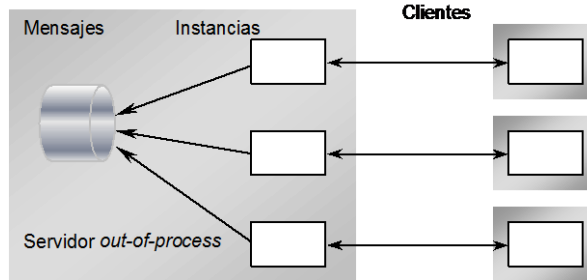
No se deje confundir por estos ejemplos, porque su único objetivo es mostrar cómo utilizar la automatización OLE a través de variantes con un servidor, Word, que está más extendido que un virus. Si quiere controlar las aplicaciones de Office, Delphi ofrece clases más convenientes para esta tarea, que mostraremos más adelante.

## Un ejemplo de automatización

Desarrollaremos ahora un pequeño servidor de automatización: un programa que simule el funcionamiento de un foro. Su función será recibir mensajes enviados como cadenas de caracteres desde sus clientes, y almacenarlos en memoria. Cada vez

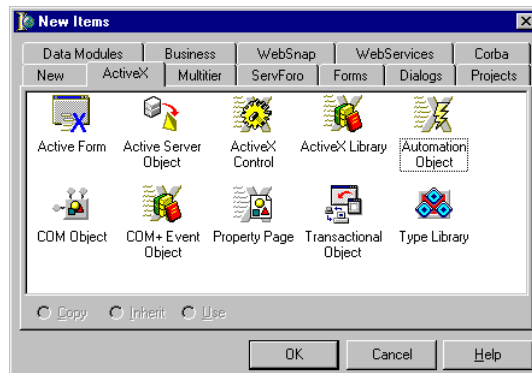
que reciba un mensaje, debe notificarlo a todos los clientes que estén utilizando el servidor.

Aunque haremos las pruebas del servidor y sus clientes utilizando un mismo ordenador, diseñaremos nuestro servidor para que pueda ejecutarse en un ordenador separado. La estructura del sistema será la siguiente:

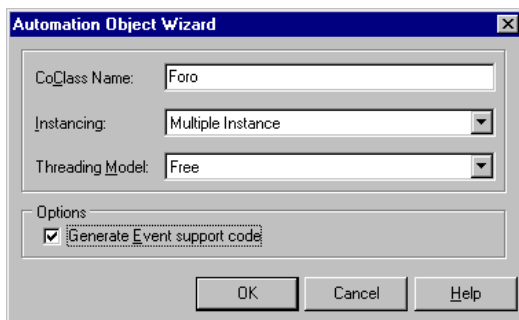


El servidor será de tipo ejecutable, con soporte para múltiples instancias de una clase dentro de un MTA. Puede parecer una elección extraña: ¿no sería mejor utilizar un solo objeto, al que accederían concurrentemente todos los clientes? Resulta que no. En primer lugar, *GetActiveObject*, la función que busca un objeto en la ROT, solamente funciona dentro un mismo ordenador. Se podría modificar el comportamiento de la fábrica de clases, de modo que la primera llamada a *CreateInstance* crease realmente un objeto, mientras que las siguientes llamadas devolviesen ese mismo objeto. Pero esto significaría alterar la semántica que COM espera de la fábrica de clases, y las consecuencias serían impredecibles.

Por otra parte, veremos que las instancias del servidor de mensajes son pequeñas, porque los datos se almacenan fuera, en variables globales: no pagaremos un coste demasiado alto por suministrar una instancia de la clase servidora por cada cliente. Finalmente, si hubiéramos elegido el modelo *Apartment* tendríamos que escoger entre utilizar un solo hilo para todas las peticiones, o un hilo por cada petición. En cambio, con el modelo *Free* será RPC quien administrará la caché de hilos, proporcionando una solución intermedia entre los extremos mencionados.



Inicie una nueva aplicación, y guarde el proyecto con el nombre de *ServForo*; da lo mismo el nombre que le de a la ventana principal, pero asigne *wsMinimized* en su propiedad *WindowState*. Ejecute el comando de menú *File|New*, seleccione la segunda página del Almacén de Objetos y haga doble clic sobre el icono *Automation Object*. El diálogo que aparecerá a continuación es el asistente para la creación de objetos de automatización:



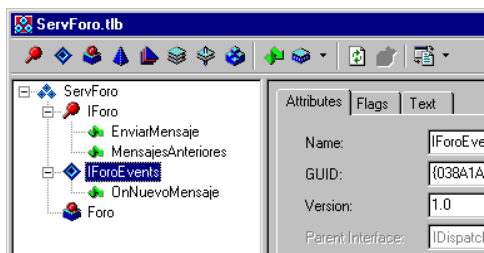
Hay un par de diferencias respecto al asistente más general que ya conocemos. Primero: no permite elegir la interfaz que vamos a implementar. ¿Para qué, si ya sabemos que es *IDispatch*? Tampoco hay una *Description* para la clase, pero se trata de un detalle sin importancia, que podemos corregir posteriormente con el Editor de Bibliotecas de Tipos. Dicho sea de paso, no es necesario pedir la inclusión de una biblioteca de tipos, porque es obligatorio, ni existe necesidad de marcar la nueva interfaz explícitamente con el indicador **oleautomation**; Delphi utilizará el atributo **dual**, que asume la presencia de dicho indicador.

Pero la principal novedad es la casilla *Generate Event support code*, que debemos dejar marcada. ¿Recuerda que nuestro servidor debe avisar a su cliente cuando alguna de sus instancias reciba un mensaje? Lograremos este efecto mediante eventos COM, que explicaremos más adelante.

### ADVERTENCIA

No debe confundir los eventos COM, o *eventos a secas*, con los eventos COM+, que son una de las características exclusivas de COM+ sobre Win2K y WinXP.

Cuando cerramos el asistente, Delphi genera una biblioteca de tipos con tres entidades definidas en su interior, en vez de las dos que son habituales:



Esta vez tenemos un nodo para *Foro*, que es la clase COM, otro para *IForo*, que es la interfaz que va a implementar la clase *Foro*, y esa especie de tartaleta o pastelillo de cianuro que lleva el título de *IForoEvents*, en perfecto *spanglish*. *IForoEvents* es una interfaz de envío, e indica qué eventos va a disparar la clase COM. Es muy importante que comprenda que nuestro servidor *no debe implementar* la interfaz de envío. Esto queda explícitamente señalado en la declaración IDL de la clase *Foro*:

```
[ uuid(038A1A65-DA32-11D5-8943-00C026263861),
  version(1.0), helpstring("Foro Object") ]
coclass Foro {
    [default] interface IForo;
    [default, source] dispinterface IForoEvents;
};
```

Observe que *IForoEvents* aparece mencionada, pero con el atributo **source**. Aquellos clientes que quieran recibir nuestros eventos serán los encargados de suministrar una implementación; en el lado cliente, por supuesto.

Como puede observar en la imagen, hay que añadir dos métodos a la interfaz *IForo*, y un método a la interfaz de envío. La declaración de la primera debe terminar siendo:

```
[ uuid(038A1A61-DA32-11D5-8943-00C026263861),
  version(1.0), helpstring("Dispatch interface for Foro Object"),
  dual, oleautomation ]
interface IForo: IDispatch {
    [ id(0x00000001) ]
    HRESULT _stdcall EnviarMensaje(
        [in] BSTR Mensaje,
        [in] long Raiz,
        [out, retval] long * Rslt );
    [ id(0x00000002) ]
    HRESULT _stdcall MensajesAnteriores(
        [out, retval] VARIANT * Rslt );
};
```

La traducción de esta interfaz a Delphi es la siguiente:

```
type
    IForo = interface(IDispatch)
        ['{038A1A61-DA32-11D5-8943-00C026263861}']
        function EnviarMensaje(const Mensaje: WideString;
            Raiz: Integer): Integer; safecall;
        function MensajesAnteriores: OleVariant; safecall;
    end;
```

Es decir, podemos enviar mensajes al servidor dentro de una cadena de caracteres, con el método *EnviarMensajes*, y recibir un valor numérico, que corresponderá al identificador que el servidor le asignará al mensaje. He incluido el parámetro *Raiz* para indicar que el mensaje se envía en respuesta a otro mensaje; si no fuese el caso, *Raiz* debería valer  $-1$ . Por otra parte, el método *MensajesAnteriores* es un extra que sirve para que un cliente que acaba de incorporarse a la discusión solicite al servidor todos los mensajes anteriores. El método devuelve un *OleVariant* con todos los mensajes. Más adelante explicaré el formato en que se “transmite” la lista de mensajes.

Esta es la declaración IDL de la interfaz *IForoEvents*:

```
[ uuid(038A1A63-DA32-11D5-8943-00C026263861),
  version(1.0), helpstring("Events interface for Foro Object") ]
dispinterface IForoEvents {
  properties:
  methods:
  [ id(0x00000001) ]
  HRESULT OnNuevoMensaje(
    [in] BSTR Mensaje, [in] long ID, [in] long Raiz );
};
```

La traducción en Delphi sería:

```
type
  IForoEvents = dispinterface
    ['{038A1A63-DA32-11D5-8943-00C026263861}']
    procedure OnNuevoMensaje(const Mensaje: WideString;
      ID: Integer; Raiz: Integer); dispid 1;
  end;
```

Cuando el servidor reciba un nuevo mensaje, debe enviar una notificación a su cliente incluyendo el texto del mensaje, el identificador asignado y el identificador del mensaje al que responde.

## La lista de mensajes

Cuando expliqué la arquitectura del servidor, mencioné que los mensajes se almacenarían en una estructura de datos global. La unidad *Mensajes*, incluida con el ejemplo en el CD, contiene las declaraciones para esa clase. El siguiente listado corresponde a la interfaz pública de la unidad, después de haber purgado las secciones que tenían que ver con los detalles de implementación:

```
type
  TMsg = class(TCollectionItem)
  published
    property MessageID: Integer;
    property ParentID: Integer;
    property Text: string;
  end;

  TMessages = class(TComponent)
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    function Add(AParentID: Integer;
      const AText: string): Integer;
    function Find(AMessageID: Integer): TMsg;
    property MessageCount: Integer;
    property Messages[I: Integer]: TMsg; default;
  end;
```

La clase *TMsg* representa un mensaje individual, y el componente *TMessages* almacena una colección de objetos *TMsg* dentro de una propiedad interna de tipo *TCollection*. A

primera vista, parece un esfuerzo vano. No lo es, por supuesto. También en la interfaz de *Mensajes* se declaran dos funciones globales:

```
function Msgs2Var(M: TMessages): Variant;
function Var2Msgs(V: Variant; Instance: TMessages = nil): TMessages;
```

La idea consiste en aprovechar las mismas rutinas que utiliza Delphi para guardar un componente en un fichero *dfm* para “aplanar” la lista de mensajes y forzarla dentro de una variable de tipo *Variant*. No voy a detenerme en los detalles, pero aquí tiene la implementación de *Msgs2Var*, como ejemplo:

```
function Msgs2Var(M: TMessages): Variant;
var
    BinStream: TMemoryStream;
    StrStream: TStringStream;
begin
    if M = nil then
        Result := Null
    else
        begin
            BinStream := TMemoryStream.Create;
            try
                BinStream.WriteComponent(M);
                StrStream := TStringStream.Create('');
                try
                    BinStream.Seek(0, soFromBeginning);
                    ObjectBinaryToText(BinStream, StrStream);
                    Result := StrStream.DataString;
                finally
                    FreeAndNil(StrStream);
                end;
            finally
                FreeAndNil(BinStream);
            end;
        end;
    end;
end;
```

¿Recuerda que la clase *Foro* debía implementar un método llamado *MensajesAnteriores*? Lo ideal habría sido que COM nos dejase “transmitir” un objeto de tipo *TMessages* desde el servidor al cliente, pero lamentablemente no es posible. Es cierto que podemos pasar como parámetro de un método COM un puntero a una interfaz. Supongamos, por ejemplo, que hemos definido una interfaz *IMessages* para describir la funcionalidad de una lista de mensajes. Podríamos declarar entonces la interfaz del foro de la siguiente manera:

```
type                                // !!! SOLAMENTE ES UNA HIPOTESIS!!!
IForo = interface(IDispatch)
    ['{038A1A61-DA32-11D5-8943-00C026263861}']
    function EnviarMensaje(const Mensaje: WideString;
        Raiz: Integer): Integer; safecall;
    function MensajesAnteriores: IMessages; safecall;
end;
```

Cuando el cliente llamase a *MensajesAnteriores* recibiría como respuesta un puntero a la interfaz *IMessages* de un objeto... ubicado en el servidor. En dos palabras: COM

transmitiría solamente el puntero, pero el objeto se quedaría en el servidor. El cliente podría, de todos modos, llamar a los métodos de *IMessages* para recorrer la lista. Sin embargo, no sería una técnica recomendable cuando el servidor y el cliente se encontrasen en distintos ordenadores, porque para recibir toda la información de la lista sería necesario efectuar múltiples invocaciones de métodos remotos.

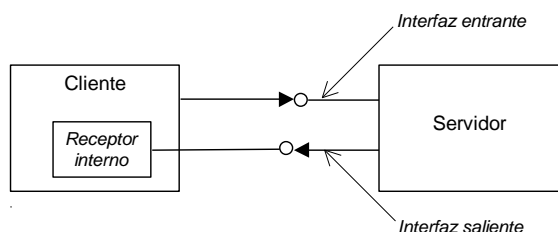
De ahí que haya preferido aplanar la lista dentro de un variante y enviársela así al cliente. Una vez que haya llegado a su destino, se ejecutaría *Var2Msgs*, la función simétrica a *Msgs2Var*, para restablecer la lista de mensajes en todo su esplendor.

#### NOTA

En CORBA sí se permite pasar objetos “por valor” a través de los parámetros de métodos remotos.

## Eventos COM

Casi al principio de este libro, en el capítulo 2, vimos que los punteros de interfaces podían utilizarse como mecanismo de implementación de eventos. Y precisamente así es como se manejan los eventos en COM. Para que un servidor pueda enviar métodos a un cliente, debe llamar a métodos de una interfaz implementada por el cliente, como se muestra en el siguiente diagrama.



El servidor es el encargado de definir la interfaz de envío de eventos, conocida con el nombre de *interfaz saliente (outgoing interface)*, y puede tener todos los métodos que se le ocurra al programador. En el ejemplo que estamos desarrollando, la interfaz saliente es una interfaz de envío (*dispatch interface*), pero no es una condición necesaria, por regla general.

Enumeremos las condiciones necesarias para que un servidor envíe notificaciones a un cliente:

- 1 El servidor debe haber definido una interfaz, con un método por cada evento que detecte el servidor.
- 2 El cliente interesado en recibir las notificaciones debe implementar internamente esa interfaz.



- 3 Muy importante: el cliente debe avisar al objeto servidor su interés en recibir las notificaciones. Dicho sin rodeos, el cliente debe pasar al servidor un puntero a la interfaz del objeto que va a recibir las notificaciones.

Hay mil formas de cumplir cada uno de los puntos anteriores... pero COM define un mecanismo estándar para el envío y tratamiento de eventos. No es un mecanismo perfecto, como veremos; por una parte, está basado en *IDispatch*, y además exige demasiados pasos para establecer la comunicación entre la fuente de eventos y el receptor. No obstante, es el sistema que utiliza Delphi cuando pedimos al asistente de automatización OLE que genere soporte para eventos, y no nos queda otra alternativa que presentarlo.

El sistema se apoya en varias interfaces. De ellas, la primera con la que tropezaremos será *IConnectionPointContainer*, y debe ser implementada por la clase que actúa como fuente de los eventos. Su declaración en Delphi es la siguiente:

```
type
    // Unidad ActiveX
    IConnectionPointContainer = interface
        ['{B196B284-BAB4-101A-B69C-00AA00341D07}']
        function EnumConnectionPoints(
            out Enum: IEnumConnectionPoints): HRESULT; stdcall;
        function FindConnectionPoint(const iid: TIID;
            out cp: IConnectionPoint): HRESULT; stdcall;
    end;

    IEnumConnectionPoints = interface
        ['{B196B285-BAB4-101A-B69C-00AA00341D07}']
        function Next(celt: Longint; out elt;
            pceltFetched: PLongint): HRESULT; stdcall;
        function Skip(celt: Longint): HRESULT; stdcall;
        function Reset: HRESULT; stdcall;
        function Clone(
            out Enum: IEnumConnectionPoints): HRESULT; stdcall;
    end;
```

Tanta “complicación” tiene como meta permitir que una misma clase soporte varias interfaces de eventos diferentes. Una agencia de noticias, por ejemplo, puede implementar por separado distintos servicios de notificaciones, para que el cliente pueda suscribirse al envío de noticias deportivas, sobre la Bolsa, o sobre lo que desee. Cada uno de estos servicios independientes vendría representado por una interfaz de eventos diferentes y, en lo que respecta al propio mecanismo de subscripción, por una interfaz llamada *IConnectionPoint*. El método *FindConnectionPoint*, de la interfaz *IConnectionPointContainer*, sirve para que el cliente localice el punto de conexión adecuado para el “servicio” (el identificador único de la interfaz de eventos) al que desea suscribirse. El otro método del contenedor sirve para recorrer todos los puntos de conexión que implementa el objeto.

La declaración de *IConnectionPoint* es la siguiente:

```
type
    IConnectionPoint = interface
```

```

['{B196B286-BAB4-101A-B69C-00AA00341D07}']
function GetConnectionInterface(
    out iid: TIID): HRESULT; stdcall;
function GetConnectionPointContainer(
    out cpc: IConnectionPointContainer): HRESULT; stdcall;
function Advise(const unkSink: IUnknown;
    out dwCookie: Longint): HRESULT; stdcall;
function Unadvise(dwCookie: Longint): HRESULT; stdcall;
function EnumConnections(
    out Enum: IEnumConnections): HRESULT; stdcall;
end;

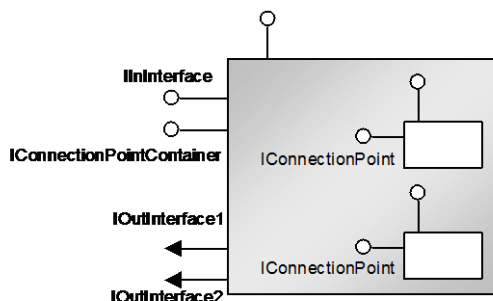
```

Los dos métodos principales de *IConnectionPoint* son *Advise* y *Unadvise*. Cuando el cliente recibe el puntero a un punto de conexión, debe llamar a *Advise* para enviarle al emisor el puntero de interfaz que implementa la recepción de eventos. A cambio, *Advise* devuelve al cliente una especie de comprobante: el parámetro *dwCookie*. En el momento en que el cliente se harte de la suscripción debe enviar ese comprobante a *Unadvise* para que el servidor deje de enviar sus notificaciones.

### MUY IMPORTANTE

Si lo piensa un poco, comprenderá que *Advise* y *Unadvise* no ponen restricciones para que más de un cliente se suscriba en un punto de conexión. El objeto servidor es el que decide si quiere enviar notificaciones a un solo cliente o a toda una lista (*multicasting*). El sistema de eventos de la VCL, en contraste, permite conectar sólo un receptor por cada emisor.

El diagrama de la siguiente página muestra las relaciones que deben existir entre las interfaces mencionadas y la clase COM que publica eventos. En él, la clase COM implementa *IConnectionPointContainer*, además de su propia interfaz *IInInterface*. Las dos flechas de la esquina inferior izquierda corresponden a dos interfaces de salida que supondremos que soporta la clase. Para cada interfaz de salida, la clase gestiona internamente un objeto que implementa la interfaz predefinida *IConnectionPoint*, que utilizará el cliente para suscribirse a una de las “flechas”. Pero el cliente no puede acceder directamente a las interfaces *IConnectionPoint*, sino que tiene que solicitarlas al objeto a través de su implementación de *IConnectionPointContainer*.



## Cómo Delphi soporta los eventos

La unidad *AxCtrls* contiene el soporte de Delphi para eventos. Allí también se declaran las clases y rutinas para el trabajo con controles ActiveX y con los formularios activos, que estudiaremos en la parte de Internet. En particular, Delphi declara en *AxCtrls* las clases *TConnectionPoints* y *TConnectionPoint* que implementan las interfaces *IConnectionPointContainer* y *IConnectionPoint*.

Como es necesario que sea la propia clase COM la que implemente el contenedor de puntos de conexión, *TConnectionPoints* se utiliza para declarar una propiedad en la clase *TForo* que a su vez implementa *IConnectionPointContainer* en beneficio de la clase por delegación. Excepto el destructor, que he añadido posteriormente, el siguiente listado muestra cómo Delphi declara la clase *TForo* con todos los accesorios necesarios para el trabajo con eventos:

```

type
  TForo = class(TAutoObject, IConnectionPointContainer, IForo)
  private
    FConnectionPoints: TConnectionPoints;
    FConnectionPoint: TConnectionPoint;
    FEvents: IForoEvents;
  public
    procedure Initialize; override;
    destructor Destroy; override;
  protected
    property ConnectionPoints: TConnectionPoints
      read FConnectionPoints
      implements IConnectionPointContainer;
    procedure EventSinkChanged(
      const EventSink: IUnknown); override;
    function EnviarMensaje(const Mensaje: WideString;
      Raiz: Integer): Integer; safecall;
    function MensajesAnteriores: OleVariant; safecall;
  end;

```

Observe que el asistente de Delphi solamente soporta una interfaz de eventos por clase, porque genera un solo punto de conexión, al cual nos da acceso a través de la variable interna *FConnectionPoint*. La inicialización tiene lugar del siguiente modo:

```

procedure TForo.Initialize;
begin
  inherited Initialize;
  FConnectionPoints := TConnectionPoints.Create(Self);
  if AutoFactory.EventTypeInfo <> nil then
    FConnectionPoint := FConnectionPoints.CreateConnectionPoint(
      AutoFactory.EventIID, ckSingle, EventConnect)
  else
    FConnectionPoint := nil;
end;

```

Aquí vemos entrar en escena otra limitación: en el segundo parámetro de la llamada a *CreateConnectionPoint* se pasa la constante *ckSingle*. Esto indica que el punto de conexión almacenará el puntero al receptor de eventos de un único cliente. Esto no es tan

malo como podría parecer, porque en definitiva cada cliente va a estar asociado a un único servidor, y viceversa. De todos modos, se puede cambiar esa constante por *ckMulti* para permitir más de un cliente por objeto.

Cada vez que un cliente se conecta o desconecta del punto de conexión, se dispara el siguiente método de *TForo*:

```
procedure TForo.EventSinkChanged(const EventSink: IUnknown);  
begin  
    FEvents := EventSink as IForoEvents;  
end;
```

Esto significa que podemos utilizar la variable interna *FEvents* de *TForo* para ejecutar, desde el servidor, métodos remotos implementados en el cliente. Dicho en términos más prosaicos, para disparar un evento debemos ejecutar el método correspondiente de *FEvents*; después de comprobar que esta variable no contenga un puntero vacío, claro está.

### IMPORTANTE

¿Por qué Delphi crea la interfaz de eventos como una **disinterface**? ¿Es acaso una condición impuesta por COM? No, es una limitación importante, pero dictada por el sentido común: en caso contrario, los lenguajes interpretados como VBScript no podrían tratar esos eventos.

## Sincronización, listas de objetos y otras tonterías

Ya estamos en condiciones de explicar la implementación completa de la clase *TForo*. En primer lugar, he declarado tres variables globales dentro de la unidad, para que todos los objetos de la clase *TForo* tengan acceso a ellas:

```
var  
    ListaInstancias: TList = nil;  
    ListaMensajes: TMessages = nil;  
    CritSect: TCriticalSection = nil;
```

En *ListaInstancias* almacenaremos los punteros a todos los objetos *TForo* que se construyan. *ListaMensajes* es la famosa estructura de datos global que almacenará los mensajes que nos envíen los clientes. Naturalmente, para que los objetos COM, que residirán en un apartamento multihilos, no tengan problemas de sincronización, he incluido además la variable *CritSect* para el control de concurrencia. La clase *TCriticalSection* se declara en la unidad *SyncObjs*.

Las variables globales se crean y destruyen en las secciones de inicialización y finalización de la unidad:

```
initialization  
    ListaInstancias := TList.Create;  
    ListaMensajes := TMessages.Create(nil);  
    CritSect := TCriticalSection.Create;  
    TAutoObjectFactory.Create(ComServer, TForo, Class_Foro,
```

```

        ciMultiInstance,
        tmFree);          // ;EL MODELO FREE DE CONCURRENCIA!
finalization
    ListaInstancias.Free;
    ListaMensajes.Free;
    CritSect.Free;
end.

```

Primero modificaremos la implementación de *Initialize*:

```

procedure TForo.Initialize;
begin
    inherited Initialize;
    CritSect.Enter;          // Desde aquí...
    try
        ListaInstancias.Add(Self);
    finally
        CritSect.Leave;
    end;                    // ...hasta aquí
    FConnectionPoints := TConnectionPoints.Create(Self);
    if AutoFactory.EventTypeInfo <> nil then
        FConnectionPoint := FConnectionPoints.CreateConnectionPoint(
            AutoFactory.EventIID, ckSingle, EventConnect)
    else
        FConnectionPoint := nil;
    end;

```

La única adición consiste en añadir un puntero al objeto, *Self*, en la lista de instancias global. ¡Cuidado!, no se guarda un puntero a interfaz, que sería dependiente del apartamento donde se genere, sino un puntero *real* al objeto. No hay problema alguno con este proceder, porque el puntero será utilizado dentro del mismo proceso.

También he implementado un destructor, para eliminar el puntero del objeto que se destruye de la lista:

```

destructor TForo.Destroy;
begin
    CritSect.Enter;
    try
        ListaInstancias.Delete(ListaInstancias.IndexOf(Self));
    finally
        CritSect.Leave;
    end;
    inherited Destroy;
end;

```

En ambos casos he protegido el núcleo de la operación mediante la sección crítica. Recuerde que si dos clientes piden simultáneamente una instancia al servidor, el servicio de RPC creará dos hilos, o los reciclará a partir de la caché de hilos que mantiene automáticamente. La misma precaución se mantiene al implementar el método *MensajesAnteriores*:

```

function TForo.MensajesAnteriores: OleVariant;
begin
    CritSect.Enter;
    try

```

```

        Result := Msgs2Var(ListaMensajes);
    finally
        CritSect.Leave;
    end;
end;

```

### ADVERTENCIA

Con demasiada frecuencia se leen predicciones apocalípticas sobre el manejo de objetos de sincronización dentro de un MTA, y se menciona una frase misteriosa: "...estos objetos tienen *thread affinity*...". La afinidad significa que las operaciones de pedir y liberar las secciones críticas, semáforos y demás deben efectuarse dentro del mismo hilo. Pero un objeto que vive en una MTA puede que ejecute ahora un método A en el hilo B, y más adelante en un hilo C completamete diferente. La solución: libere el objeto de sincronización que vaya a utilizar dentro del mismo método en que lo pide.

La implementación más interesante es la de *EnviarMensaje*:

```

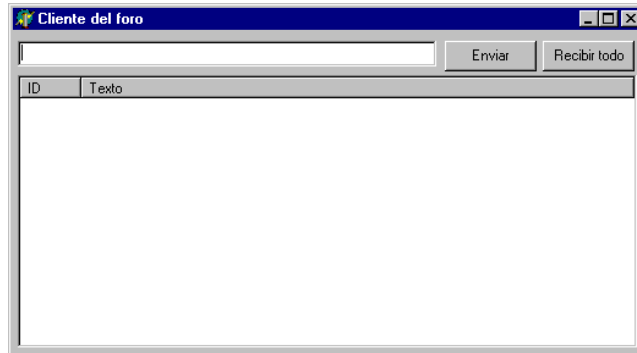
function TForo.EnviarMensaje(const Mensaje: WideString;
    Raiz: Integer): Integer;
var
    I: Integer;
begin
    CritSect.Enter;
    try
        Result := ListaMensajes.Add(Raiz, Mensaje);
        for I := 0 to ListaInstancias.Count - 1 do
            with TForo(ListaInstancias[I]) do
                if FEvents <> nil then
                    FEvents.OnNuevoMensaje(Mensaje, Result, Raiz);
            end;
        finally
            CritSect.Leave;
        end;
    end;
end;

```

Lo primero que hacemos es guardar el nuevo mensaje en la lista global de mensajes. A continuación debemos notificar a todos los clientes que estén conectados al servidor. Por este motivo, recorremos la lista de instancias de *TForo*, y enviamos la notificación a través del atributo *FEvents* de la clase, en el caso en que no contenga un puntero vacío.

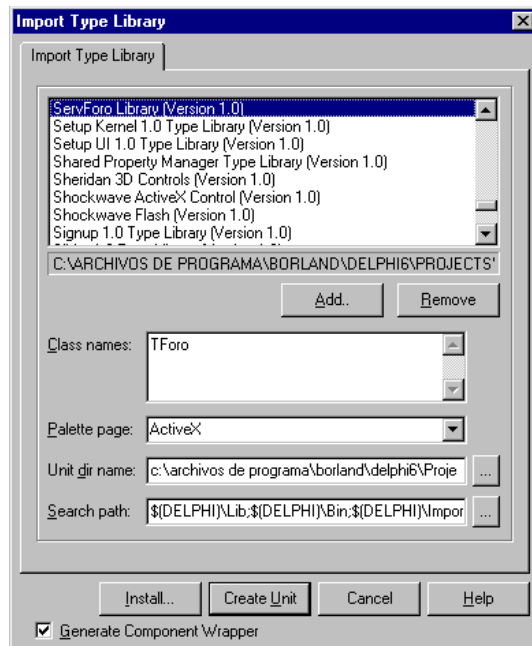
## Importación de bibliotecas de tipos

Necesitamos ahora un cliente para poner a prueba el servidor, así que iniciamos una nueva aplicación. En su ventana principal dejamos caer un cuadro de edición, al que llamaremos *edMessage*, un par de botones para enviar al servidor el texto que tecleemos en *edMessage* y para recibir todos los mensajes almacenados en el servidor, y finalmente, un componente *TListView*, que bautizaremos *ListView*.



Cambiamos el valor de la propiedad *ViewStyle* del control de listas a *vmReport*, y configuramos dos columnas, tal como se muestra en la imagen anterior. Para simplificar, no voy a organizar los mensajes en un árbol. Dejaré esa tarea a su cargo.

Como vamos a necesitar la clase *TMessages* para recibir la lista de mensajes entera, añadimos a este proyecto la unidad *Mensajes*, que desarrollamos para el proyecto del servidor. Y ahora sería el momento de añadir también la unidad *ServForo\_TLB*, con las declaraciones de la biblioteca de tipos del servidor, pero...



... ¡las manos en alto, no lo haga! Normalmente, reutilizaríamos esa unidad, pero hay un pequeño detalle en nuestro servidor que nos aconseja proceder de otro modo: tenemos entre manos un servidor que dispara eventos. Resulta que, utilizando el asistente adecuado, Delphi puede echarnos una mano para crear un cliente que reac-

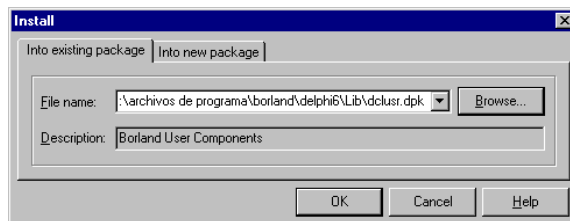
cione frente a esos eventos. Ejecute el comando de menú *Project | Import Type Library*, para que aparezca el diálogo de la página anterior.

Esta es la herramienta que permite a Delphi traducir una biblioteca de tipos en una unidad con declaraciones en Object Pascal. En los casos en que tuviésemos un servidor COM y no nos hubieran dado una unidad con declaraciones para Delphi, ejecutaríamos este asistente para crear esa unidad y poder acceder a las clases implementadas dentro del servidor desconocido. Como se aprecia en la imagen anterior, al programador se le presenta una lista de las bibliotecas que Delphi encuentra en el Registro de Windows. La búsqueda se efectúa dentro de la clave:

```
HKEY_CLASSES_ROOT\TypeLib
```

Si no se hubiera registrado ni el servidor ni la biblioteca de tipos, pero supiéramos que esta última se encontrase dentro de determinado fichero, pulsaríamos el botón *Add* para añadir la biblioteca a la lista; sin registrarla, claro.

Cada vez que seleccionamos una biblioteca, en la parte inferior del diálogo se muestran las clases COM que contiene, y se nos ofrece la oportunidad de crear una unidad en Delphi con las declaraciones equivalentes. Hay dos botones preparados para ello: con *Create Unit* se crea solamente la unidad con las declaraciones, pero si pulsamos *Install*, la unidad se añade al paquete que seleccionemos y, si se han creado componentes como resultado de la importación, estos aparecerán en la Paleta de Componentes de Delphi:



## Componentes para servidores OLE

¿De qué componentes estoy hablando? Si la casilla *Generate Component Wrapper* no está marcada, la unidad que se crea solamente tendrá las declaraciones de la interfaz, y una pequeña clase auxiliar, derivada directamente de *TObject* y que simplifica la creación de instancias. En nuestro ejemplo, se crearía la siguiente clase:

```
type
  CoForo = class
    class function Create: IForo;
    class function CreateRemote(const MachineName: string): IForo;
  end;
```

Pero si marcamos la casilla en cuestión, se crea un componente adicional por cada clase COM presente en la biblioteca. Para este ejemplo, se crearía una clase *TForo*,



utilizando como clase base *TOleServer*, una clase definida en la unidad *OleServer* a partir de *TComponent*. Si se tratase de un control ActiveX, la clase base sería *TOleControl*.

Esta es la declaración de *TForo*, después de quitar algunas declaraciones superfluas:

```

type
  TForo = class(TOleServer)
  private
    FOnNuevoMensaje: TForoOnNuevoMensaje;
    FIntf: IForo;
    function GetDefaultInterface: IForo;
  protected
    procedure InitServerData; override;
    procedure InvokeEvent(DispID: TDispID;
      var Params: TVariantArray); override;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    procedure Connect; override;
    procedure ConnectTo(svrIntf: IForo);
    procedure Disconnect; override;
    function EnviarMensaje(const Mensaje: WideString;
      Raiz: Integer): Integer;
    function MensajesAnteriores: OleVariant;
    property DefaultInterface: IForo read GetDefaultInterface;
  published
    property OnNuevoMensaje: TForoOnNuevoMensaje
      read FOnNuevoMensaje write FOnNuevoMensaje;
end;

```

Y la interfaz pública de *TOleServer* es la que muestro a continuación:

```

type
  TConnectKind = (ckRunningOrNew, ckNewInstance,
    ckRunningInstance, ckRemote, ckAttachToInterface);

  TOleServer = class(TComponent, IUnknown)
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    procedure Connect; virtual; abstract;
    procedure Disconnect; virtual; abstract;
  published
    property AutoConnect: Boolean;
    property ConnectKind: TConnectKind;
    property RemoteMachineName: string;
end;

```

Comprobamos entonces que:

- *TForo* soporta los mismos métodos que la interfaz *IForo*.
- Sin embargo, *TForo* no “implementa” la interfaz *IForo*.
- Delphi ha traducido los eventos COM de la clase en eventos comunes y corrientes de la VCL.

¿Merece la pena crear un componente para trabajar con un servidor, en vez de utilizar directamente la interfaz, como en ejemplos anteriores? Mi recomendación es que utilice componentes solamente cuando el servidor dispare eventos. Si lo único que vamos a hacer es ejecutar métodos remotos, no vale la pena añadir la complejidad de un componente adicional.

¿Y si la interfaz exporta propiedades? La interfaz *IForo* no contiene propiedades, pero incluso en este ejemplo, Delphi define una clase auxiliar:

```
type
  TForoProperties = class(TPersistent)
  private
    FServer: TForo;
    function GetDefaultInterface: IForo;
    constructor Create(AServer: TForo);
  protected
  public
    property DefaultInterface: IForo read GetDefaultInterface;
  published
  end;
```

*TForoProperties* agruparía las propiedades de la interfaz por omisión de la clase, de modo que pudieran ser manejadas desde la propiedad *Server* del componente:

```
type
  TForo = class(TOleServer)
  private
    {$IFDEF LIVE_SERVER_AT_DESIGN_TIME}
    FProps: TForoProperties;
    function GetServerProperties: TForoProperties;
    {$ENDIF}
  published
    {$IFDEF LIVE_SERVER_AT_DESIGN_TIME}
    property Server: TForoProperties read GetServerProperties;
    {$ENDIF}
    // ...
  end;
```

Note, sin embargo, que hay una directiva de compilación condicional, para que las propiedades aparezcan sólo si lo pedimos explícitamente al compilador. ¿Cuál es el problema? El código para las propiedades que genera actualmente Delphi es incompleto y contiene errores. Por ejemplo, no se maneja correctamente la asignación de valores en las propiedades. Incluso si corregimos este error, encontraremos que al establecer una conexión se pierde el valor que hayamos asignado dentro de las propiedades en tiempo de diseño.

Veamos ahora cómo el componente maneja los eventos que recibe. En primer lugar, Delphi ha declarado un tipo de evento:

```
type
  TForoOnNuevoMensaje = procedure (Sender: TObject;
    const Mensaje: string; ID, Raiz: Integer) of object;
```

Aquí he hecho trampas, porque el asistente generó una interfaz incorrecta para el tipo del evento. Es un *bug* muy fastidioso, porque perdí casi tres horas buscando qué era lo que estaba fallando. Y es que no sólo está el error con los tipos de parámetros, sino que para colmo, el código que dispara el evento pasaba los parámetros en orden inverso. El disparo del evento tiene lugar dentro del método *InvokeEvent*, que es llamado a su vez por el método *Invoke* de un objeto interno que implementa la interfaz *IDispatch* que recibe las notificaciones desde el servidor:

```
procedure TForo.InvokeEvent(DispID: TDispID; var Params: TVariantArray);
begin
    case DispID of
        -1: Exit; // DISPID_UNKNOWN
        1: if Assigned(FOnNuevoMensaje) then
            FOnNuevoMensaje(Self, Params[0], Params[1], Params[2]);
            // Aquí he tenido que invertir los parámetros
    end;
end;
```

## Implementando el cliente

No es necesario que registremos el componente *TForo* dentro del Entorno de Desarrollo. Para el cliente del foro voy a crear el componente en tiempo de ejecución. Modificamos la sección **private** de la clase de la ventana principal añadiendo estas dos declaraciones:

```
private
    Foro: TForo;
    procedure NuevoMensaje(Sender: TObject;
        const Mensaje: string; ID: Integer; Raiz: Integer);
```

El componente *TForo* se instancia y conecta en la respuesta al evento *OnCreate* del formulario, y allí también se le asocia la respuesta al evento *OnNuevoMensaje*:

```
procedure TwndMain.FormCreate(Sender: TObject);
begin
    Foro := TForo.Create(Self);
    Foro.Connect;
    Foro.OnNuevoMensaje := NuevoMensaje;
end;
```

Cada vez que alguien envía un mensaje al servidor, éste nos lo hace saber activando remotamente nuestro método *NuevoMensaje*. Como respuesta, debemos añadir el texto del mensaje al control *ListView*. Recuerde que no estamos manteniendo las relaciones de pregunta/respuesta entre los mensajes:

```
procedure TwndMain.NuevoMensaje(Sender: TObject;
    const Mensaje: string; ID, Raiz: Integer);
begin
    with ListView.Items do
        begin
            Caption := FormatFloat('0,', ID);
```

```

        SubItems.Add(Mensaje);
    end;
end;

```

Para enviar un mensaje aprovechamos el puntero de interfaz *IForo* que hemos recibido al conectarnos al servidor, interceptando el evento *OnClick* del primero de los botones:

```

procedure TwndMain.bnEnviarClick(Sender: TObject);
begin
    Foro.EnviaMensaje(edMessage.Text, -1);
end;

```

El segundo botón llama al método remoto *MensajesAnteriores*. El valor variante recibido como respuesta se utilizará para restaurar una lista de mensajes completa, que se recorre para añadir todos sus elementos en el control de lista:

```

procedure TwndMain.bnReceiveClick(Sender: TObject);
var
    I: Integer;
    M: TMessages;
begin
    M := Var2Msgs(Foro.MensajesAnteriores);
    try
        ListView.Items.BeginUpdate;
        try
            ListView.Clear;
            for I := 0 to M.MessageCount - 1 do
                with M[I] do
                    NuevoMensaje(nil, Text, MessageID, ParentID);
                finally
                    ListView.Items.EndUpdate;
                end;
            finally
                M.Free;
            end;
        end;
    end;

```

Si se ha quedado con ganas de seguir programando, le sugiero que organice los mensajes en forma de árbol, como preguntas y respuestas. Este cambio puede hacerse modificando solamente el cliente, porque los métodos del servidor ya contemplan esta posibilidad.

# 2

## **El Lenguaje SQL**

---

- **Sistemas de bases de datos**
- **InterBase**
- **Consultas y modificaciones**
- **Procedimientos almacenados y triggers**
- **Transacciones**
- **Microsoft SQL Server**
- **Oracle**
- **DB2 Universal Database**

# Parte



## Sistemas de bases de datos

**E**S EL MOMENTO APROPIADO PARA presentar a los protagonistas de este drama: los sistemas de gestión de bases de datos con los que intentaremos trabajar. En mi trabajo de consultor, la primera pregunta que escucho, y la más frecuente, es en qué lenguaje debe realizarse determinado proyecto. Enfoque equivocado. La mayoría de los proyectos que llegan a mis manos son aplicaciones para redes de área local, y os garantizo que la elección del lenguaje es asunto relativamente secundario para el éxito de las mismas; por supuesto, siempre recomendando algún lenguaje “decente”, sin limitaciones intrínsecas, como C++ Builder o Delphi. La primera pregunta debería ser: ¿qué sistema de bases de datos es el más apropiado a mis necesidades?

En este capítulo recordaremos los principios básicos de los sistemas de bases de datos relacionales, y haremos una rápida introducción a algunos sistemas concretos con los que Delphi puede trabajar de modo directo. La explicación relativa a los sistemas de bases de datos locales será más detallada que la de los sistemas cliente/servidor, pues las características de estos últimos (implementación de índices, integridad referencial, seguridad) irán siendo desveladas a lo largo del libro.

### **Acerca del acceso transparente a bases de datos**

¿Pero acaso Delphi no ofrece cierta transparencia con respecto al formato de los datos con los que estamos trabajando? Pues sí, sobre todo para los formatos soportados por el Motor de Bases de Datos de Borland (BDE), que estudiaremos en el capítulo correspondiente. Sin embargo, esta transparencia no es total, incluso dentro de las bases de datos accesibles mediante el BDE. Hay una primera gran división entre las bases de datos locales y los denominados sistemas SQL. Luego vienen las distintas posibilidades expresivas de los formatos; incluso las bases de datos SQL, que son las más parecidas entre sí, se diferencian en las operaciones que permiten o no. Si usted está dispuesto a utilizar el mínimo común denominador entre todas ellas, su aplicación puede que funcione sin incidentes sobre determinado rango de posibles sistemas ... pero le aseguro que del mismo modo desperdiciará recursos de programación y diseño que le habrían permitido terminar mucho antes la aplicación, y que ésta se ejecutara más eficientemente.

Por supuesto, estimado amigo, todos los sistemas de bases de datos con los que vamos a trabajar en Delphi serán sistemas relacionales. Por desgracia. ¿Cómo que por desgracia, hereje insensato? Para saber qué nos estamos perdiendo con los sistemas relacionales tendríamos que conocer las alternativas. Necesitaremos, lo lamento, un poco de vieja Historia.

En el principio no había ordenadores, claro está. Pero cuando los hubo, y después de largos años de almacenar información “plana” en grandes cintas magnéticas o perforadas, los informáticos comenzaron a organizar sus datos en dos tipos de modelos: el modelo jerárquico y el modelo de redes. El modelo jerárquico era un invento digno de un oficial prusiano. Los diferentes tipos de información se clasificaban en forma de árbol. En determinada base de datos, por ejemplo, la raíz de este árbol eran los registros de empresas. Cada empresa almacenaría los datos de un conjunto de departamentos, estos últimos serían responsables de guardar los datos de sus empleados, y así sucesivamente. Además del conjunto de departamentos, una empresa podría ser propietaria de otro conjunto de registros, como bienes inmuebles o algo así. El problema, como es fácil de imaginar, es que el mundo real no se adapta fácilmente a este tipo de organización. Por lo tanto, a este modelo de datos se le añaden chapuzas tales como “registros virtuales” que son, en el fondo, una forma primitiva de punteros entre registros.

El modelo de redes era más flexible. Un registro podía contener un conjunto de otros registros. Y cada uno de estos registros podía pertenecer a más de un conjunto. La implementación más frecuente de esta característica se realizaba mediante punteros. El sistema más famoso que seguía este modelo, curiosamente, fue comprado por cierta compañía tristemente conocida por hacerse con sistemas de bases de datos para convertirlos en historia... No, no es esa Compañía en la que estáis pensando... sí, esa Otra...

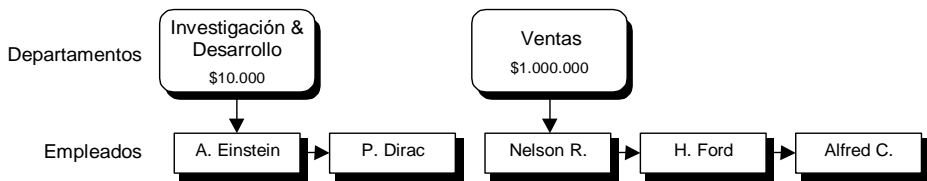
Vale, el modelo jerárquico no era muy completo, pero el modelo de redes era razonablemente bueno. ¿Qué pasó con ellos, entonces? ¿Por qué se extinguieron en el Jurásico? Básicamente, porque eran sistemas *navegacionales*. Para obtener cualquier información había que tener una idea muy clara de cómo estaban organizados los datos. Pero lo más molesto era que no existían herramientas sencillas que permitieran realizar consultas arbitrarias en una base de datos. Si el presidente de la compañía quería saber cuántos clientes del área del Pacífico bebían Coca Cola a las cinco de la tarde en vez de té, tenía que llamar al programador para que le desarrollara una pequeña aplicación.

Entonces apareció Mr. Codd, un matemático de IBM. No inventó el concepto de registro, que ya existía hacía tiempo. Pero se dio cuenta que si obligaba a que todos los campos de los registros fueran campos simples (es decir, que no fueran punteros, vectores o subregistros) podía diseñarse un grácil sistema matemático que permitía descomponer información acerca de objetos complejos en estos registros planos, con la seguridad de poder restaurar la información original más adelante, con la ayuda de operaciones algebraicas. Lo más importante: casi cualquier tipo de información podía



descomponerse de este modo, así que el modelo era lo suficientemente general. A la teoría matemática que desarrolló se le conoce con el nombre de *álgebra relacional*, y es la base de nuestro conocido lenguaje SQL y del casi olvidado *Query By Example*, o QBE. De este modo, el directivo del párrafo anterior podía sentarse frente a una consola, teclear un par de instrucciones en SQL y ahorrarse el pago de las horas extras del programador<sup>5</sup>.

Tomemos como ejemplo una base de datos que almacene datos acerca de Departamentos y sus Empleados. Los modelos jerárquicos y de redes la representarían con un diagrama similar al siguiente:



Como puede verse, los punteros son parte ineludible del modelo. ¿Hay algún matemático que sepa cómo comportarse frente a un puntero? Al parecer, no los había en los 60 y 70. Sin embargo, los datos anteriores pueden expresarse, en el modelo relacional, mediante dos conjuntos uniformes de registros y sin utilizar punteros, al menos de forma explícita. De esta manera, es factible analizar matemáticamente los datos, y efectuar operaciones algebraicas sobre los mismos:

DEPARTAMENTOS		
Codigo	Nombre	Presupuesto
I+D	Investigación y Desarrollo	\$10.000
V	Ventas	\$1.000.000

EMPLEADOS	
Dpto	Nombre
I+D	A. Einstein
I+D	P. Dirac
V	Nelson R.
V	Henri F.
V	Alfred C.

¡Mira, mamá, sin punteros! *Departamentos* y *Empleados* son *tablas*, aunque matemáticamente se les denomina *relaciones*. A los registros de estas tablas se les llama *filas*, para hacer rabiar a los matemáticos que les llaman *tuplas*. Y para no desentonar, *Codigo*, *Nombre* y *Presupuesto* son *columnas* para unos, mientras que para los otros son *campos*. ¡Qué más da! Ah, la colección de tablas o relaciones es lo que se conoce como *base de datos*.

Las personas inteligentes (como usted y como yo) se dan cuenta enseguida de que en realidad no hemos eliminado los punteros, sino que los hemos disfrazado. Existe un

<sup>5</sup> Aunque personalmente no conozco a ningún individuo con alfiler de corbata y BMW que sepa SQL, no cabe duda de que debe existir alguno por ahí.

vínculo<sup>6</sup> entre los campos *Código* y *Dpto* que es el sustituto de los punteros. Pero cuando se representan los datos de esta manera, es más fácil operar con ellos matemáticamente. Note, por ejemplo, que para ilustrar los modelos anteriores necesité un dibujo, mientras que una vulgar tabla de mi procesador de texto me ha bastado en el segundo caso. Bueno, en realidad han sido dos tablas.

¿Me deja el lector que resuma en un par de frases lo que lograba Codd con su modelo relacional? Codd apuntaba su pistola de rayos desintegradores a cualquier objeto que se ponía a tiro, incluyendo a su perro, y lo reducía a cenizas atómicas. Después, con un elegante par de operaciones matemáticas, podía resucitar al animalito, si antes el viento no barría sus restos de la alfombra.

## Información semántica = restricciones

Todo lo que he escrito antes le puede sonar al lector como un disco rayado de tanto escucharlo. Sin embargo, gran parte de los programadores que se inician en Delphi solamente han llegado a asimilar esta parte básica del modelo relacional, y presentan lagunas aterradoras en el resto de las características del modelo, como veremos dentro de poco. ¿Qué le falta a las ideas anteriores para que sean completamente prácticas y funcionales? Esencialmente, información semántica: algo que nos impida o haga improbable colocar la cabeza del perro donde va la cola, o viceversa; el perro de una vecina mía da esa impresión.

Casi siempre, esta información semántica se expresa mediante restricciones a los valores que pueden tomar los datos de una tabla. Las restricciones más sencillas tienen que ver con el tipo de valores que puede albergar una columna. Por ejemplo, la columna *Presupuesto* solamente admite valores enteros. Pero no cualquier valor entero: tienen que ser valores positivos. A este tipo de verificaciones se les conoce como *restricciones de dominio*.

El nivel siguiente lo conforman las restricciones que pueden verificarse analizando los valores de cada fila de forma independiente. Estas no tienen un nombre especial. En el ejemplo de los departamentos y los empleados, tal como lo hemos presentado, no hay restricciones de este tipo. Pero nos podemos inventar una, que deben satisfacer los registros de empleados:

```
Dpto <> "I+D" or Especialidad <> "Psiquiatría"
```

Es decir, que no pueden trabajar psiquiatras en Investigación y Desarrollo (terminarían igual de locos). Lo más importante de todo lo que he explicado en esta sección es que las restricciones más sencillas pueden expresarse mediante elegantes fórmulas matemáticas que utilizan los nombres de las columnas, o campos, como variables.

---

<sup>6</sup> Observe con qué exquisito cuidado he evitado aquí la palabra *relación*. En inglés existen dos palabras diferentes: *relation* y *relationship*. Pero el equivalente más cercano a esta última sería algo así como *relacionalidad*, y eso suena peor que un párrafo del BOE.

## Restricciones de unicidad y claves primarias

Los tipos de restricciones siguen complicándose. Ahora se trata de realizar verificaciones que afectan los valores almacenados en varias filas. Las más importantes de estas validaciones son las denominadas *restricciones de unicidad*. Son muy fáciles de explicar en la teoría. Por ejemplo, no pueden haber dos filas en la tabla de departamentos con el mismo valor en la columna *Codigo*. Abusando del lenguaje, se dice que “la columna *Codigo* es única”.

En el caso de la tabla de departamentos, resulta que también existe una restricción de unicidad sobre la columna *Nombre*. Y no pasa nada. Sin embargo, quiero que el lector pueda distinguir sin problemas entre estas dos diferentes situaciones:

- 1 (Situación real) Hay una restricción de unicidad sobre *Codigo* y otra restricción de unicidad sobre *Nombre*.
- 2 (Situación ficticia) Hay una restricción de unicidad sobre la combinación de columnas *Codigo* y *Nombre*.

Esta segunda restricción posible es más relajada que la combinación real de dos restricciones (compruébelo). La unicidad de una combinación de columnas puede visualizarse de manera sencilla: si se “recortan” de la tabla las columnas que no participan en la restricción, no deben quedar registros duplicados después de esta operación. Por ejemplo, en la tabla de empleados, la combinación *Dpto* y *Nombre* es única.

La mayoría de los sistemas de bases de datos se apoyan en índices para hacer cumplir las restricciones de unicidad. Estos índices se crean automáticamente tomando como base a la columna o combinación de columnas que deben satisfacer estas condiciones. Antes de insertar un nuevo registro, y antes de modificar una columna de este tipo, se busca dentro del índice correspondiente para ver si se va a generar un valor duplicado. En tal caso se aborta la operación.

Así que una tabla puede tener una o varias restricciones de unicidad (o ninguna). Escoja una de ellas y désignela como *clave primaria*. ¿Cómo, de forma arbitraria? Casi: se supone que la clave primaria identifica unívocamente y de forma algo misteriosa la más recóndita esencia de los registros de una tabla. Pero para esto vale lo mismo cualquier otra restricción de unicidad, y en programación no vale recurrir al misticismo. Hay quienes justifican la elección de la clave primaria entre todas las restricciones de unicidad en relación con la integridad referencial, que estudiaremos en la próxima sección. Esto tampoco es una justificación, como veremos en breve. En definitiva, que todas las restricciones de unicidad son iguales, aunque algunas son más iguales que otras.

¿Quiere saber la verdad? He jugado con trampa en el párrafo anterior. Esa esencia misteriosa del registro no es más que un mecanismo utilizado por el modelo relacional para sustituir a los desterrados punteros. Podréis llamarlo *identidad del registro*, o

cualquier otro nombre rimbombante, pero no es más que una forma de disfrazar un puntero, y muy poco eficiente, por cierto. Observe la tabla de departamentos: entre el código y el nombre, ¿cuál columna elegiría como clave primaria? Por supuesto que el código, pues es el tipo de datos que menos espacio ocupa, y cuando tengamos el código en la mano podremos localizar el registro de forma más rápida que cuando tengamos el nombre.

De todos modos, hay quienes aprovechan inteligentemente la existencia de claves primarias: el Motor de Datos de Borland, ADO y en general casi todas las interfaces de acceso a bases de datos. De hecho, estas claves primarias muchas veces ofrecen una forma para simular el concepto de registro activo en una tabla SQL. Pero tendremos que esperar un poco para desentrañar cómo lo hacen.

## Integridad referencial

Este tipo de restricción es aún más complicada, y se presenta en la columna *Dpto* de la tabla de empleados. Es evidente que todos los valores de esta columna deben corresponder a valores almacenados en la columna *Codigo* de la tabla de departamentos. Siguiendo mi teoría de la conspiración de los punteros encubiertos, esto equivale a que cada registro de empleado tenga un puntero a un registro de departamento.

Un poco de terminología: a estas restricciones se les denomina *integridad referencial*, o *referential integrity*. También se dice que la columna *Dpto* de *Empleados* es una *clave externa* de esta tabla, o *foreign key*, en el idioma de Henry Morgan. Podemos llamar tabla dependiente, o tabla de detalles, a la tabla que contiene la clave externa, y tabla maestra a la otra. En el ejemplo que consideramos, la clave externa consiste en una sola columna, pero en el caso general podemos tener claves externas compuestas.

Como es fácil de ver, las columnas de la tabla maestra a las que se hace referencia en una integridad referencial deben satisfacer una restricción de unicidad. En la teoría original, estas columnas deberían ser la clave primaria, pero la mayoría de los sistemas relacionales actuales admiten cualquiera de las combinaciones de columnas únicas definidas.

Cuando se establece una relación de integridad referencial, la tabla dependiente asume responsabilidades:

- No se puede insertar una nueva fila con un valor en las columnas de la clave externa que no se encuentre en la tabla maestra.
- No se puede modificar la clave externa en una fila existente con un valor que no exista en la tabla maestra.

Pero también la tabla maestra tiene su parte de responsabilidad en el contrato, que se manifiesta cuando alguien intenta eliminar una de sus filas, o modificar el valor de su

clave primaria. En las modificaciones, en general, pueden desearse dos tipos diferentes de comportamiento:

- Se prohíbe la modificación de la clave primaria de un registro que tenga filas de detalles asociadas.
- La modificación de la clave se propagaría a las tablas dependientes.

Si se trata de un borrado, son tres los comportamientos posibles:

- Prohibir el borrado, si existen filas dependientes del registro.
- Borrar también las filas dependientes.
- Permitir el borrado y, en vez de borrar las filas dependientes, romper el vínculo asociando a la clave externa un valor por omisión, que en SQL casi siempre es el valor *nulo* (ver el siguiente capítulo).

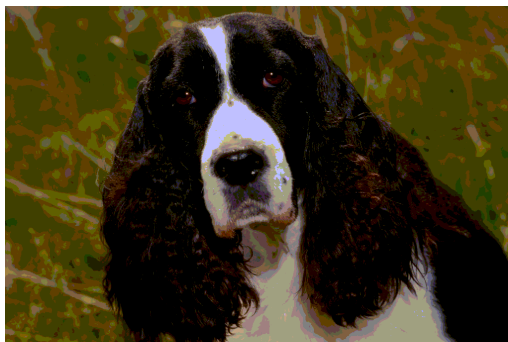
La forma más directa de implementar las verificaciones de integridad referencial es utilizar índices. Las responsabilidades de la tabla dependiente se resuelven comprobando la existencia del valor a insertar o a modificar en el índice asociado a la restricción de unicidad definida en la tabla maestra. Las responsabilidades de la tabla maestra se resuelven generalmente mediante índices definidos sobre la tabla de detalles.

## ¿Qué tiene de malo el modelo relacional?

El modelo relacional funciona. Además, funciona razonablemente bien. Pero como he tratado de explicar a lo largo de las secciones anteriores, en determinados aspectos significa un retroceso en comparación con modelos de datos anteriores. Me refiero a lo artificioso del proceso de eliminar los punteros implícitos de forma natural en el modelo semántico a representar. Esto se refleja también en la eficiencia de las operaciones. Supongamos que tenemos un registro de empleado en nuestras manos y queremos saber el nombre del departamento al que pertenece. Bien, pues tenemos que buscar el código de departamento dentro de un índice para después localizar físicamente el registro del departamento correspondiente y leer su nombre. Al menos, un par de accesos al disco duro. Compare con la sencillez de buscar directamente el registro de departamento dado su puntero (existen implementaciones eficientes del concepto de puntero cuando se trabaja con datos persistentes).

En este momento, las investigaciones de vanguardia se centran en las bases de datos orientadas a objetos, que retoman algunos conceptos del modelo de redes y del propio modelo relacional. Desde la década de los 70, la investigación matemática ha avanzado lo suficiente como para disponer de potentes lenguajes de interrogación sobre bases de datos arbitrarias. No quiero entrar a analizar el porqué no se ha acabado de imponer el modelo orientado a objetos sobre el relacional, pues en esto influyen tanto factores técnicos como comerciales. Pero es bueno que el lector sepa qué puede pasar en un futuro cercano.

Es sumamente significativo que la principal ventaja del modelo relacional, la posibilidad de realizar consultas *ad hoc* estuviera fuera del alcance del modelo “relacional” más popular a lo largo de los ochenta: dBase, y sus secuelas Clipper y FoxPro. Cuando escucho elogios sobre Clipper por parte de programadores que hicieron carrera con este lenguaje, pienso con tristeza que los elogios los merecen los propios programadores que pudieron realizar software funcional con herramientas tan primitivas. Mi aplauso para ellos; en ningún caso para el lenguaje. Y mi consejo de que abandonen el viejo buque (y las malas costumbres aprendidas durante la travesía) antes de que termine de hundirse.



**Ilustración 1** El perro de Codd

Y a todas estas, ¿qué pasó con la mascota de Mr. Codd? Lo inevitable: murió como consecuencia de un experimento fallido. Sin embargo, Brahma se apiadó de él, y reencarnó al año en un chico que, con el tiempo, se ha convertido en un exitoso programador afincado en el sur de la Florida. Al verlo, nadie pensaría que fue un perro en su vida anterior, de no ser por la manía que tiene de rascarse periódicamente la oreja. No obstante, por haber destrozado la tapicería del sofá de su dueño y orinarse un par de veces en la alfombra del salón, fue condenado a programar dos largos años en Visual Basic, hasta que Delphi (¡sí, fue Delphi!) llegó a su vida y se convirtió en un hombre feliz.

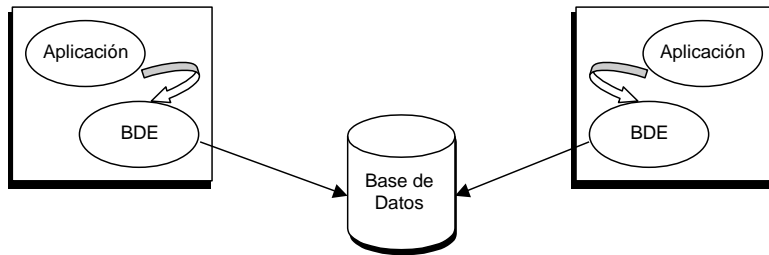
## **Bases de datos locales y servidores SQL**

Basta ya de teoría, y veamos los ingredientes con que contamos para cocinar aplicaciones de bases de datos. La primera gran división entre los sistemas de bases de datos existentes se produce entre los sistemas locales, o de escritorio, y las bases de datos SQL, o cliente/servidor.

A los sistemas de bases de datos locales se les llama de este modo porque comenzaron su existencia como soluciones baratas para un solo usuario, ejecutándose en un solo ordenador. Sin embargo, no es un nombre muy apropiado, porque más adelante estos sistemas crecieron para permitir su explotación en red. Tampoco es adecuado clasificarlas como “lo que queda después de quitar las bases de datos SQL”. Es cierto

que en sus inicios ninguna de estas bases de datos soportaba un lenguaje de consultas decente, pero esta situación también ha cambiado.

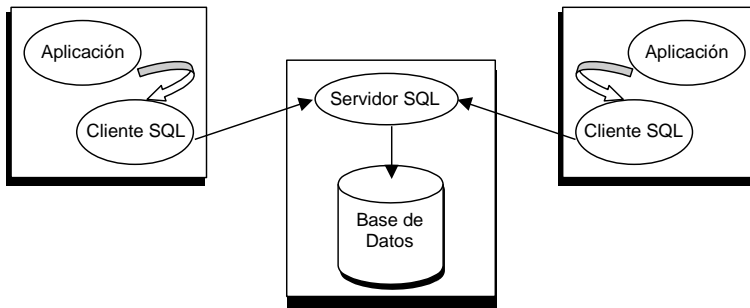
En definitiva, ¿cuál es la esencia de las bases de datos de escritorio? Pues el hecho de que la programación usual con las mismas se realiza en una sola capa. Todos estos sistemas utilizan como interfaz de aplicaciones un motor de datos que, en la era de la supremacía de Windows, se implementa como una DLL. En la época de MS-DOS, en cambio, eran funciones que se enlazaban estáticamente dentro del ejecutable. Observe el siguiente diagrama, que representa el uso en red de una base de datos “de escritorio”:



Aunque la segunda burbuja dice “BDE”, el Motor de Datos de Borland, sustituya estas siglas por DAO, y podrá aplicar el diagrama a Access. He representado la aplicación y el motor de datos en dos burbujas separadas, pero en realidad, constituyen un mismo ejecutable. Lo más importante, sin embargo, es que no existe un software central que sirva de árbitro para el acceso a la base de datos “física”. Es como si las dos aplicaciones deambularan a ciegas en un cuarto oscuro, tratando de sentarse en algún sitio libre. Por supuesto, la única forma que tienen de saberlo es intentar hacerlo y confiar en que no haya nadie debajo.

Debido a esta forma primitiva de resolver las inevitables colisiones, la implementación de la concurrencia, las transacciones y, en último término, la recuperación después de fallos, ha sido tradicionalmente el punto débil de las bases de datos de escritorio. Si está pensando en decenas o cientos de usuarios atacando simultáneamente a sus datos, y en ejércitos de extremidades inferiores listas para tropezar con cables, olvídense de Paradox, dBase y Access, pues necesitará una base de datos cliente/servidor.

Las bases de datos cliente/servidor, o bases de datos SQL, se caracterizan por utilizar al menos dos capas de software, como se aprecia en el siguiente diagrama:



El par aplicación + motor local de datos ya no tiene acceso directo a los ficheros de la base de datos, pues hay un nuevo actor en el drama: el servidor SQL. He cambiado el rótulo de la burbuja del motor de datos, y ahora dice “cliente SQL”. Esta es una denominación genérica. Para las aplicaciones desarrolladas con Delphi y el Motor de Datos de Borland, este cliente consiste en la combinación del BDE propiamente dicho *más* alguna biblioteca dinámica o DLL suministrada por el fabricante de la base de datos. En cualquier caso, todas estas bibliotecas se funden junto a la aplicación dentro de una misma capa de software, compartiendo el mismo espacio de memoria y procesamiento.

La división entre bases de datos de escritorio y las bases de datos SQL no es una clasificación tajante, pues se basa en la combinación de una serie de características. Puede que uno de estos días aparezca un sistema que mezcle de otra forma estos rasgos definitorios.

## Características generales de los sistemas SQL

El hecho de que exista un árbitro en las aplicaciones cliente/servidor hace posible implementar una gran variedad de técnicas y recursos que están ausentes en la mayoría de los sistemas de bases de datos de escritorio. Por ejemplo, el control de concurrencia se hace más sencillo y fiable, pues el servidor puede llevar la cuenta de qué clientes están accediendo a qué registros durante todo el tiempo. También es más fácil implementar transacciones atómicas, esto es, agrupar operaciones de modificación de forma tal que, o se efectúen todas, o ninguna llegue a tener efecto.

Pero una de las principales características de las bases de datos con las que vamos a trabajar es la forma peculiar en que “conversan” los clientes con el servidor. Resulta que estas conversaciones tienen lugar en forma de petición de ejecución de comandos del lenguaje SQL. De aquí el nombre común que reciben estos sistemas. Supongamos que un ordenador necesita leer la tabla de inventario. Recuerde que ahora no podemos abrir directamente un fichero situado en el servidor. Lo que realmente hace la estación de trabajo es pedirle al servidor que ejecute la siguiente instrucción SQL:

```
select * from Inventario order by CodigoProducto asc
```



El servidor calcula qué registros pertenecen al resultado de la consulta, y todo este cálculo tiene lugar en la máquina que alberga al propio servidor. Entonces, cada vez que el usuario de la estación de trabajo se mueve de registro a registro, el cliente SQL pide a su servidor el siguiente registro mediante la siguiente instrucción:

**fetch**

Más adelante necesitaremos estudiar cómo se realizan las modificaciones, inserciones, borrados y búsquedas, pero con esto basta por el momento. Hay una conclusión importante a la que ya podemos llegar: ¿qué es más rápido, pulsar un botón en la máquina de bebidas o mandar a un acólito a que vaya por una Coca-Cola? A no ser que usted sea más vago que yo, preferirá la primera opción. Bien, pues un sistema SQL es inherentemente más lento que una base de datos local. Antes manipulábamos directamente un fichero. Ahora tenemos que pedirselo a alguien, con un protocolo y con reglas de cortesía. Ese alguien tiene que entender nuestra petición, es decir, compilar la instrucción. Luego debe ejecutarla y solamente entonces procederá a enviarnos el primer registro a través de la red. ¿Está de acuerdo conmigo?

No pasa una semana sin que conozca a alguien que ha desarrollado una aplicación para bases de datos locales, haya decidido pasarla a un entorno SQL, y haya pensado que con un par de modificaciones en su aplicación era suficiente. Entonces descubre que la aplicación se ejecuta con mayor lentitud que antes, cae de rodillas, mira al cielo y clama: ¿dónde está entonces la ventaja de trabajar con sistemas cliente/servidor? Está, amigo mío, en la posibilidad de meter código en el servidor, y si fallamos en hacerlo estaremos desaprovechando las mejores dotes de nuestra base de datos.

Hay dos formas principales de hacerlo: mediante procedimientos almacenados y mediante *triggers*. Los primeros son conjuntos de instrucciones que se almacenan dentro de la propia base de datos. Se activan mediante una petición explícita de un cliente, pero se ejecutan en el espacio de aplicación del servidor. Por descontado, estos procedimientos no deben incluir instrucciones de entrada y salida. Cualquier proceso en lote que no contenga este tipo de instrucciones es candidato a codificarse como un procedimiento almacenado. ¿La ventaja?, que evitamos que los registros procesados por el procedimiento tengan que atravesar la barrera del espacio de memoria del servidor y viajar a través de la red hasta el cliente.

Los *triggers* son también secuencias de instrucciones, pero en vez de ser activados explícitamente, se ejecutan como preludio y coda de las tres operaciones básicas de actualización de SQL: **update**, **insert** y **delete**. No importa la forma en que estas tres operaciones se ejecuten, si es a instancias de una aplicación o mediante alguna herramienta incluida en el propio sistema; los *triggers* que se hayan programado se activarán en cualquier caso. Estos recursos se estudiarán más adelante, cuando exploremos el lenguaje SQL.



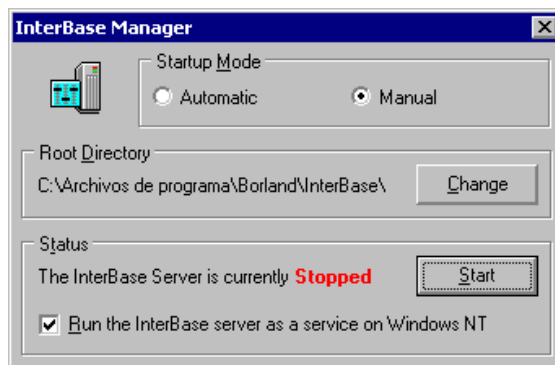
# InterBase

**L**A HISTORIA DE INTERBASE ES LARGA y agitada. Partiendo de una idea brillante, el uso de versiones para el control de la concurrencia, el producto fue creciendo y madurando hasta llegar a la cumbre con las versiones 5, 5.5 y 5.6; la versión 6 ofreció pocas adiciones o mejoras realmente útiles. Además, mientras el motor y las interfaces de programación evolucionaban, las herramientas de administración se mantenían en un estadio sumamente primitivo y poco funcional. Esa situación hizo mucho daño a la difusión del producto.

No obstante, InterBase sigue siendo una buena elección como sistema de bases de datos. Borland ha retomado el desarrollo del producto, y eso se ha dejado ver en las múltiples mejoras de la versión 6.5. Espero que sigamos viendo cambios espectaculares, a corto plazo, en la base de datos más utilizada por todos los que programamos en Delphi.

## Instalación y configuración

La instalación de InterBase es rápida y sencilla, y ocupa muy poco espacio en el disco duro una vez terminada. Existe una instalación completa, que contiene el servidor SQL y las herramientas del lado cliente, y una instalación especial sólo para el cliente. Cuando estudiemos los componentes IB Express, veremos cómo crear nuestras propias rutinas de instalación y desinstalación en Delphi, e integrar estos procesos en nuestras aplicaciones, si nos fuese necesario.



Una vez terminada la instalación, podemos comprobar el estado en que se encuentra el servidor, ejecutando la aplicación *InterBase Server Manager*, desde el grupo de programas de InterBase. La imagen de la página anterior muestra al *Server Manager* ejecutándose sobre Windows 2000 Server. Es fácil reconocer que se trata de Windows 2000 o Windows NT porque la casilla de verificación inferior (*Run... as a service...*) está disponible. Hay dos formas en la que puede ejecutarse el servidor de InterBase:

- Como una aplicación más, mostrando solamente un icono en la Bandeja de Iconos del escritorio de Windows. Esta es la única opción disponible cuando se instala el servidor en Windows 9x.
- Como un servicio de Windows NT/2000/XP.

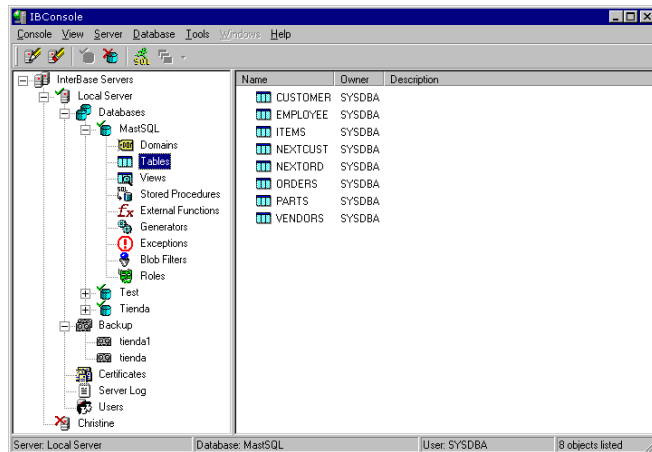
La segunda opción es la más recomendable, en general. Al poder ejecutarse como un servicio, no hace falta que un usuario del servidor active una sesión interactiva en ese ordenador. Un servidor en el que no existe una sesión interactiva es menos vulnerable a un ataque de *hackers*. Además, si por algún motivo se produce un error que reinicia el ordenador, el servidor volverá a estar disponible, aunque no haya ningún usuario cerca para iniciar una sesión.

Demos un salto a la parte superior de la ventana, al *Startup Mode*, o modo de arranque. Este es otro grado de libertad de la configuración: si dejamos activado *Automatic*, el servidor se iniciará automáticamente, ya sea como servicio o aplicación, al reiniciar el ordenador. Naturalmente, si hemos dicho que se ejecute como aplicación, tendremos que esperar a que un usuario inicie una sesión interactiva. Si elegimos *Manual*, tendremos que iniciar el servicio cuando tengamos necesidad de él, utilizando preferiblemente el *InterBase Server Manager*. En el ordenador donde capturé la imagen, InterBase se ejecuta solamente de manera excepcional; por esa razón aparece activada la opción *Manual*.

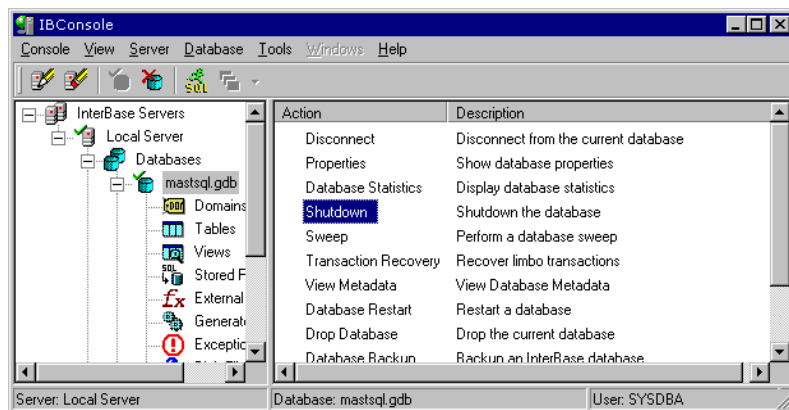
## La Consola de InterBase

La principal, y casi única, herramienta de administración que InterBase 6 nos proporciona es la Consola de InterBase. Se trata de una de esas aplicaciones que parecen fruto de un experimento diabólico de la ingeniería genética. En primer lugar, a pesar de su innegable inspiración en la Consola de Administración de Microsoft, la MMC, la mayoría de la información se muestra sólo para lectura, sin darnos otra posibilidad de modificarla que no sea lanzando las correspondientes instrucciones SQL.

En segundo lugar, parece una aplicación programada por una persona que nunca se ha sentado frente a Windows el tiempo suficiente. Existen una serie de convenciones acumuladas a lo largo de años sobre el significado de los elementos de una interfaz gráfica. Por ejemplo, ¿a quién se le ocurriría situar las acciones aplicables sobre un objeto en una lista plana, y exigir que el usuario haga doble clic sobre una línea para ejecutar el comando asociado? Pues al señor que programó la IB Console.



El comportamiento de los menús de contexto es errático. Cuando seleccionas un nodo como *Domains* en el árbol de la izquierda, si todavía no se han creado dominios, no se actualizarán las cabeceras de columnas en la lista de la derecha, dando la impresión de que la aplicación se ha colgado. Si quiere constatar los extremos de dejadez y chapucería a los que se llegaron durante el desarrollo de esta “herramienta”, ejecute el comando de menú *Help | About*. Si tiene instalada la versión que acompaña a Delphi 6, aparecerá un cuadro de diálogo con una gran imagen en su interior. Las proporciones de la imagen están distorsionadas, pero lo que molesta es ver el *copyright* al pie de la imagen: han comprimido el fichero en formato JPEG y el texto del *copyright* aparece emborronado. ¿Qué dice usted, que no es algo importante? Hombre, se trata de la imagen de la empresa...



## Conexiones con el servidor

Una vez que el servidor de InterBase está activo, es capaz de aceptar peticiones de clientes mediante varios protocolos, como TCP/IP, IPX/SPX, NetBEUI a través de *named pipes* (¿conductos con nombre?) y un protocolo especial de comunicación “lo-

cal". De los protocolos que permiten la comunicación entre diferentes ordenadores, TCP/IP es el más recomendado. Es incluso preferible la comunicación TCP/IP sobre la local cuando se trata de configurar aplicaciones CGI/ISAPI en las que el servidor HTTP reside en el mismo ordenador que el servidor de InterBase.

Para que un cliente puede comunicarse vía TCP/IP con el servidor de InterBase debe tener la siguiente línea dentro del fichero *services*, que se encuentra en el directorio de Windows:

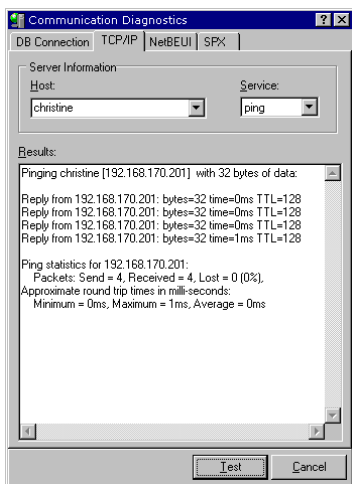
```
gds_db          3050/tcp
```

La comunicación la establece realmente la DLL *gds32.dll*, que normalmente se instala también en el directorio de sistema de Windows. Cuando la DLL solicita la conexión, pide que se abra un *socket*, o zócalo, entre el cliente y el servidor utilizando el nombre de servicio TCP simbólico *gds\_db*, que debe asociarse físicamente al puerto 3050 del mismo protocolo.

## NOTA

Si el servidor de InterBase reside en un ordenador remoto con una dirección fija en Internet, y el administrador del sistema "abre" el puerto 3050 para que cualquiera pueda acceder a él, es posible entonces realizar conexiones "directas" a ese servidor a través de Internet. Pero no es una técnica recomendable, salvo circunstancias extremas. Dejando a un lado incluso las consideraciones sobre seguridad, estas conexiones suelen ser poco eficientes.

Quiero también advertirle que es posible que el cliente de la versión con la que está trabajando actualmente no necesite ya la entrada mencionada en el fichero de servicios, porque este asunto se encuentra en la lista de adiciones y mejoras para la propia versión 6.



Para comprobar la conexión desde un cliente con un servidor debemos utilizar, ¡qué remedio!, la Consola de InterBase. Abra la Consola; si acaba de instalar la copia de InterBase que viene con Delphi 6 recibirá un mensaje de error lamentándose del estado del registro de Windows, o algo así de ininteligible. Ignore el mensaje y vuelva a ejecutar la aplicación.

Sitúese entonces en el panel de la izquierda, que es donde deberá desplegarse más adelante todo el árbol de objetos de administración; inicialmente solamente veremos un nodo llamado *InterBase Servers*. Vamos a ejecutar el comando de menú *Diagnose Connection...* pero eso se dice más rápido de lo que se

hace. El ingeniero genético que programó la consola tenía ideas propias sobre cómo despistar al usuario de una aplicación, y se prodigó en aplicarlas. Por ejemplo, cuando aún no hemos registrado servidor alguno, el comando de diagnóstico se selecciona

en el menú contextual de la raíz del árbol. Sin embargo, cuando ya hemos registrado un servidor, el menú contextual del nodo raíz desaparece, y tenemos que diagnosticar las conexiones seleccionando uno de los servidores registrados. Esto, por supuesto, es un disparate total, porque da la impresión de que la conexión que se va a verificar corresponde al servidor que ya sabemos que funciona perfectamente. Una vez que hemos dado con el comando, lo ejecutamos para que aparezca el cuadro de diálogo que he mostrado un poco antes.

Podemos efectuar dos comprobaciones diferentes, conectándonos a un servidor o a una base de datos concreta, y debemos comenzar por la primera, utilizando la página *TCP/IP*. Simplemente, teclee en *Host* el nombre del servidor, seleccione el servicio *ping*, pulse el botón *Test* y examine el resultado que aparece en el centro de la ventana.

Esta prueba tan sencilla puede ahorrarnos dolores de cabeza. Algunos administradores configuran los ordenadores clientes con TCP/IP pero no proporcionan medio alguno para traducir nombres de máquinas en direcciones IP numéricas. Todo parece funcionar estupendamente, porque pueden pedirle al Explorador de Windows que se conecte a un recurso del servidor. Sin embargo, no pueden establecer conexiones con el servidor de InterBase. La explicación es que el servicio de compartir archivos e impresoras dispone de métodos propios de resolución de nombres, que no pueden ser aprovechados por el protocolo TCP/IP básico.

En versiones anteriores de InterBase podíamos utilizar solamente el nombre simbólico del servidor en una conexión; no se aceptaba la dirección numérica. El problema descrito se resolvía entonces creando un fichero con el nombre *hosts* y situándolo en uno de estos directorios:

```
Win9x, ME  C:\Windows\System
WinNT, W2K C:\WinNT\System32\Drivers\Etc
```

Un fichero *hosts* típico tiene el siguiente aspecto:

```
127.0.0.1      localhost
192.168.170.200 nara
192.168.170.201 christine
192.168.170.202 julia
192.168.170.203 lonewolf
```

En este caso se trata del fichero *hosts* de mi red “doméstica”. Observe la línea correspondiente a *localhost*: se trata de un convenio habitual en redes TCP/IP para referirnos al ordenador en que estamos trabajando.

Para finalizar la prueba, asegúrese de que el servidor de InterBase se encuentra activo en el ordenador correspondiente, y sustituya el servicio *ping* por *gds\_db*. Un fallo en esta última prueba indicaría, con casi total seguridad, que el fichero *services* no contiene la descripción del servicio solicitado... o que se nos ha olvidado iniciar InterBase en el servidor.

**NOTA**

Se aconseja que las aplicaciones para servidores de Internet que se conectan a bases de datos InterBase no utilicen el protocolo local, sino TCP/IP. El motivo son las restricciones de acceso que establece Windows para procesos que se ejecutan desde una cuenta de servicio. Al parecer, el protocolo local exige más permisos que TCP/IP. No piense que se trata solamente de un problema de InterBase. También sucede lo mismo con Personal Oracle: es muy probable que tengamos problemas si utilizamos conexiones con el protocolo local *bequeath*.

## Las bases de datos de InterBase

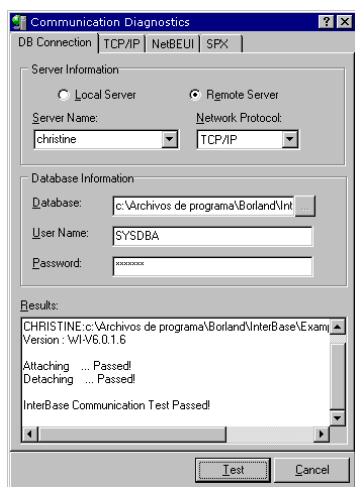
Una vez superada la primera prueba, podemos intentar conectarnos a una base de datos del servidor. Y aquí encontramos una de las características más incómodas de InterBase: para referirnos a una base de datos en un servidor remoto tenemos que utilizar el nombre del fichero físico donde ésta reside, ¡incluyendo la ruta completa!

Por ejemplo, la instalación de InterBase copia los siguientes ficheros de bases de datos en el servidor:

```
c:\Archivos de programa\Borland\InterBase\Examples\Database\employee.gdb
c:\Archivos de programa\Borland\InterBase\Examples\Database\intlemp.gdb
```

Y la instalación de Delphi nos deja estos otros dos:

```
c:\Archivos de programa\Archivos comunes\Borland Shared\Data\mastsql.gdb
c:\Archivos de programa\Archivos comunes\Borland Shared\Data\employee.gdb
```



Como podemos comprobar en la imagen, ¡tenemos que ser capaces de teclear esa retahíla de directorios sin pestañear, si queremos verificar el estado de salud de nuestro servidor! Pero lo más chocante es que, si el servidor estuviese correctamente configurado, no deberíamos tener acceso a ese fichero a través de la red.

Efectivamente, el acceso al fichero debe ser responsabilidad única del servicio de InterBase en el ordenador remoto; la ruta que indicamos se utiliza en beneficio del servidor. Como comprobaremos al estudiar el sistema de seguridad de InterBase, si permitimos que los usuarios tengan acceso al fichero mediante el sistema de archivos, estaríamos

poniendo en peligro la seguridad de todo el sistema.

¿Existen alternativas al sistema de identificación de bases de datos de InterBase, o se trata de algo inevitable? La respuesta es la primera: se podrían utilizar identificadores simbólicos para las bases de datos, que se registrarían en el servidor, y asociarían allí



el nombre o alias con el fichero físico que contiene los datos. Es la técnica empleada por SQL Server, por ejemplo, y tiene dos importantes ventajas:

- Como el cliente no emplea la ruta física, es más sencillo cambiar la ubicación de la base de datos dentro del servidor.
- Es más difícil piratear la base de datos cuando se desconoce la ubicación exacta de sus ficheros.

Cuando nos conectamos a una base de datos local, podemos utilizar un botón para mostrar el conocido diálogo de apertura de ficheros. Esto no sucede sólo en la ventana de diagnóstico, sino casi en cualquier opción que nos obligue a señalar una base de datos.

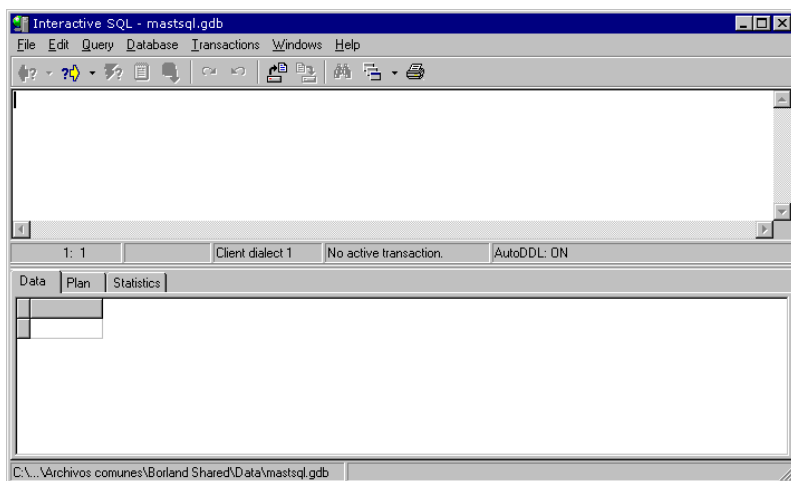
Como ayuda parcial para este problema, la Consola de InterBase nos permite registrar en su árbol de objetos *alias* asociados a los ficheros físicos de las bases de datos cuya existencia conozcamos. La información sobre los alias se guarda en el Registro de Windows, bajo el nodo *HKEY\_CURRENT\_USER*. Una vez que hemos registrado una base de datos para un servidor, podemos efectuar las típicas operaciones: crear y restaurar copias de seguridad, forzar la recogida de basura, verificar la integridad, extraer información sobre el esquema relacional, etc.

Lo malo es que no podemos realizar casi ninguna de las anteriores operaciones si no registramos antes la base de datos y le asignamos un alias.

## Interactive SQL

Antes de seguir con el capítulo, tengo que presentar una de las “ventanas” de la IB Console que utilizaremos con frecuencia. En versiones anteriores, esta ventana se ejecutaba como una aplicación por separado, con nombre propio: *Interactive SQL*. Ahora debe invocarla desde el menú *Tools* de la consola, o pulsando un botón verde y amarillo, como la bandera de Brasil, de la barra de herramientas. Si forzamos un poco la vista para intentar descifrar el significado del icono, veremos que se trata de un hombrecillo verde corriendo sobre las siglas SQL. Seguramente, representa la reacción de cualquier persona normal cuando descubre que tiene que trabajar con la IB Console: ponerse verde y salir huyendo.

Esta ventana es donde tecleamos instrucciones SQL y le suplicamos al motor de InterBase que sea bueno y las ejecute. Si ha trabajado con versiones anteriores del servidor, preste mucha atención, porque hay cambios en la metodología de trabajo. Podemos teclear directamente las instrucciones en el panel superior, y ejecutarlas de forma inmediata, al igual que siempre. Pero ahora podemos hacer algo nuevo: teclear varias instrucciones, convenientemente separadas, y ejecutarlas en conjunto. Claro, en el panel inferior solamente se reflejará el resultado de la última instrucción de selección ejecutada.



El separador que se admite inicialmente es el punto y coma. Realmente, InterBase llama *terminador* al separador, pero es una denominación inexacta, porque no hace falta utilizarlo en la última instrucción de un grupo. Cuando estudiemos los *triggers* y procedimientos almacenados, descubriremos la necesidad de cambiar el separador para poder ejecutar sentencias de creación para estos objetos.

La otra forma de ejecutar instrucciones SQL consiste en agruparlas en un fichero. Podemos cargar uno de estos ficheros mediante el comando de menú *Query | Load script*<sup>7</sup>. En esta versión de InterBase, el comando se limita a cargar las instrucciones dentro del panel superior, y somos nosotros los encargados de pegarle un golpe a la tecla de ejecución. Pero en versiones anteriores, el comando de carga de *scripts* estaba rodeado de un halo místico:

- 1 Era la única forma de ejecutar varias instrucciones de golpe.
- 2 Cuando utilizabas el comando, las instrucciones del fichero se ejecutaban automáticamente.

Además, ciertas instrucciones solamente se podían ejecutar desde un fichero, como las de creación de base de datos, que explicaré en la siguiente sesión.

## Creación de bases de datos

Podemos crear una base de datos utilizando dos métodos diferentes: mediante un cuadro de diálogo que se puede disparar desde la Consola de InterBase e Interactive SQL, o ejecutando la instrucción SQL en modo texto en esta última herramienta, ya sea tecleándola directamente o incluyéndola en un fichero de *script*. Antes de aden-

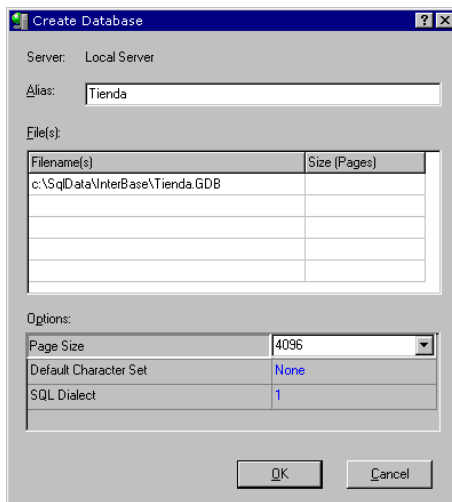
---

<sup>7</sup> No es por ser quisquilloso, pero en Windows los comandos de menú que muestran un cuadro de diálogo se marcan con puntos suspensivos al final. Claro, como el insigne programador de la consola estuvo trabajando demasiado tiempo con su PlayStation ...

trarnos en la sintaxis, veamos las características físicas de una base de datos de InterBase.

Ya hemos visto que una base de datos en InterBase se aloja en uno o más ficheros, aunque lo usual es que se utilice uno solamente. La división en varios ficheros se utiliza para evitar que un fichero individual sobrepase la barrera de los 4GB, pero también sirve como forma primitiva de distribuir datos entre varios discos físicamente distintos. Digo “primitiva” porque no hay modo de controlar si la tabla XYZ se almacena en tal o cual partición: los ficheros se van llenando secuencialmente hasta que llegamos al límite fijado, y es entonces que comienza el llenado del siguiente.

Da lo mismo las extensiones que utilicemos para los ficheros, pero es común utilizar las siglas *gdb*, por *Grotton Database (System)*, que fue el nombre de la compañía que diseñó originalmente el producto. Cada fichero está formado por un número variable de *páginas*, todas con el mismo tamaño. En el momento de crear la base de datos, o de restaurarla a partir de una copia de seguridad, es cuando podemos especificar el tamaño de página que deseamos.



La instrucción que crea una base de datos es la siguiente:

```
create database 'fichero'
[user 'usuario' [password 'contraseña']]
[page_size páginas]
[length páginas]
[default character set 'conjunto-caracteres']
[file 'fichero' [length páginas | starting at página ... ] ... ]
```

El siguiente ejemplo muestra el ejemplo más común de creación:

```
create database 'C:\Marteens\IntrBase\Examples\Prueba.GDB'
user 'SYSDBA' password 'masterkey'
page_size 4096;
```

### LA CONTRASEÑA DEL ADMINISTRADOR

Cuando InterBase acaba de instalarse, el único usuario que reconoce el servidor es el administrador del mismo, que recibe el nombre predefinido de *SYSDBA*, con la contraseña *masterkey*. En el manual *Operations Guide*, se afirma que esta contraseña inicial es *changeme*, algo rotundamente falso.

En versiones anteriores, InterBase utilizaba páginas de 1024 bytes por omisión. En la mayoría de los casos, es conveniente utilizar un tamaño mayor de página; de este modo, el acceso a disco es más eficiente, entran más claves en las páginas de un índice con lo cual disminuye la profundidad de estos, y mejora también el almacenamiento de campos de longitud variable. Mi sugerencia es utilizar páginas de 4096 ó 8192 bytes. Sin embargo, si sus aplicaciones trabajan con pocas filas de la base de datos, como la típica aplicación del cajero automático, puede ser más conveniente mantener un tamaño pequeño de página, pues la lectura de éstas tarda entonces menos tiempo, y el *buffer* puede realizar las operaciones de reemplazo de páginas en memoria más eficientemente.

También podemos indicar el tamaño inicial de la base de datos en páginas. Normalmente esto no hace falta, pues InterBase hace crecer automáticamente el fichero *gdb* cuando es necesario. Pero si tiene en mente insertar grandes cantidades de datos sobre la base recién creada, puede ahorrar el tiempo de crecimiento utilizando la opción **length**. La siguiente instrucción crea una base de datos reservando un tamaño inicial de 1 MB:

```
create database 'C:\Marteens\IntrBase\Examples\Prueba.GDB'
user 'SYSDBA' password 'masterkey'
page_size 2048
length 512
default character set ISO8859_1;
```

Esta instrucción muestra también cómo especificar el *conjunto de caracteres* utilizado por omisión en la base de datos. Más adelante, se pueden definir conjuntos especiales para cada tabla, de ser necesario. El conjunto de caracteres determina, fundamentalmente, de qué forma se ordenan alfabéticamente los valores alfanuméricos. Los primeros 127 caracteres de todos los conjuntos de datos coinciden; es en los restantes valores donde puede haber diferencias.

### ADVERTENCIA

¿Se ha dado cuenta de que el nombre del usuario y su contraseña son opcionales? Es que podemos crear variables de entorno (*environment variables*) en el sistema operativo con los nombres *ISC\_USER* e *ISC\_PASSWORD*, y asignarles nuestro nombre de usuario y contraseña. Por supuesto, es una *pésima* idea.

También existe una instrucción de conexión, con la siguiente sintaxis:

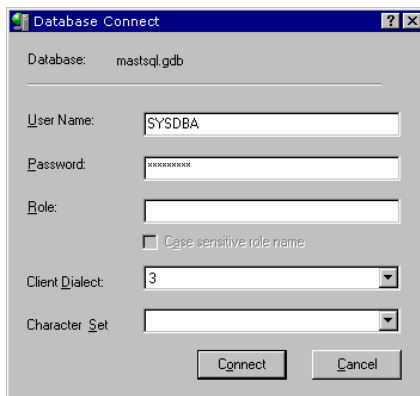
```
connect 'fichero'
[user 'usuario' [password 'contraseña']]
[cache páginas]
[role 'nombre']
```

Observe que hay dos nuevos parámetros: el número de páginas en caché y una cadena que corresponde a “algo” llamado *role* en inglés, y que traduciré como *función*, aunque debería ser *papel*, como el que desempeñan los actores<sup>8</sup>; espero que el contexto donde use la traducción me eche una mano. Más adelante veremos qué son esas *funciones*. En cuanto al tamaño de la caché, como he dicho, se indica el número de páginas; el tamaño total en bytes depende, naturalmente, del tamaño de las páginas de la base de datos en juego. Por omisión, las conexiones se configuran con una caché de 75 páginas.

Por último una curiosidad: no existe una instrucción simétrica **disconnect** que podamos utilizar para cerrar una conexión desde el *Interactive SQL*.

## Dialectos

Si prestó atención al cuadro de diálogo de InterBase para la creación de bases de datos, habrá notado la existencia de un parámetro llamado *SQL Dialect*. Encontraremos también un parámetro similar al conectarnos a una base de datos desde *Interactive SQL*. Esta vez, el nombre del parámetro es *Client Dialect*:



Este rollo de los dialectos es el mecanismo que InterBase 6 nos ofrece para facilitar la transición a la nueva versión. O al menos, eso es lo que se pretende. El dialecto 1 es que el más se parece a InterBase 5.6, y el dialecto 3 quiere decir InterBase 6 desenfrenado. El dialecto 2 corresponde a un estado intermedio entre ambos extremos.

---

<sup>8</sup> He encontrado verdadera resistencia en España al uso del anglicismo *rol*: al oír la palabra, el oyente tuerce la boca como si le hubieran hecho masticar un limón. Sin embargo, estoy sorprendido, porque aquí se utiliza sin aspavientos la expresión “juegos de rol”.

El dialecto se aplica a dos entidades diferentes, con distinto propósito. Se puede aplicar a una base de datos, en el momento de su creación, pero también a una conexión de un cliente. El dialecto por omisión de las conexiones se puede establecer en el comando de menú *Edit|Options*.

Aunque la documentación afirma que se puede cambiar el dialecto de una base de datos existente, no he podido hacerlo. Tampoco funciona, en mi versión de la consola, la presunta instrucción **show sql dialect**, mencionada en la también presunta documentación.

## Tipos de datos

Antes de poder crear tablas, tenemos que saber qué tipos de datos podemos emplear. SQL estándar define un conjunto de tipos de datos que todo sistema debe implementar. Ahora bien, la interpretación exacta de estos tipos no está completamente especificada. Cada sistema de bases de datos ofrece, además, sus propios tipos nativos.

Estos son los tipos de datos aceptados por InterBase 6:

Tipo de dato	Tamaño	Observaciones
char( <i>n</i> ), varchar( <i>n</i> )	<i>n</i> bytes	Longitud fija; longitud variable
integer, int	32 bits	
smallint	16 bits	
float	4 bytes	Equivale al tipo homónimo de C
double precision	8 bytes	Equivale al <i>double</i> de C
numeric( <i>prec</i> , <i>esc</i> )	<i>prec</i> =1-18, <i>esc</i> <= <i>prec</i>	Variante “exacta” de <i>decimal</i>
decimal( <i>prec</i> , <i>esc</i> )	<i>prec</i> =1-18, <i>esc</i> <= <i>prec</i>	
date		Almacena la fecha
time		Almacena la hora
timestamp	64 bits	Almacena la fecha y la hora
blob	No hay límite	

Las versiones anteriores de InterBase no soportaban los tres tipos relacionados con fechas y horas: **date**, **time** y **timestamp**. Solamente se admitía **date**, pero con el significado del actual **timestamp**, es decir, conteniendo tanto la fecha como la hora. Otra mejora de la versión 6 ha sido el rango de valores aceptado por **numeric** y **decimal**. Antes, la máxima precisión era de 15 dígitos decimales, mientras que ahora se admiten hasta 18. Ha cambiado también la representación de estos tipos, pero eso lo dejaremos para la próxima sección.

### MITOLOGIA

En la documentación se menciona la existencia de un tipo numérico de 64 bits. Este tipo se utiliza en la implementación de **numeric** y **decimal**, cuando la precisión supera cierto límite. Pero la documentación insiste en que se pueden declarar directamente columnas pertenecientes a ese tipo, que en ocasiones recibe el nombre de **int64**.

Creo que se trata de una leyenda, como la del Hombre del Saco, o la del Perro Cazador que Devolvió Granada a su Dueño, porque no encuentro la forma de crear una tabla con una columna perteneciente a ese tipo.

El tipo **blob** (*Binary Large Object* = Objeto Binario Grande) se utiliza para almacenar información de longitud variable, generalmente de gran tamaño: documentos enteros o imágenes, generalmente. En principio, a InterBase no le preocupa qué formato tiene la información almacenada. Pero para cada columna de tipo **blob** se define un *subtipo*, un valor entero que ofrece una pista acerca del formato del campo. InterBase interpreta el subtipo 0 como el formato por omisión: ningún formato. El subtipo 1 representa texto, como el tipo memo de otros sistemas. Se pueden especificar subtipos definidos por el programador; en este caso, los valores empleados deben ser negativos. La especificación de subtipos se realiza mediante la cláusula **sub\_type**:

```
Comentarios blob sub_type 1
```

Existe otro parámetro asociado a los tipos **blob**: el *segment length*, o longitud del segmento. La documentación de InterBase se contradice acerca del significado de esta opción. En uno de los manuales se nos asegura que el tamaño de segmento tiene que ver con un *buffer* que utiliza la interfaz de programación, al parecer en el lado cliente, y que simplemente representa el tamaño medio esperado para el contenido de la columna. La redacción del párrafo en cuestión debe haber estado a cargo de algún graduado en Alquimia, porque un poco más adelante nos advierte con mucho misterio que si pasamos datos más grandes que el segmento, podemos provocar un cataclismo. Esto es lo que dice la *Data Definition Guide*.

Por otra parte, en la *Operations Guide* se nos aconseja para que definamos el tamaño de los segmentos **blob** de modo que coincidan con el tamaño de página. Y añade algo así como que si nos portamos bien y hacemos lo que nos dicen, se pueden alcanzar tasas de transferencia de 20MB por segundo. ¡Un conflicto de autoridad! Para ser sinceros, no tengo por qué sentir la más mínima confianza en un señor que me planta en la cara una cifra “abstracta” de tantos megas por segundo ... sin aclararme con qué hardware ha hecho la prueba. Puestos a escoger, elijo creer al autor de la *Data Definition Guide*, que al menos no va presumiendo de velocidades.

Como en estas cosas hay que ser prácticos, he hecho la prueba de insertar montones de registros en tablas con el mismo esquema pero longitudes de segmento distintas: en una utilicé el valor por omisión, de 80 bytes, y en la otra hice lo que recomendaba el velocista: segmentos de 4096 bytes. El tamaño medio de los blobs que utilicé era de unos 2.048 bytes. Tras insertar 32.000 registros varias veces, no noté diferencias significativas de velocidad. Y para rematar, el espacio ocupado por ambas tablas era casi idéntico.

En definitiva, si queremos especificar un tamaño de segmento, a la vez que un subtipo, debemos utilizar una declaración como la siguiente:

```
Comentarios blob sub_type 1 segment size 2048
```

Si le irrita tener que teclear tantas palabras claves juntas, puede escribir también:

```
Comentarios blob(2048, 1)
```

Es decir, ¡invirtiendo el orden de los parámetros respecto a la sintaxis alternativa!

Otra de las peculiaridades de InterBase que tiene que ver con los tipos de datos es el soporte de *matrices multidimensionales*. Se pueden crear columnas que contengan matrices de tipos simples, con excepción del tipo **blob**, de hasta 16 dimensiones. Sin embargo, el soporte desde SQL es mínimo, y Delphi no reconoce estos campos correctamente, y debemos trabajar con ellos como si fueran campos blob.

## Representación de datos en InterBase

¿Qué diferencias hay entre los tipos **char** y **varchar**? Cuando una aplicación graba una cadena en una columna de tipo **varchar**, InterBase almacena exactamente la misma cantidad de caracteres que ha especificado la aplicación, con independencia del ancho máximo de la columna. Cuando se recupera el valor más adelante, la cadena obtenida tiene la misma longitud que la original. Ahora bien, si la columna de que hablamos ha sido definida como **char**, en el proceso de grabación se le añaden automáticamente espacios en blanco al final del valor para completar la longitud de la cadena. Cuando se vuelve a leer la columna, la aplicación recibe estos espacios adicionales.

¿Quiere esto decir que ahorraremos espacio en la base de datos utilizando siempre el tipo **varchar**? ¡Conclusión prematura! InterBase utiliza registros de longitud variable para representar las filas de una tabla, con el fin de empaquetar la mayor cantidad posible de registros en cada página de la base de datos. Como parte de la estrategia de disminución del tamaño, cuando se almacena una columna de tipo **char** se eliminan automáticamente los espacios en blanco que puede contener al final, y estos espacios se restauran cuando alguien recupera el valor de dicha columna. Más aún: para almacenar un **varchar** es necesario añadir a la propia representación de la cadena un valor entero con la longitud de la misma. Como resultado final, ¡una columna de tipo **varchar** consume más espacio que una de tipo **char**!

¿Para qué, entonces, quiero el tipo **varchar**?, se preguntará el lector. Conviene que recuerde que si utiliza el tipo **char** recibirá valores con espacios en blanco adicionales al final de los mismos, y que tendrá que utilizar frecuentemente la función *TrimRight* para eliminarlos. El tipo **varchar** le ahorra este incordio.

También le será útil conocer cómo InterBase representa los tipos **numeric** y **decimal**. El factor decisivo de la representación es el número de dígitos de la precisión. Si es menor que 5, **numeric** y **decimal** pueden almacenarse dentro de un tipo entero de 16 bits, o **smallint**; si es menor que 10, en un **integer** de 32 bits; en caso contrario, se almacenan en columnas de tipo **double precision**.



## Creación de tablas

Como fuente de ejemplos para este capítulo, utilizaremos el típico esquema de un sistema de pedidos. Las tablas que utilizaremos serán las siguientes:

Tabla	Propósito
<i>Clientes</i>	Los clientes de nuestra empresa
<i>Empleados</i>	Los empleados que reciben los pedidos
<i>Articulos</i>	Las cosas que intentamos vender
<i>Pedidos</i>	Pedidos realizados
<i>Detalles</i>	Una fila por cada artículo vendido

La instrucción de creación de tablas tiene la siguiente sintaxis en InterBase:

```
create table NombreDeTabla [external file NombreFichero] (
    DefColumna [, DefColumna | Restriccion ... ]
);
```

La opción **external file** es propia de InterBase e indica que los datos de la tabla deben residir en un fichero externo al principal de la base de datos. Aunque el formato de este fichero no es ASCII, es relativamente sencillo de comprender y puede utilizarse para importar y exportar datos de un modo fácil entre InterBase y otras aplicaciones. En lo sucesivo no haremos uso de esta cláusula.

Para crear una tabla tenemos que definir columnas y restricciones sobre los valores que pueden tomar estas columnas. La forma más sencilla de definición de columna es la que sigue:

```
NombreColumna TipoDeDato
```

Por ejemplo:

```
create table Empleados (
    Codigo          integer,
    Nombre          varchar(30),
    Contrato        date,
    Salario         integer
);
```

## Columnas calculadas

Con InterBase tenemos la posibilidad de crear *columnas calculadas*, cuyos valores se derivan a partir de columnas existentes, sin necesidad de ser almacenados físicamente, para lo cual se utiliza la cláusula **computed by**. Aunque para este tipo de columnas podemos especificar explícitamente un tipo de datos, es innecesario, porque se puede deducir de la expresión que define la columna:

```

create table Empleados (
    Codigo          integer,
    Nombre          varchar,
    Apellidos       varchar,
    Salario         integer,
    NombreCompleto  computed by (Nombre || ' ' || Apellidos),
    /* ... */
);

```

El operador `||` sirve para concatenar cadenas de caracteres en InterBase.

En general, no es buena idea definir columnas calculadas en el servidor, sino que es preferible el uso de campos calculados en el cliente. Si utilizamos **computed by** hacemos que los valores de estas columnas viajen por la red con cada registro, aumentando el tráfico en la misma.

## Valores por omisión

Otra posibilidad es la de definir valores por omisión para las columnas. Durante la inserción de filas, es posible no mencionar una determinada columna, en cuyo caso se le asigna a esta columna el valor por omisión. Si no se especifica algo diferente, el valor por omisión de SQL es **null**, el valor desconocido. Con la cláusula **default** cambiamos este valor:

```

Salario          integer default 0,
FechaContrato    date default "Now",

```

Observe en el ejemplo anterior el uso del literal *"Now"*, para inicializar la columna con la fecha y hora actual en InterBase. En Oracle se utilizaría la función *sysdate*:

```

FechaContrato    date default sysdate

```

Si se mezclan las cláusulas **default** y **not null** en Oracle o InterBase, la primera debe ir antes de la segunda. En MS SQL Server, por el contrario, la cláusula **default** debe ir después de las especificaciones **null** ó **not null**:

```

FechaContrato    datetime not null default (getdate())

```

## Restricciones de integridad

Durante la definición de una tabla podemos especificar condiciones que deben cumplirse para los valores almacenados en la misma. Por ejemplo, no nos basta saber que el salario de un empleado es un entero; hay que aclarar también que en circunstancias normales es también un entero positivo, y que no podemos dejar de especificar un salario a un trabajador. También nos puede interesar imponer condiciones más complejas, como que el salario de un empleado que lleva menos de un año con nosotros no puede sobrepasar cierta cantidad fija. En este epígrafe veremos cómo expresar estas *restricciones de integridad*.

La restricción más frecuente es pedir que el valor almacenado en una columna no pueda ser nulo. En el capítulo sobre manipulación de datos estudiaremos en profundidad este peculiar valor. El que una columna no pueda tener un valor nulo quiere decir que hay que suministrar un valor para esta columna durante la inserción de un nuevo registro, pero también que no se puede modificar posteriormente esta columna de modo que tenga un valor nulo. Esta restricción, como veremos dentro de poco, es indispensable para poder declarar claves primarias y claves alternativas. Por ejemplo:

```
create table Empleados (
    Codigo integer not null,
    Nombre varchar(30) not null,
    /* ... */
);
```

Cuando la condición que se quiere verificar es más compleja, se puede utilizar la cláusula **check**. Por ejemplo, la siguiente restricción verifica que los códigos de provincias se escriban en mayúsculas:

```
Provincia          varchar(2) check (Provincia = upper(Provincia))
```

Existen dos posibilidades con respecto a la ubicación de la mayoría de las restricciones: colocar la restricción a nivel de columna o a nivel de tabla. A nivel de columna, si la restricción afecta solamente a la columna en cuestión; a nivel de tabla si hay varias columnas involucradas. En mi humilde opinión, es más claro y legible expresar todas las restricciones a nivel de tabla, pero esto en definitiva es materia de gustos.

La cláusula **check** de InterBase permite incluso expresiones que involucran a otras tablas. Más adelante, al tratar la integridad referencial, veremos un ejemplo sencillo de esta técnica. Por el momento, analice la siguiente restricción, expresada a nivel de tabla:

```
create table Detalles (
    RefPedido      int not null,
    NumLinea       int not null,
    RefArticulo    int not null,
    Cantidad       int default 1 not null,
    Descuento      int default 0 not null,

    check (Descuento between 0 and 50 or 'Intuitive Sight' =
        (select Nombre from Clientes
         where Codigo =
            (select RefCliente from Pedidos
             where Numero = Detalles.RefPedido))),
    /* ... */
);
```

Esta cláusula dice, en pocas palabras, que solamente el autor de este libro puede beneficiarse de descuentos superiores al 50%. ¡Algún privilegio tenía que corresponderme!

## Claves primarias y alternativas

Las restricciones **check** nos permiten con relativa facilidad imponer condiciones sobre las filas de una tabla que pueden verificarse examinando solamente el registro activo. Cuando las reglas de consistencia involucran a varias filas a la vez, la expresión de estas reglas puede complicarse bastante. En último caso, una combinación de cláusulas **check** y el uso de *triggers* o disparadores nos sirve para expresar *imperativamente* las reglas necesarias. Ahora bien, hay casos típicos de restricciones que afectan a varias filas a la vez que se pueden expresar *declarativamente*; estos casos incluyen a las restricciones de claves primarias y las de integridad referencial.

Mediante una clave primaria indicamos que una columna, o una combinación de columnas, debe tener valores únicos para cada fila. Por ejemplo, en una tabla de clientes, el código de cliente no debe repetirse en dos filas diferentes. Esto se expresa de la siguiente forma:

```
create table Clientes (
    Codigo integer not null primary key,
    Nombre varchar(30) not null,
    /* ... */
);
```

Si una columna pertenece a la clave primaria, debe estar especificada como no nula. Observe que en este caso hemos utilizado la restricción a nivel de columna. También es posible tener claves primarias compuestas, en cuyo caso la restricción hay que expresarla a nivel de tabla. Por ejemplo, en la tabla de detalles de pedidos, la clave primaria puede ser la combinación del número de pedido y el número de línea dentro de ese pedido:

```
create table Detalles (
    NumPedido integer not null,
    NumLinea integer not null,
    /* ... */
    primary key (NumPedido, NumLinea)
);
```

Solamente puede haber una clave primaria por cada tabla. De este modo, la clave primaria representa la *identidad* de los registros almacenados en una tabla: la información necesaria para localizar unívocamente un objeto. No es imprescindible especificar una clave primaria al crear tablas, pero es recomendable como método de trabajo.

Sin embargo, es posible especificar que otros grupos de columnas también poseen valores únicos dentro de las filas de una tabla. Estas restricciones son similares en sintaxis y semántica a las claves primarias, y utilizan la palabra reservada **unique**. En la jerga relacional, a estas columnas se le denominan *claves alternativas*. Una buena razón para tener claves alternativas puede ser que la columna designada como clave primaria sea en realidad una *clave artificial*. Se dice que una clave es artificial cuando no tiene un equivalente semántico en el sistema que se modela. Por ejemplo, el código

de cliente no tiene una existencia *real*; nadie va por la calle con un 666 grabado en la frente. La verdadera clave de un cliente puede ser, además de su alma inmortal, su DNI<sup>9</sup>. Pero el DNI debe almacenarse en una cadena de caracteres, y esto ocupa mucho más espacio que un código numérico. En este caso, el código numérico se utiliza en las referencias a clientes, pues al tener menor tamaño la clave, pueden existir más entradas en un bloque de índice, y el acceso por índices es más eficiente. Entonces, la tabla de clientes puede definirse del siguiente modo:

```
create table Clientes (
    Codigo integer not null,
    DNI      varchar(9) not null,
    /* ... */
    primary key (Codigo),
    unique (DNI)
);
```

Por cada clave primaria o alternativa definida, InterBase crea un índice único para mantener la restricción. Este índice se bautiza según el patrón *rdbs\$primaryN*, donde *N* es un número único asignado por el sistema.

## Integridad referencial

Un caso especial y frecuente de restricción de integridad es la conocida como restricción de *integridad referencial*. También se le denomina restricción por *clave externa* o *foránea* (*foreign key*). Esta restricción especifica que el valor almacenado en un grupo de columnas de una tabla debe encontrarse en los valores de las columnas en alguna fila de otra tabla, o de sí misma. Por ejemplo, en una tabla de pedidos se almacena el código del cliente que realiza el pedido. Este código debe corresponder al código de algún cliente almacenado en la tabla de clientes. La restricción puede expresarse de la siguiente manera:

```
create table Pedidos (
    Codigo      integer not null primary key,
    Cliente     integer not null references Clientes(Codigo),
    /* ... */
);
```

O, utilizando restricciones a nivel de tabla:

```
create table Pedidos (
    Codigo      integer not null,
    Cliente     integer not null,
    /* ... */
    primary key (Codigo),
    foreign key (Cliente) references Clientes(Codigo)
);
```

La columna o grupo de columnas a la que se hace referencia en la tabla maestra, la columna *Codigo* de *Clientes* en este caso, debe ser la clave primaria de esta tabla o ser

---

<sup>9</sup> El Documento Nacional de Identidad español.

una clave alternativa, esto es, debe haber sido definida una restricción **unique** sobre la misma.

Si todo lo que pretendemos es que no se pueda introducir una referencia a cliente inválida, se puede sustituir la restricción declarativa de integridad referencial por esta cláusula:

```
create table Pedidos(
    /* ... */
    check (Cliente in (select Codigo from Clientes))
);
```

La sintaxis de las expresiones será explicada en profundidad en el próximo capítulo, pero esta restricción **check** sencillamente comprueba que la referencia al cliente exista en la columna *Codigo* de la tabla *Clientes*.

## Acciones referenciales

Sin embargo, las restricciones de integridad referencial ofrecen más que esta simple comprobación. Cuando tenemos una de estas restricciones, el sistema toma las riendas cuando tratamos de eliminar una fila maestra que tiene filas dependientes asociadas, y cuando tratamos de modificar la clave primaria de una fila maestra con las mismas condiciones. El estándar SQL-3 dicta una serie de posibilidades y reglas, denominadas *acciones referenciales*, que pueden aplicarse.

Lo más sencillo es prohibir estas operaciones, y es la solución que adoptan MS SQL Server (sólo hasta la versión 7, porque la situación ha cambiado en SQL Server 2000) y las versiones de InterBase anteriores a la 5. En la sintaxis más completa de SQL-3, esta política puede expresarse mediante la siguiente cláusula:

```
create table Pedidos(
    Codigo          integer not null primary key,
    Cliente         integer not null,
    /* ... */
    foreign key (Cliente) references Clientes(Codigo)
    on delete no action
    on update no action
);
```

Otra posibilidad es permitir que la acción sobre la tabla maestra se propague a las filas dependientes asociadas: eliminar un cliente puede provocar la desaparición de todos sus pedidos, y el cambio del código de un cliente modifica todas las referencias a este cliente. Por ejemplo:

```
create table Pedidos(
    Codigo          integer not null primary key,
    Cliente         integer not null
                    references Clientes(Codigo)
                    on delete no action on update cascade
    /* ... */
);
```

Observe que puede indicarse un comportamiento diferente para los borrados y para las actualizaciones.

En el caso de los borrados, puede indicarse que la eliminación de una fila maestra provoque que en la columna de referencia en las filas de detalles se asigne el valor nulo, o el valor por omisión de la columna de referencia:

```
insert into Empleados(Codigo, Nombre, Apellidos)
values(-1, 'D''Arche', 'Jeanne');

create table Pedidos(
  Codigo          integer not null primary key,
  Cliente         integer not null
                  references Clientes(Codigo)
                  on delete no action on update cascade,
  Empleado        integer default -1 not null
                  references Empleados(Codigo)
                  on delete set default on update cascade
  /* ... */
);
```

InterBase actualmente implementa todas estas estrategias, para lo cual necesita crear índices que le ayuden a verificar las restricciones de integridad referencial. La comprobación de la existencia de la referencia en la tabla maestra se realiza con facilidad, pues se trata en definitiva de una búsqueda en el índice único que ya ha sido creado para la gestión de la clave. Para prohibir o propagar los borrados y actualizaciones que afectarían a filas dependientes, la tabla que contiene la cláusula **foreign key** crea automáticamente un índice sobre las columnas que realizan la referencia. De este modo, cuando sucede una actualización en la tabla maestra, se pueden localizar con rapidez las posibles filas afectadas por la operación. Este índice nos ayuda en Delphi en la especificación de relaciones *master/detail* entre tablas. Los índices creados automáticamente para las relaciones de integridad referencial reciben nombres con el formato *rdbs\$foreignN*, donde *N* es un número generado automáticamente.

## Nombres para las restricciones

Cuando se define una restricción sobre una tabla, sea una verificación por condición o una clave primaria, alternativa o externa, es posible asignarle un nombre a la restricción. Este nombre es utilizado por InterBase en el mensaje de error que se produce al violarse la restricción, pero su uso fundamental es la manipulación posterior por parte de instrucciones como **alter table**, que estudiaremos en breve. Por ejemplo:

```
create table Empleados(
  /* ... */
  Salario         integer default 0,
  constraint      SalarioPositivo check(Salario >= 0),
  /* ... */
  constraint      NombreUnico
                  unique(Apellidos, Nombre)
);
```

También es posible utilizar nombres para las restricciones cuando éstas se expresan a nivel de columna. Las restricciones a las cuales no asignamos nombre reciben uno automáticamente por parte del sistema.

## Definición y uso de dominios

SQL permite definir algo similar a los tipos de datos de los lenguajes tradicionales. Si estamos utilizando cierto tipo de datos con frecuencia, podemos definir un dominio para ese tipo de columna y utilizarlo consistentemente durante la definición del esquema de la base de datos. Un dominio, sin embargo, va más allá de la simple definición del tipo, pues permite expresar restricciones sobre la columna y valores por omisión. La sintaxis de una definición de dominio en InterBase es la siguiente:

```
create domain NombreDominio [as
    TipoDeDato
    [ValorPorOmisión]
    [not null]
    [check (Condición)]
    [collate Criterio]
```

Cuando sobre un dominio se define una restricción de chequeo, no contamos con el nombre de la columna. Si antes expresábamos la restricción de que los códigos de provincia estuvieran en mayúsculas de esta forma:

```
Provincia      varchar(2) check(Provincia = upper(Provincia))
```

ahora necesitamos la palabra reservada **value** para referirnos al nombre de la columna:

```
create domain CodProv as
    varchar(2)
    check(value = upper(value));
```

El dominio definido, *CodProv*, puede utilizarse ahora para definir columnas:

```
Provincia      CodProv
```

Las cláusulas **check** de las definiciones de dominio no pueden hacer referencia a columnas de otras tablas.

Si vamos a utilizar el BDE para acceder a InterBase, tendremos un estímulo adicional para definir dominios: el Diccionario de Datos los reconoce y asocia automáticamente a conjuntos de atributos (*attribute sets*). Así se ahorra mucho tiempo en la configuración de los objetos de acceso a campos.

## Creación de índices

Como ya he explicado, InterBase crea índices de forma automática para mantener las restricciones de clave primaria, unicidad y de integridad referencial. En la mayoría de



los casos, estos índices bastan para que el sistema funcione eficientemente. No obstante, es necesario en ocasiones definir índices sobre otras columnas. Esta decisión depende de la frecuencia con que se realicen consultas según valores almacenados en estas columnas, o de la posibilidad de pedir que una tabla se ordene de acuerdo al valor de las mismas. Por ejemplo, en la tabla de empleados es sensato pensar que el usuario de la base de datos deseará ver a los empleados listados por orden alfabético, o que querrá realizar búsquedas según un nombre y unos apellidos.

La sintaxis para crear un índice es la siguiente:

```
create [unique] [asc[ending] | desc[ending]] index Indice
on Tabla (Columna [, Columna ...])
```

Por ejemplo, para crear un índice sobre los apellidos y el nombre de los empleados necesitamos la siguiente instrucción:

```
create index NombreEmpleado on Empleados (Apellidos, Nombre)
```

Los índices creados por InterBase son todos sensibles a mayúsculas y minúsculas, y todos son mantenidos por omisión. El concepto de índice definido por expresiones y con condición de filtro es ajeno a la filosofía de SQL; este tipo de índices no se adapta fácilmente a la optimización automática de consultas. InterBase no permite tampoco crear índices sobre columnas definidas con la cláusula **computed by**.

Aunque definamos índices descendentes sobre una tabla en una base de datos SQL, el Motor de Datos de Borland no lo utilizará para ordenar tablas. Exactamente lo que sucede es que el BDE no permite que una tabla (no una consulta) pueda estar ordenada descendentemente por alguna de sus columnas, aunque la tabla mencione un índice descendente en su propiedad *IndexName*. En tal caso, el orden que se establece utiliza las mismas columnas del índice, pero ascendentemente.

Hay otro problema relacionado con los índices de InterBase: solamente pueden recorrerse en un sentido. Si definimos un índice ascendente sobre determinada columna de una tabla, y realizamos una consulta sobre la tabla con los resultados ordenados descendentemente por el valor de esa columna, InterBase no podrá aprovechar el índice creado. También se ven afectadas las consultas que piden el mínimo y el máximo de una columna indizada. Para encontrar el mínimo se aprovecha un índice ascendente, pero para el máximo hace falta contar también con el descendente.

Esto se debe, al parecer, a que los índices de InterBase utilizan una técnica de “compresión” de prefijos para ahorrar espacio en el almacenamiento de las claves y, consecuentemente, ahorrar también tiempo de búsqueda. Abriré un diccionario en una página cualquiera: estoy viendo la traducción de *descodificar*, *descolgar*, *descollante*... Tengo la impresión de haberme saltado alguna palabra importante, pero lo importante es que todas esas palabras consecutivas comparten unas mismas letras iniciales: *desco*. Podríamos, por lo tanto, aislar este prefijo y no repetirlo en todas las palabras. Cuando se trata de índices arbóreos, existen varias formas de implementar esta téc-

nica, pero todas tienen algo en común: no es posible restablecer una palabra completa a no ser que sigamos una ruta prevista. Si no pasamos por el sitio donde hemos establecido que el prefijo actual es *desco*, nunca sabremos el significado real de la “palabra” *jonamiento*.

## Modificación de tablas e índices

SQL nos permite ser sabios y humanos a la vez: podemos equivocarnos en el diseño de una tabla o de un índice, y corregir posteriormente nuestro disparate. Sin caer en el sentimentalismo filosófico, es bastante común que una vez terminado el diseño de una base de datos surja la necesidad de añadir nuevas columnas a las tablas para almacenar información imprevista, o que tengamos que modificar el tipo o las restricciones activas sobre una columna determinada.

La forma más simple de la instrucción de modificación de tablas es la que elimina una columna de la misma:

```
alter table Tabla drop Columna [, Columna ...]
```

También se puede eliminar una restricción si conocemos su nombre. Por ejemplo, esta instrucción puede originar graves disturbios sociales:

```
alter table Empleados drop constraint SalarioPositivo;
```

Se pueden añadir nuevas columnas o nuevas restricciones sobre una tabla existente:

```
alter table Empleados add EstadoCivil varchar(8);  
alter table Empleados  
  add check (EstadoCivil in ("Soltero", "Casado", "Polígamo"));
```

Para los índices existen también instrucciones de modificación. En este caso, el único parámetro que se puede configurar es si el índice está activo o no:

```
alter index Indice (active | inactive);
```

Si un índice está inactivo, las modificaciones realizadas sobre la tabla no se propagan al índice, por lo cual necesitan menos tiempo para su ejecución. Si va a efectuar una entrada masiva de datos, quizás sea conveniente desactivar algunos de los índices secundarios, para mejorar el rendimiento de la operación. Luego, al activar el índice, éste se reconstruye dando como resultado una estructura de datos perfectamente balanceada. Estas instrucciones pueden ejecutarse periódicamente, para garantizar índices con tiempo de acceso óptimo:

```
alter index NombreEmpleado inactive;  
alter index NombreEmpleado active;
```

Otra instrucción que puede mejorar el rendimiento del sistema y que está relacionada con los índices es **set statistics**. Este comando calcula las estadísticas de uso de las

claves dentro de un índice. El valor obtenido, conocido como *selectividad* del índice, es utilizado por InterBase para elaborar el plan de implementación de consultas. Normalmente no hay que invocar a esta función explícitamente, pero si las estadísticas de uso del índice han variado mucho es quizás apropiado utilizar la instrucción:

```
set statistics index NombreEmpleado;
```

Por último, las instrucciones **drop** nos permiten borrar objetos definidos en la base de datos, tanto tablas como índices:

```
drop table Tabla;  
drop index Indice;
```

## Creación de vistas

Uno de los recursos más potentes de SQL, y de las bases de datos relacionales en general, es la posibilidad de definir tablas “virtuales” a partir de los datos almacenados en tablas “físicas”. Para definir una de estas tablas virtuales hay que definir qué operaciones relacionales se aplican a qué tablas bases. Este tipo de tabla recibe el nombre de *vista*.

Como todavía no conocemos el lenguaje de consultas, que nos permite especificar las operaciones sobre tablas, postergaremos el estudio de las vistas para más adelante.

## Creación de usuarios

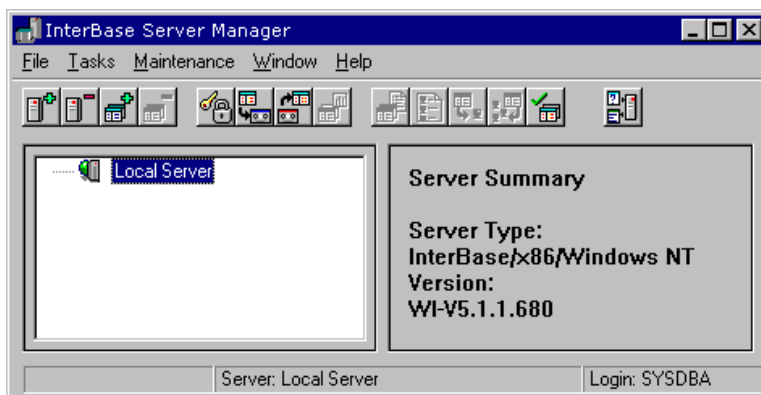
InterBase soporta el concepto de usuarios a nivel del servidor, no de las bases de datos. Inicialmente, todos los servidores definen un único usuario especial: *SYSDBA*. Este usuario tiene los derechos necesarios para crear otros usuarios y asignarles contraseñas. Toda esa información se almacena en la base de datos *isc4.gdb*, que se instala automáticamente con InterBase.

### ADVERTENCIA

El sistema de seguridad de InterBase tiene un punto débil demasiado evidente. Cualquier usuario con acceso al disco duro puede sustituir el fichero *isc4.gdb* con uno suyo. Más grave aún: si copiamos el fichero *gdb* de la base de datos en un servidor en el cual conozcamos la contraseña del administrador, tendremos acceso total a los datos, aunque este acceso nos hubiera estado vedado en el servidor original.

La moraleja es que hay que impedir que cualquier mequetrefe pueda acceder, ya sea físicamente o a través del sistema de compartición de ficheros, a los ficheros donde residen nuestras queridas bases de datos. Quiero que sepa también que casi cualquier otro sistema de bases de datos tendrá un punto débil similar, pero por lo general será más complicado realizar este tipo de cambalaches.

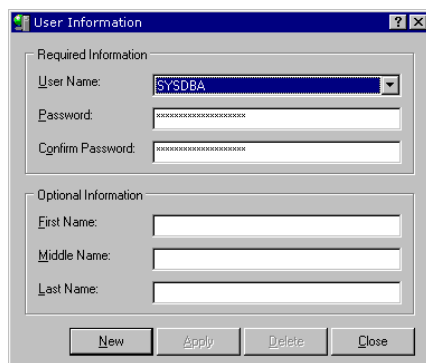
Curiosamente, el SQL de InterBase no tiene instrucciones para la creación y modificación de usuarios. La gestión de los nombres de usuarios y sus contraseñas se realizaba, en las versiones anteriores a la 6, por medio de la utilidad *Server Manager*.



Pero a partir de InterBase 6, hay que armarse de valor y utilizar la temida Consola. Primero tenemos que seleccionar uno de los servidores registrados, e identificarnos como el administrador, utilizando la cuenta *sysdba*. ¿Ve entonces ese último nodo, en el árbol de objetos que cuelga del servidor? Selecciónelo, y a la derecha aparecerá la lista de usuarios registrados en ese servidor.

Para dar con el menú de contexto que permite añadir, modificar o eliminar usuarios, tendrá que pasar por unas cuantas peripecias. Por ejemplo, si pulsa el botón derecho del ratón sobre ese melancólico icono que muestra a dos personajes con peinados de los años setenta, no aparecerá el menú de contexto. Si lo hace sobre la lista de la derecha, puede que aparezca y puede que no: hace falta que uno de los usuarios existentes esté seleccionado. Yo digo: ¿para qué necesito seleccionar un usuario cuando lo que quiero es crear uno nuevo?

Podemos también acceder a los comandos de gestión de usuarios seleccionando el comando de menú *Server|User security*. En tal caso aparece directamente el siguiente cuadro de diálogo modal:



En la lista desplegable del primer control se incluyen todos los usuarios registrados en el servidor. Podemos seleccionar uno de ellos, modificar su contraseña o los campos de información opcional, y pulsar el botón *Apply*. Si deseamos crear un nuevo

usuario, debemos pulsar *New*. Entonces el combo se transforma en un cuadro de edición para que tecleemos el nombre del nuevo usuario. Para grabarlo, al igual que antes, hay que pulsar *Apply*.

La gestión de usuarios es un privilegio exclusivo del administrador de la base de dato; InterBase, por su parte, reconoce como administrador solamente al usuario que se identifique como *SYSDBA*. Este nombre no puede cambiarse, pero es casi una obligación cambiar su contraseña al terminar la instalación de InterBase. Sin embargo, podemos eliminar al administrador de la lista de usuarios del sistema. Si esto sucede, ya no será posible añadir o modificar nuevos usuarios en ese servidor. Así que tenga cuidado con lo que hace.

## Asignación de privilegios

Una vez creados los objetos de la base de datos, es necesario asignar derechos sobre los mismos a los demás usuarios. Inicialmente, el dueño de una tabla es el usuario que la crea, y tiene todos los derechos de acceso sobre la tabla. Los derechos de acceso indican qué operaciones pueden realizarse con la tabla. Naturalmente, los nombres de estos derechos o privilegios coinciden con los nombres de las operaciones correspondientes:

Privilegio	Operación
<b>select</b>	Lectura de datos
<b>update</b>	Modificación de datos existentes
<b>insert</b>	Creación de nuevos registros
<b>delete</b>	Eliminación de registros
<b>references</b>	Acceso a la clave primaria para integridad referencial
<b>all</b>	Los cinco privilegios anteriores
<b>execute</b>	Ejecución (para procedimientos almacenados)

La instrucción que otorga derechos sobre una tabla es la siguiente:

```
grant Privilegios on Tabla to Usuarios [with grant option]
```

Por ejemplo:

```
/* Derecho de sólo-lectura al público en general */
grant select on Articulos to public;
/* Todos los derechos a un par de usuarios */
grant all privileges on Clientes to Spade, Marlowe;
/* Monsieur Poirot sólo puede modificar salarios (¡qué peligro!) */
grant update(Salario) on Empleados to Poirot;
/* Privilegio de inserción y borrado, con opción de concesión */
grant insert, delete on Empleados to Vance with grant option;
```

He mostrado unas cuantas posibilidades de la instrucción. En primer lugar, podemos utilizar la palabra clave **public** cuando queremos conceder ciertos derechos a todos los usuarios. En caso contrario, podemos especificar uno o más usuarios como desti-

natarios del privilegio. Luego, podemos ver que el privilegio **update** puede llevar entre paréntesis la lista de columnas que pueden ser modificadas. Por último, vemos que a Mr. Philo Vance no solamente le permiten contratar y despedir empleados, sino que también, gracias a la cláusula **with grant option**, puede conceder estos derechos a otros usuarios, aún no siendo el creador de la tabla. Esta opción debe utilizarse con cuidado, pues puede provocar una propagación descontrolada de privilegios entre usuarios indeseables.

¿Y qué pasa si otorgamos privilegios y luego nos arrepentimos? No hay problema, pues para esto tenemos la instrucción **revoke**:

```
revoke [grant option for] Privilegios on Tabla from Usuarios
```

Hay que tener cuidado con los privilegios asignados al público. La siguiente instrucción no afecta a los privilegios de Sam Spade sobre la tabla de artículos, porque antes se le ha concedido al público en general el derecho de lectura sobre la misma:

```
/* Spade se ríe de este ridículo intento */  
revoke all on Articulos from Spade;
```

Existen variantes de las instrucciones **grant** y **revoke** pensadas para asignar y retirar privilegios sobre tablas a procedimientos almacenados, y para asignar y retirar derechos de ejecución de procedimientos a usuarios.

## Privilegios e integridad referencial

El privilegio **references** está relacionado con las restricciones de integridad referencial. Supongamos que usted no quiere que la persona que teclea los pedidos en su base de datos tenga acceso a los datos privados de sus clientes. Pero SQL no permite otorgar el privilegio **select** sobre un subconjunto de las columnas: o todas o ninguna.

La solución llevaría dos pasos: el primero sería otorgar **references** al vendedor sobre la tabla *CLIENTES*; doy por sentado que esta persona ya tiene concedido **select** y al menos **insert** sobre *PEDIDOS*:

```
grant references(IDCliente) on CLIENTES to Vendedor;
```

Como puede ver, **references** nos permite teclear las columnas que el sistema puede leer, en representación del vendedor, para verificar que el código de cliente que éste que ha incluido al insertar el pedido. Dicho sea de paso, también **update** permite restringir el acceso sobre un subconjunto de todas las columnas:

```
grant update(Pagado, Enviado) on PEDIDOS to Vendedor;
```

Ahora bien, ¿de dónde podría el vendedor extraer los códigos de clientes? Ya hemos dicho que no se pueden incluir columnas al otorgar **select**. Pero hay un rodeo a nuestra disposición: podemos crear una *vista* sobre la tabla de clientes, que solamente

incluya las columnas públicas de la tabla. No hemos estudiado la instrucción de creación de vistas, pero incluyo su código para que se haga una idea:

```
create view DatosClientes as
select IDCliente, Nombre, Apellidos
from Clientes;

grant select on DatosClientes to Vendedor;
```

Tenga presente que, incluso otorgando permisos sobre esta vista, sigue siendo necesario otorgar **references** sobre la tabla original. Recuerde que el motor de InterBase va a realizar las verificaciones necesarias sobre esa tabla base, no sobre nuestra vista.

## Perfiles

Los perfiles (*roles*, en inglés) son una especificación del SQL-3 que InterBase 5 ha implementado. Si los usuarios se almacenan y administran a nivel de servidor, los roles, en cambio, se definen a nivel de cada base de datos. De este modo, podemos trasladar con más facilidad una base de datos desarrollada en determinado servidor, con sus usuarios particulares, a otro servidor, en el cual existe históricamente otro conjunto de usuarios.

El sujeto de la validación por contraseña sigue siendo el usuario. La relación entre usuarios y perfiles es la siguiente: un usuario puede *asumir* un perfil al conectarse a la base de datos; actualmente, InterBase no permite asumir otros perfiles a partir de este momento. A un perfil se le pueden otorgar privilegios exactamente igual que a un usuario, utilizando **grant** y **revoke**. Cuando un usuario asume un perfil, los privilegios de éste se suman a los privilegios que se le han concedido como usuario. Por supuesto, un usuario debe contar con la autorización para asumirlo; la autorización se extiende también mediante las instrucciones **grant** y **revoke**.

¿Un ejemplo? Primero necesitamos crear los perfiles adecuados en la base de datos:

```
create role Domador;
create role Payaso;
create role Mago;
```

Ahora debemos asignar los permisos sobre tablas y otros objetos a los roles. Esto no impide que, en general, se puedan también asignar permisos específicos a usuarios puntuales:

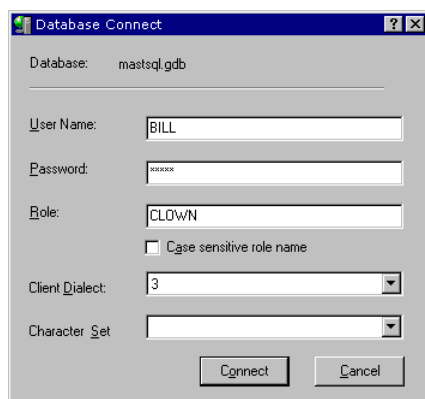
```
grant all privileges on Animales to Domador, Mago;
grant select on Animales to Payaso;
```

Hasta aquí no hemos mencionado a los usuarios, por lo que los resultados de estas instrucciones son válidos de servidor a servidor. Finalmente, debemos asignar los usuarios en sus respectivos perfiles, y esta operación sí depende del conjunto de usuarios de un servidor:

```
grant Payaso to Bill, Steve, RonaldMcDonald;  
grant Domador to Ian with admin option;
```

La opción **with admin option** me permite asignar el perfil de domador a otros usuarios. De este modo, siempre habrá quien se ocupe de los animales cuando me ausente del circo por vacaciones.

La pregunta importante es: ¿cómo puede el usuario indicar a InterBase que desea asumir determinado perfil al iniciar una conexión? Si nos conectamos con las herramientas del propio InterBase, comprobaremos que existe un cuadro de edición que nos hace esa pregunta. Por supuesto, podemos dejarlo vacío, si nos queremos atener exclusivamente a nuestros privilegios como usuarios:



Pero si la conexión la realizamos desde Delphi tendremos que programar un poco. Los perfiles se añaden al soporte de InterBase del BDE en la versión 5.0.1.23; las versiones anteriores no contaban con esta posibilidad. El SQL Link de InterBase contiene ahora el parámetro *ROLE NAME*, y es aquí donde debemos indicar el perfil del usuario. Lamentablemente, el diálogo de conexión del componente *TDatabase* tampoco tiene en consideración la posibilidad de admitir perfiles, por lo que si necesitamos que el usuario dicte dinámicamente los parámetros de conexión tendremos que interceptar el evento *OnLogin* del componente *TDatabase*.

Por supuesto, IB Express y DB Express, por ser interfaces más recientes, implementan la especificación de perfiles desde el primer momento.

## Un ejemplo completo de *script SQL*

Incluyo a continuación un ejemplo completo de *script SQL* con la definición de tablas e índices para una sencilla aplicación de entrada de pedidos. En un capítulo posterior, ampliaremos este *script* para incluir *triggers*, generadores y procedimientos almacenados que ayuden a expresar las reglas de empresa de la base de datos.



```

create database "C:\Pedidos\Pedidos.GDB"
user "SYSDBA" password "masterkey"
page_size 2048;

/* Creación de las tablas */

create table Clientes (
    Codigo          integer not null,
    Nombre          varchar(30) not null,
    Direccion1      varchar(30),
    Direccion2      varchar(30),
    Telefono        varchar(15),
    UltimoPedido    date default "Now",

    primary key     (Codigo)
);

create table Empleados (
    Codigo          integer not null,
    Apellidos       varchar(20) not null,
    Nombre          varchar(15) not null,
    FechaContrato   date default "Now",
    Salario         integer,
    NombreCompleto  computed by (Nombre || ' ' || Apellidos),

    primary key     (Codigo)
);

create table Articulos (
    Codigo          integer not null,
    Descripcion     varchar(30) not null,
    Existencias     integer default 0,
    Pedidos         integer default 0,
    Costo           integer,
    PVP             integer,

    primary key     (Codigo)
);

create table Pedidos (
    Numero          integer not null,
    RefCliente      integer not null,
    RefEmpleado     integer,
    FechaVenta      date default "Now",
    Total           integer default 0,

    primary key     (Numero),
    foreign key     (RefCliente) references Clientes (Codigo)
                    on delete no action on update cascade
);

create table Detalles (
    RefPedido       integer not null,
    NumLinea        integer not null,
    RefArticulo     integer not null,
    Cantidad        integer default 1 not null,
    Descuento       integer default 0 not null,
    check (Descuento between 0 and 100),

```

```

        primary key      (RefPedido, NumLinea),
        foreign key      (RefPedido) references Pedidos (Numero),
                           on delete cascade on update cascade
        foreign key      (RefArticulo) references Articulos (Codigo)
                           on delete no action on update cascade
    );

    /* Indices secundarios */

    create index NombreCliente on Clientes(Nombre);
    create index NombreEmpleado on Empleados(Apellidos, Nombre);
    create index Descripcion on Articulos(Descripcion);

    /***** FIN DEL SCRIPT *****/

```

## Consultas y modificaciones

**D**ESDE SU MISMO ORIGEN, la definición del modelo relacional de Codd incluía la necesidad de un lenguaje para realizar consultas *ad-hoc*. Debido a la forma particular de representación de datos utilizada por este modelo, el tener relaciones o tablas y no contar con un lenguaje de alto nivel para reintegrar los datos almacenados es más bien una maldición que una bendición. Es asombroso, por lo tanto, cuánto tiempo vivió el mundo de la programación sobre ordenadores personales sin poder contar con SQL o algún mecanismo similar. Aún hoy, cuando un programador de Clipper o de COBOL comienza a trabajar en Delphi, se sorprende de las posibilidades que le abre el uso de un lenguaje de consultas integrado dentro de sus aplicaciones.

La instrucción **select**, del Lenguaje de Manipulación de Datos de SQL, nos permite consultar la información almacenada en una base de datos relacional. La sintaxis y posibilidades de esta sola instrucción son tan amplias y complicadas como para merecer un capítulo para ella solamente. En éste estudiaremos las posibilidades de las instrucciones **update**, **insert** y **delete**, que permiten la modificación del contenido de las tablas de una base de datos.

Para los ejemplos de este capítulo utilizaré la base de datos *mastsql.gdb* que viene con los ejemplos de Delphi, en el siguiente directorio:

*C:\Archivos de programa\Archivos comunes\Borland Shared\Data*

### La instrucción **select**: el lenguaje de consultas

A grandes rasgos, la estructura de la instrucción **select** es la siguiente:

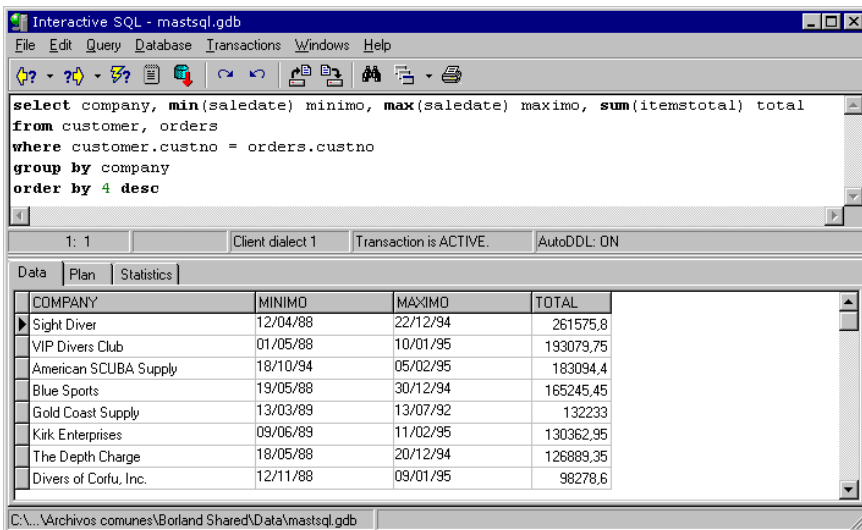
```
select [distinct] lista-de-expresiones
from lista-de-tablas
[where condición-de-selección]
[group by lista-de-columnas]
[having condición-de-selección-de-grupos]
[order by lista-de-columnas]
[union instrucción-de-selección]
```

¿Qué se supone que “hace” una instrucción **select**? Esta es la pregunta del millón: una instrucción **select**, en principio, no “hace” sino que “define”. La instrucción define un conjunto virtual de filas y columnas, o más claramente, define una tabla virtual. Qué se hace con esta “tabla virtual” es ya otra cosa, y depende de la aplicación que le estemos dando. Si estamos en un intérprete que funciona en modo texto, puede ser que la ejecución de un **select** se materialice mostrando en pantalla los resultados, página a página, o quizás en salvar el resultado en un fichero de texto. En Delphi, las instrucciones **select** se utilizan para “alimentar” un componente denominado *TQuery*, al cual se le puede dar casi el mismo uso que a una tabla “real”, almacenada físicamente.

A pesar de la multitud de secciones de una selección completa, el formato básico de la misma es muy sencillo, y se reduce a las tres primeras secciones:

```
select lista-de-expresiones
from lista-de-tablas
[where condición-de-selección]
```

La cláusula **from** indica de dónde se extrae la información de la consulta, en la cláusula **where** opcional se dice qué filas deseamos en el resultado, y con **select** especificamos los campos o expresiones de estas filas que queremos obtener. Muchas veces se dice que la cláusula **where** limita la tabla “a lo largo”, pues elimina filas de la misma, mientras que la cláusula **select** es una selección “horizontal”.



The screenshot shows a window titled "Interactive SQL - mastsq1.gdb". The menu bar includes File, Edit, Query, Database, Transactions, Windows, and Help. The toolbar contains various icons for file operations and SQL execution. The main text area contains the following SQL query:

```
select company, min(saledate) minimo, max(saledate) maximo, sum(itemstotal) total
from customer, orders
where customer.custno = orders.custno
group by company
order by 4 desc
```

Below the query, the status bar shows "1: 1", "Client dialect 1", "Transaction is ACTIVE.", and "AutoDDL: ON". The results are displayed in a table with the following data:

COMPANY	MINIMO	MAXIMO	TOTAL
Sight Diver	12/04/88	22/12/94	261575.8
VIP Divers Club	01/05/88	10/01/95	193079.75
American SCUBA Supply	18/10/94	05/02/95	183094.4
Blue Sports	19/05/88	30/12/94	165245.45
Gold Coast Supply	13/03/89	13/07/92	132233
Kirk Enterprises	09/06/89	11/02/95	130362.95
The Depth Charge	18/05/88	20/12/94	126889.35
Divers of Corfu, Inc.	12/11/88	09/01/95	98278.6

The status bar at the bottom shows the file path: "C:\...\Archivos comunes\Borland Shared\Data\mastsq1.gdb".

## La condición de selección

La forma más simple de instrucción **select** es la que extrae el conjunto de filas de una sola tabla que satisfacen cierta condición. Por ejemplo:

```
select *
from Customer
where State = "HI"
```

Esta consulta simple debe devolver todos los datos de los clientes ubicados en Hawái. El asterisco que sigue a la cláusula **select** es una alternativa a listar todos los nombres de columna de la tabla que se encuentra en la cláusula **from**.

En este caso hemos utilizado una simple igualdad. La condición de búsqueda de la cláusula **where** admite los seis operadores de comparación (=, <>, <, >, <=, >=) y la creación de condiciones compuestas mediante el uso de los operadores lógicos **and**, **or** y **not**. La prioridad entre estos tres es la misma que en C. No hace falta encerrar las comparaciones entre paréntesis, porque incluso **not** se evalúa después de cualquier comparación:

```
select *
from Customer
where State = "HI"
and LastInvoiceDate > "1/1/1993"
```

Observe cómo la constante de fecha puede escribirse como si fuera una cadena de caracteres.

## Operadores de cadenas

Además de las comparaciones usuales, necesitamos operaciones más sofisticadas para trabajar con las cadenas de caracteres. Uno de los operadores admitidos por SQL estándar es el operador **like**, que nos permite averiguar si una cadena satisface o no cierto patrón de caracteres. El segundo operando de este operador es el patrón, una cadena de caracteres, dentro de la cual podemos incluir los siguientes comodines:

Carácter	Significado
%	Cero o más caracteres arbitrarios.
_ (subrayado)	Un carácter cualquiera.

No vaya a pensar que el comodín % funciona como el asterisco en los nombres de ficheros de MS-DOS; SQL es malo, pero no tanto. Después de colocar un asterisco en un nombre de fichero, MS-DOS ignora cualquier otro carácter que escribamos a continuación, mientras que **like** sí los tiene en cuenta. También es diferente el comportamiento del subrayado con respecto al signo de interrogación de DOS: en el intérprete de comandos de este sistema operativo significa cero o un caracteres, mientras que en SQL significa exactamente un carácter.

Expresión	Cadena aceptada	Cadena no aceptada
Customer <b>like</b> '% Ocean'	'Pacific Ocean'	'Ocean Paradise'
Fruta <b>like</b> 'Manzana_'	'Manzanas'	'Manzana'

También es posible aplicar funciones para extraer o modificar información de una cadena de caracteres; el repertorio de funciones disponibles depende del sistema de bases de datos con el que se trabaje. Por ejemplo, el intérprete SQL para dBase y Paradox acepta las funciones **upper**, **lower**, **trim** y **substring**. Esta última función tiene una sintaxis curiosa. Por ejemplo, para extraer las tres primeras letras de una cadena se utiliza la siguiente expresión:

```
select substring(Nombre from 1 for 3)
from Empleados
```

Si estamos trabajando con InterBase, podemos aumentar el repertorio de funciones utilizando *funciones definidas por el usuario*. En el capítulo 10 mostraremos cómo.

## El valor nulo: enfrentándonos a lo desconocido

La edad de una persona es un valor no negativo, casi siempre menor de 969 años, que es la edad a la que dicen que llegó Matusalén. Puede ser un entero igual a 1, 20, 40 ... o no conocerse. Se puede “resolver” este problema utilizando algún valor especial para indicar el valor desconocido, digamos -1. Claro, el valor especial escogido no debe formar parte del dominio posible de valores. Por ejemplo, en el archivo de Urgencias de un hospital americano, *John Doe* es un posible valor para los pacientes no identificados.

¿Y qué pasa si no podemos prescindir de valor alguno dentro del rango? Porque John Doe es un nombre raro, pero posible. ¿Y qué pasaría si se intentan operaciones con valores desconocidos? Por ejemplo, para representar un envío cuyo peso se desconoce se utiliza el valor -1, un peso claramente imposible excepto para entes como Kate Moss. Luego alguien pregunta a la base de datos cuál es el peso total de los envíos de un período dado. Si en ese período se realizaron dos envíos, uno de 25 kilogramos y otro de peso desconocido, la respuesta errónea será un peso total de 24 kilogramos. Es evidente que la respuesta debería ser, simplemente, “peso total desconocido”.

La solución de SQL es introducir un nuevo valor, **null**, que pertenece a cualquier dominio de datos, para representar la información desconocida. La regla principal que hay que conocer cuando se trata con valores nulos es que cualquier expresión, aparte de las expresiones lógicas, en la que uno de sus operandos tenga el valor nulo se evalúa automáticamente a nulo. Esto es: nulo más veinticinco vale nulo, ¿de acuerdo?

Cuando se trata de evaluar expresiones lógicas en las cuales uno de los operandos puede ser nulo las cosas se complican un poco, pues hay que utilizar una lógica de tres valores. De todos modos, las reglas son intuitivas. Una proposición falsa en conjunción con cualquier otra da lugar a una proposición falsa; una proposición verdadera en disyunción con cualquier otra da lugar a una proposición verdadera. La siguiente tabla resume las reglas del uso del valor nulo en expresiones lógicas:

AND	false	null	true
false	false	false	false
null	false	null	null
true	false	null	true

OR	false	null	true
false	false	null	true
null	null	null	true
true	true	true	true

Por último, si lo que desea es saber si el valor de un campo es nulo o no, debe utilizar el operador **is null**:

```
select *
from   Events
where  Event_Description is null
```

La negación de este operador es el operador **is not null**, con la negación en medio. Esta sintaxis no es la usual en lenguajes de programación, pero se suponía que SQL debía parecerse lo más posible al idioma inglés.

## Eliminación de duplicados

Normalmente, no solemos guardar filas duplicadas en una tabla, por razones obvias. Pero es bastante frecuente que el resultado de una consulta contenga filas duplicadas. El operador **distinct** se puede utilizar, en la cláusula **select**, para corregir esta situación. Por ejemplo, si queremos conocer en qué ciudades residen nuestros clientes podemos preguntar lo siguiente:

```
select City
from   Customer
```

Pero en este caso obtenemos 55 ciudades, algunas de ellas duplicadas. Para obtener las 47 diferentes ciudades de la base de datos tecleamos:

```
select distinct City
from   Customer
```

## Productos cartesianos y encuentros

Como para casi todas las cosas, la gran virtud del modelo relacional es, a la vez, su mayor debilidad. Me refiero a que cualquier modelo del “mundo real” puede representarse atomizándolo en relaciones: objetos matemáticos simples y predecibles, de fácil implementación en un ordenador (jaquellos ficheros *dbfs...*). Para reconstruir el modelo original, en cambio, necesitamos una operación conocida como “encuentro natural” (*natural join*).

Comencemos con algo más sencillo: con los productos cartesianos. Un producto cartesiano es una operación matemática entre conjuntos, la cual produce todas las parejas posibles de elementos, perteneciendo el primer elemento de la pareja al primer conjunto, y el segundo elemento de la pareja al segundo conjunto. Esta es la operación habitual que efectuamos mentalmente cuando nos ofrecen el menú en un

restaurante. Los dos conjuntos son el de los primeros platos y el de los segundos platos. Desde la ventana de la habitación donde escribo puedo ver el menú del mesón de la esquina:

Primer plato	Segundo plato
Macarrones a la boloñesa	Escalope a la milanese
Judías verdes con jamón	Pollo a la parrilla
Crema de champiñones	Chuletas de cordero

Si *PrimerPlato* y *SegundoPlato* fuesen tablas de una base de datos, la instrucción

```
select *  
from PrimerPlato, SegundoPlato
```

devolvería el siguiente conjunto de filas:

Primer plato	Segundo plato
Macarrones a la boloñesa	Escalope a la milanese
Macarrones a la boloñesa	Pollo a la parrilla
Macarrones a la boloñesa	Chuletas de cordero
Judías verdes con jamón	Escalope a la milanese
Judías verdes con jamón	Pollo a la parrilla
Judías verdes con jamón	Chuletas de cordero
Crema de champiñones	Escalope a la milanese
Crema de champiñones	Pollo a la parrilla
Crema de champiñones	Chuletas de cordero

Es fácil ver que, incluso con tablas pequeñas, el tamaño del resultado de un producto cartesiano es enorme. Si a este ejemplo “real” le añadimos el hecho también “real” de que el mismo mesón ofrece al menos tres tipos diferentes de postres, elegir nuestro menú significa seleccionar entre 27 posibilidades distintas. Por eso siempre pido café al terminar con el segundo plato.

Claro está, no todas las combinaciones de platos hacen una buena comida. Pero para eso tenemos la cláusula **where**: para eliminar aquellas combinaciones que no satisfacen ciertos criterios. ¿Volvemos al mundo de las facturas y órdenes de compra? En la base de datos *dbdemos*, la información sobre pedidos está en la tabla *orders*, mientras que la información sobre clientes se encuentra en *customer*. Queremos obtener la lista de clientes y sus totales por pedidos. Estupendo, pero los totales de pedidos están en la tabla *orders*, en el campo *ItemsTotal*, y en esta tabla sólo tenemos el código del cliente, en el campo *CustNo*. Los nombres de clientes se encuentran en el campo *Company* de la tabla *customer*, donde además volvemos a encontrar el código de cliente, *CustNo*. Así que partimos de un producto cartesiano entre las dos tablas, en el cual mostramos los nombres de clientes y los totales de pedidos:

```
select Company, ItemsTotal  
from Customer, Orders
```



Como tenemos unos 55 clientes y 205 pedidos, esta inocente consulta genera unas 11275 filas. La última vez que hice algo así fue siendo estudiante, en el centro de cálculos de mi universidad, para demostrarle a una profesora de Filosofía lo ocupado que estaba en ese momento.

En realidad, de esas 11275 filas nos sobran unas 11070, pues solamente son válidas las combinaciones en las que coinciden los códigos de cliente. La instrucción que necesitamos es:

```
select Company, ItemsTotal
from Customer, Orders
where Customer.CustNo = Orders.CustNo
```

Esto es un *encuentro natural*, un producto cartesiano restringido mediante la igualdad de los valores de dos columnas de las tablas básicas.

El ejemplo anterior ilustra también un punto importante: cuando queremos utilizar en la instrucción el nombre de los campos *ItemsTotal* y *Company* los escribimos tal y como son. Sin embargo, cuando utilizamos *CustNo* hay que aclarar a qué tabla original nos estamos refiriendo. Esta técnica se conoce como *calificación de campos*.

¿Un ejemplo más complejo? Suponga que desea añadir el nombre del empleado que recibió el pedido. La tabla *orders* tiene un campo *EmpNo* para el código del empleado, mientras que la información sobre empleados se encuentra en la tabla *employee*. La instrucción necesaria es una simple ampliación de la anterior:

```
select Company, ItemsTotal, FirstName || ' ' || LastName
from Customer, Orders, Employee
where Customer.CustNo = Orders.CustNo and
      Orders.EmpNo = Employee.EmpNo
```

Con 42 empleados en la base de datos de ejemplo y sin las restricciones de la cláusula **where**, habríamos obtenido un resultado de 473.550 filas.

## Ordenando los resultados

Una de las garantías de SQL es que podemos contar con que el compilador SQL genere automáticamente, o casi, el mejor código posible para evaluar las instrucciones. Esto también significa que, en el caso general, no podemos predecir con completa seguridad cuál será la estrategia utilizada para esta evaluación. Por ejemplo, en la instrucción anterior no sabemos si el compilador va a recorrer cada fila de la tabla de clientes para encontrar las filas correspondientes de pedidos o empleados, o si resultará más ventajoso recorrer las filas de pedidos para recuperar los nombres de clientes y empleados. Esto quiere decir, en particular, que no sabemos en qué orden se nos van a presentar las filas. En mi ordenador, utilizando Database Desktop sobre las tablas originales en formato Paradox, parece ser que se recorren primeramente las filas de la tabla de empleados.

¿Qué hacemos si el resultado debe ordenarse por el nombre de compañía? Para esto contamos con la cláusula **order by**, que se sitúa siempre al final de la consulta. En este caso, ordenamos por nombre de compañía el resultado con la instrucción:

```
select Company, ItemsTotal, FirstName || ' ' || LastName
from Customer, Orders, Employee
where Customer.CustNo = Orders.CustNo and
      Orders.EmpNo = Employee.EmpNo
order by Company
```

Aunque no es el caso de InterBase, algunos dialectos de SQL no permiten ordenar por una fila que no existe en el resultado de la instrucción. Si quisiéramos que los pedidos de cada compañía se ordenaran, a su vez, por la fecha de venta, habría que añadir el campo *SaleDate* al resultado y modificar la cláusula de ordenación del siguiente modo:

```
select Company, ItemsTotal, SaleDate, FirstName || ' ' || LastName
from Customer, Orders, Employee
where Customer.CustNo = Orders.CustNo and
      Orders.EmpNo = Employee.EmpNo
order by Company, SalesDate desc
```

Con la opción **desc** obtenemos los registros por orden descendente de fechas: primero los más recientes. Existe una opción **asc**, para cuando queremos enfatizar el sentido ascendente de una ordenación. Generalmente no se usa, pues es lo que asume el compilador.

Otra posibilidad de la cláusula **order by** es utilizar números en vez de nombres de columnas. Esto es necesario si se utilizan expresiones en la cláusula **select** y se quiere ordenar por dicha expresión. Por ejemplo:

```
select OrderNo, SalesDate, ItemsTotal - AmountPaid
from Orders
order by 3 desc
```

No se debe abusar de los números de columnas, pues esta técnica puede desaparecer en SQL-3 y hace menos legible la consulta. Una forma alternativa de ordenar por columnas calculadas es utilizar sinónimos para las columnas:

```
select OrderNo, SalesDate, ItemsTotal - AmountPaid as Diferencia
from Orders
order by Diferencia desc
```

## Sólo quiero los diez primeros...

El resultado de una sentencia **select** puede contener un número impredecible de registros, pero en muchos casos a usted solamente le interesa un puñado de filas representativas del resultado. Por ejemplo: queremos veinte clientes cualesquiera de Hawai. O, si calculamos las ventas por cada cliente y ordenamos el resultado de

acuerdo a este valor, puede que necesitemos sólo los cinco clientes que más han comprado.

Lamentablemente, no existe un estándar en SQL para expresar la condición anterior, y cada sistema que implementa algo parecido lo hace a su aire. Microsoft SQL Server ofrece una cláusula en la sentencia **select** para elegir un grupo inicial de filas:

```
select top 20 *
from Clientes
where Estado = 'HI'
```

Incluso nos deja recuperar un porcentaje del conjunto resultado:

```
select top 20 percent *
from Clientes
where Estado = 'HI'
```

En el caso anterior, no se trata de los veinte primeros clientes, sino de la quinta parte de la cantidad total de clientes.

DB2 pone a nuestra disposición un mecanismo similar:

```
select *
from Clientes
where Estado = 'HI'
fetch first 20 rows only
```

Por mucho que busquemos en los lenguajes de programación modernos, nunca veremos aparecer dos palabras reservadas consecutivamente. Aquí estamos viendo dos palabras claves juntas antes y después del número entero, como si estuvieran escoltándolo. Para colmo de la verbosidad, sepa usted que también podíamos haber escrito **row** en singular si quisiéramos únicamente la primera fila.

Oracle, por su parte, dispone de una pseudo columna, *rownum*, que puede utilizarse del siguiente modo:

```
select *
from Clientes
where Estado = 'HI' and
      rownum <= 20
```

El problema de *rownum* es que se evalúa antes de ordenar el conjunto resultado, y pierde así la mayor parte de su posible utilidad. Supongamos que la tabla de clientes tiene una columna con el importe total de las compras de ese cliente:

```
select * from Clientes
where rownum <= 20
order by ImporteTotal desc
```

La consulta anterior no devuelve los veinte “mejores” clientes, que es lo que sería realmente útil, sino que escoge veinte clientes y los ordena de acuerdo al total de sus

compras. ¿Quiénes son los clientes escogidos? Depende del algoritmo de optimización que el servidor ese día tenga a bien aplicar...

En contraste, la cláusula de limitación de filas de InterBase sí se aplica después de cualquier posible ordenación del resultado. Es cierto que sólo puede usarse a partir de InterBase 6.5. La consulta sobre los veinte mejores clientes se escribiría así en InterBase:

```
select *
from Clientes
order by ImporteTotal desc
rows 20
```

Si el cliente número 20 ha comprado lo mismo que el 21, y queremos ser justos e incluir los empates, podemos añadir una opción similar a la de SQL Server:

```
select *
from Clientes
order by ImporteTotal desc
rows 20 with ties
```

La sintaxis completa de la cláusula **rows** es la que sigue:

```
[rows n [to m] [by j] [percent] [with ties]
```

El significado de **percent** es el mismo que en SQL Server. La opción **to** cambia el significado del primer valor; la siguiente instrucción, después de ordenar los clientes, devuelve los que han caído en las posiciones que van de la décima a la vigésima:

```
select *
from Clientes
order by ImporteTotal desc
rows 10 to 20
```

Finalmente, la opción **by** permite saltos en la secuencia, que pueden medirse según un número absoluto de filas, o en un porcentaje del total de filas.

Las cláusulas de limitación del número de filas son sumamente importantes para el desarrollo de aplicaciones en Internet, y para la programación en general sobre redes de ancho de banda limitado.

## El uso de grupos

Ahora queremos sumar todos los totales de los pedidos para cada compañía, y ordenar el resultado por este total de forma descendente, para obtener una especie de *ranking* de las compañías según su volumen de compras. Esto es, hay que agrupar todas las filas de cada compañía y mostrar la suma de cierta columna dentro de cada uno de esos grupos.

Para producir grupos de filas en SQL se utiliza la cláusula **group by**. Cuando esta cláusula está presente en una consulta, va situada inmediatamente después de la cláusula **where**, o de la cláusula **from** si no se han efectuado restricciones. En nuestro caso, la instrucción con la cláusula de agrupamiento debe ser la siguiente:

```
select Company, sum(ItemsTotal)
from Customer, Orders
where Customer.CustNo = Orders.OrderNo
group by Company
order by 2 desc
```

Observe la forma en la cual se le ha aplicado la función **sum** a la columna *ItemsTotal*. Aunque pueda parecer engorroso el diseño de una consulta con grupos, hay una regla muy fácil que simplifica los razonamientos: en la cláusula **select** solamente pueden aparecer columnas especificadas en la cláusula **group by**, o funciones estadísticas aplicadas a cualquier otra expresión. *Company*, en este ejemplo, puede aparecer directamente porque es la columna por la cual se está agrupando. Si quisiéramos obtener además el nombre de la persona de contacto en la empresa, el campo *Contact* de la tabla *customer*, habría que agregar esta columna a la cláusula de agrupación:

```
select Company, Contact, sum(ItemsTotal)
from Customer, Orders
where Customer.CustNo = Orders.OrderNo
group by Company, Contact
order by 2 desc
```

En realidad, la adición de *Contact* es redundante, pues *Company* es única dentro de la tabla *customer*, pero eso lo sabemos nosotros, no el compilador de SQL. Sin embargo, InterBase puede optimizar mejor la consulta anterior si la planteamos del siguiente modo:

```
select Company, max(Contact), sum(ItemsTotal)
from Customer, Orders
where Customer.CustNo = Orders.OrderNo
group by Company
order by 2 desc
```

Si agrupamos por el nombre de compañía, dentro de cada grupo todas las personas de contacto serán iguales. Por lo tanto, nos da lo mismo mostrar el máximo o el mínimo de todos esos nombres, y así eliminamos una columna de la cláusula **group by**.

## Funciones de conjuntos

Existen cinco funciones de conjuntos en SQL, conocidas en inglés como *aggregate functions*. Estas funciones son:

Función	Significado
<b>count</b>	Cantidad de valores no nulos en el grupo
<b>min</b>	El valor mínimo de la columna dentro del grupo

Función	Significado
<b>max</b>	El valor máximo de la columna dentro del grupo
<b>sum</b>	La suma de los valores de la columna dentro del grupo
<b>avg</b>	El promedio de la columna dentro del grupo

Por supuesto, no toda función es aplicable a cualquier tipo de columna. Las sumas, por ejemplo, solamente valen para columnas de tipo numérico. Hay otros detalles curiosos relacionados con estas funciones, como que los valores nulos son ignorados por todas, o que se puede utilizar un asterisco como parámetro de la función **count**. En este último caso, se calcula el número de filas del grupo. Así que no apueste a que la siguiente consulta dé siempre dos valores idénticos, si es posible que la columna involucrada contenga valores nulos:

```
select avg(Columna), sum(Columna)/count(*)
from Tabla
```

En el ejemplo se muestra la posibilidad de utilizar funciones de conjuntos sin utilizar grupos. En esta situación se considera que toda la tabla constituye un único grupo. Es también posible utilizar el operador **distinct** como prefijo del argumento de una de estas funciones:

```
select Company, count(distinct EmpNo)
from Customer, Orders
where Customer.CustNo = Orders.CustNo
```

La consulta anterior muestra el número de empleados que han atendido los pedidos de cada compañía.

Oracle añade un par de funciones a las que ya hemos mencionado: **variance** y **stddev**, para la varianza y la desviación estándar. En realidad, estas funciones pueden calcularse a partir de las anteriores. Por ejemplo, la varianza de la columna *x* de la tabla *Tabla* puede obtenerse mediante la siguiente instrucción:

```
select (sum(x*x) - sum(x) * sum(x) / count(*)) / (count(*) - 1)
from Tabla
```

En cuanto a la desviación estándar, basta con extraer la raíz cuadrada de la varianza. Es cierto que InterBase no tiene una función predefinida para las raíces cuadradas, pero es fácil implementarla mediante una función de usuario.

## La cláusula having

Según lo que hemos explicado hasta el momento, en una instrucción **select** se evalúa primeramente la cláusula **from**, que indica qué tablas participan en la consulta, luego se eliminan las filas que no satisfacen la cláusula **where** y, si hay un **group by** por medio, se agrupan las filas resultantes. Hay una posibilidad adicional: después de agrupar se pueden descartar filas consolidadas de acuerdo a otra condición, esta vez

expresada en una cláusula **having**. En la parte **having** de la consulta solamente pueden aparecer columnas agrupadas o funciones estadísticas aplicadas al resto de las columnas. Por ejemplo:

```
select Company
from Customer, Orders
where Customer.CustNo = Orders.CustNo
group by Company
having count(*) > 1
```

La consulta anterior muestra las compañías que han realizado más de un pedido. Es importante darse cuenta de que no podemos modificar esta consulta para que nos muestre las compañías que *no* han realizado todavía pedidos.

Una regla importante de optimización: si en la cláusula **having** existen condiciones que implican solamente a las columnas mencionadas en la cláusula **group by**, estas condiciones deben moverse a la cláusula **where**. Por ejemplo, si queremos eliminar de la consulta utilizada como ejemplo a las compañías cuyo nombre termina con las siglas 'S.L.' debemos hacerlo en **where**, no en **group by**. ¿Para qué esperar a agrupar para eliminar filas que podían haberse descartado antes? Aunque muchos compiladores realizan esta optimización automáticamente, es mejor no fiarse.

## El uso de sinónimos para tablas

Es posible utilizar dos o más veces una misma tabla en una misma consulta. Si hacemos esto tendremos que utilizar *sinónimos* para distinguir entre los distintos usos de la tabla en cuestión. Esto será necesario al calificar los campos que utilicemos. Un sinónimo es simplemente un nombre que colocamos a continuación del nombre de una tabla en la cláusula **from**, y que en adelante se usa como sustituto del nombre de la tabla.

Por ejemplo, si quisiéramos averiguar si hemos introducido por error dos veces a la misma compañía en la tabla de clientes, pudiéramos utilizar la instrucción:

```
select distinct C1.Company
from Customer C1, Customer C2
where C1.CustNo < C2.CustNo and
      C1.Company = C2.Company
```

En esta consulta *C1* y *C2* se utilizan como sinónimos de la primera y segunda aparición, respectivamente, de la tabla *customer*. La lógica de la consulta es sencilla. Buscamos todos los pares que comparten el mismo nombre de compañía y eliminamos aquellos que tienen el mismo código de compañía. Pero en vez de utilizar una desigualdad en la comparación de códigos, utilizamos el operador “menor que”, para eliminar la aparición de pares dobles en el resultado previo a la aplicación del operador **distinct**. Estamos aprovechando, por supuesto, la unicidad del campo *CustNo*.

La siguiente consulta muestra otro caso en que una tabla aparece dos veces en una cláusula **from**. En esta ocasión, la base de datos es *iblocal*, el ejemplo InterBase que

viene con Delphi. Queremos mostrar los empleados junto con los jefes de sus departamentos:

```
select e2.full_name, e1.full_name
from employee e1, department d, employee e2
where d.dept_no = e1.dept_no and
      d.mngr_no = e2.emp_no and
      e1.emp_no <> e2.emp_no
order by 1, 2
```

Aquellos lectores que hayan trabajado en algún momento con lenguajes xBase reconocerán en los sinónimos SQL un mecanismo similar al de los “alias” de xBase. Delphi utiliza, además, los sinónimos de tablas en el intérprete de SQL local cuando el nombre de la tabla contiene espacios en blanco o el nombre de un directorio.

## Subconsultas: selección única

Si nos piden el total vendido a una compañía determinada, digamos a Ocean Paradise, podemos resolverlo ejecutando dos instrucciones diferentes. En la primera obtenemos el código de Ocean Paradise:

```
select Customer.CustNo
from Customer
where Customer.Company = 'Ocean Paradise'
```

El código buscado es, supongamos, 1510. Con este valor en la mano, ejecutamos la siguiente instrucción:

```
select sum(Orders.ItemsTotal)
from Orders
where Orders.CustNo = 1510
```

Incómodo, ¿no es cierto? La alternativa es utilizar la primera instrucción como una expresión dentro de la segunda, del siguiente modo:

```
select sum(Orders.ItemsTotal)
from Orders
where Orders.CustNo = (
    select Customer.CustNo
    from Customer
    where Customer.Company = 'Ocean Paradise')
```

Para que la subconsulta anterior pueda funcionar correctamente, estamos asumiendo que el conjunto de datos retornado por la subconsulta produce una sola fila. Esta es, realmente, una apuesta arriesgada. Puede fallar por dos motivos diferentes: puede que la subconsulta no devuelva ningún valor o puede que devuelva más de uno. Si no se devuelve ningún valor, se considera que la subconsulta devuelve el valor **null**. Si devuelve dos o más valores, el intérprete produce un error.



A este tipo de subconsulta que debe retornar un solo valor se le denomina *selección única*, en inglés, *singleton select*. Las selecciones únicas también pueden utilizarse con otros operadores de comparación, además de la igualdad. Así por ejemplo, la siguiente consulta retorna información sobre los empleados contratados después de Pedro Pérez:

```
select *
from Employee E1
where E1.HireDate > (
    select E2.HireDate
    from Employee E2
    where E2.FirstName = 'Pedro' and
          E2.LastName = 'Pérez')
```

Si está preguntándose acerca de la posibilidad de cambiar el orden de los operandos, ni lo sueñe. La sintaxis de SQL es muy rígida, y no permite este tipo de virtuosismos.

## Subconsultas: los operadores *in* y *exists*

En el ejemplo anterior garantizábamos la singularidad de la subconsulta gracias a la cláusula **where**, que especificaba una búsqueda sobre una clave única. Sin embargo, también se pueden aprovechar las situaciones en que una subconsulta devuelve un conjunto de valores. En este caso, el operador a utilizar cambia. Por ejemplo, si queremos los pedidos correspondientes a las compañías en cuyo nombre figura la palabra Ocean, podemos utilizar la instrucción:

```
select *
from Orders
where Orders.CustNo in (
    select Customer.CustNo
    from Customer
    where upper(Customer.Company) like '%OCEAN%')
```

El nuevo operador es el operador **in**, y la expresión es verdadera si el operando izquierdo se encuentra en la lista de valores retornada por la subconsulta. Esta consulta puede descomponerse en dos fases. Durante la primera fase se evalúa el segundo **select**:

```
select Customer.CustNo
from Customer
where upper(Customer.Company) like '%OCEAN%'
```

El resultado de esta consulta consiste en una serie de códigos: aquellos que corresponden a las compañías con Ocean en su nombre. Supongamos que estos códigos sean 1510 (Ocean Paradise) y 5515 (Ocean Adventures). Entonces puede ejecutarse la segunda fase de la consulta, con la siguiente instrucción, equivalente a la original:

```
select *
from Orders
where Orders.OrderNo in (1510, 5515)
```

Este otro ejemplo utiliza la negación del operador **in**. Si queremos las compañías que no nos han comprado nada, hay que utilizar la siguiente consulta:

```
select *
from Customer
where Customer.CustNo not in (
    select Orders.CustNo
    from Orders)
```

Otra forma de plantearse las consultas anteriores es utilizando el operador **exists**. Este operador se aplica a una subconsulta y devuelve verdadero en cuanto localiza una fila que satisface las condiciones de la instrucción **select**. El primer ejemplo de este epígrafe puede escribirse de este modo:

```
select *
from Orders
where exists (
    select *
    from Customer
    where upper(Customer.Company) like '%OCEAN%' and
           Orders.CustNo = Customer.CustNo)
```

Observe el asterisco en la cláusula **select** de la subconsulta. Como lo que nos interesa es saber si existen filas que satisfacen la expresión, nos da lo mismo qué valor se está retornando. El segundo ejemplo del operador **in** se convierte en lo siguiente al utilizar **exists**:

```
select *
from Customer
where not exists (
    select *
    from Orders
    where Orders.CustNo = Customer.CustNo)
```

## Subconsultas correlacionadas

Preste atención al siguiente detalle: la última subconsulta del epígrafe anterior tiene una referencia a una columna perteneciente a la tabla definida en la cláusula **from** más externa. Esto quiere decir que no podemos explicar el funcionamiento de la instrucción dividiéndola en dos fases, como con las selecciones únicas: la ejecución de la subconsulta y la simplificación de la instrucción externa. En este caso, para cada fila retornada por la cláusula **from** externa, la tabla *customer*, hay que volver a evaluar la subconsulta teniendo en cuenta los valores actuales: los de la columna *CustNo* de la tabla de clientes. A este tipo de subconsultas se les denomina, en el mundo de la programación SQL, *subconsultas correlacionadas*.

Si hay que mostrar los clientes que han pagado algún pedido contra reembolso (en inglés, *COD*, o *cash on delivery*), podemos realizar la siguiente consulta con una subselección correlacionada:

```

select *
from Customer
where 'COD' in (
    select distinct PaymentMethod
    from Orders
    where Orders.CustNo = Customer.CustNo)

```

En esta instrucción, para cada cliente se evalúan los pedidos realizados por el mismo, y se muestra el cliente solamente si dentro del conjunto de métodos de pago está la cadena 'COD'. El operador **distinct** de la subconsulta es redundante, pero nos ayuda a entenderla mejor.

Otra subconsulta correlacionada: queremos los clientes que no han comprado nada aún. Ya vimos como hacerlo utilizando el operador **not in** ó el operador **not exists**. Una alternativa es la siguiente:

```

select *
from Customer
where 0 = (
    select count(*)
    from Orders
    where Orders.CustNo = Customer.CustNo)

```

Sin embargo, utilizando SQL Local esta consulta es más lenta que las otras dos soluciones. La mayor importancia del concepto de subconsulta correlacionada tiene que ver con el hecho de que algunos sistemas de bases de datos limitan las actualizaciones a vistas definidas con instrucciones que contienen subconsultas de este tipo.

## Equivalencias de subconsultas

En realidad, las subconsultas son un método para aumentar la expresividad del lenguaje SQL, pero no son imprescindibles. Con esto quiero decir que muchas consultas se formulan de modo más natural si se utilizan subconsultas, pero que existen otras consultas equivalentes que no hacen uso de este recurso. La importancia de esta equivalencia reside en que el intérprete de SQL Local de Delphi 1 no permitía subconsultas. También sucede que la versión 4 de InterBase no optimizaba correctamente ciertas subconsultas.

Un problema relacionado es que, aunque un buen compilador de SQL debe poder identificar las equivalencias y evaluar la consulta de la forma más eficiente, en la práctica el utilizar ciertas construcciones sintácticas puede dar mejor resultado que utilizar otras equivalentes, de acuerdo al compilador que empleemos.

Veamos algunas equivalencias. Teníamos una consulta, en el epígrafe sobre selecciones únicas, que mostraba el total de compras de Ocean Paradise:

```

select sum(Orders.ItemsTotal)
from Orders
where Orders.CustNo = (

```

```

select Customer.CustNo
from Customer
where Customer.Company = 'Ocean Paradise')

```

Esta consulta es equivalente a la siguiente:

```

select sum(ItemsTotal)
from Customer, Orders
where Customer.Company = 'Ocean Paradise' and
      Customer.CustNo = Orders.OrderNo

```

Aquella otra consulta que mostraba los pedidos de las compañías en cuyo nombre figuraba la palabra “Ocean”:

```

select *
from Orders
where Orders.CustNo in (
      select Customer.CustNo
      from Customer
      where upper(Customer.Company) like '%OCEAN%')

```

es equivalente a esta otra:

```

select *
from Customer, Orders
where upper(Customer.Company) like '%OCEAN%' and
      Customer.CustNo = Orders.CustNo

```

Para esta consulta en particular, ya habíamos visto una consulta equivalente que hacía uso del operador **exists**; en este caso, es realmente más difícil de entender la consulta con **exists** que su equivalente sin subconsultas.

La consulta correlacionada que buscaba los clientes que en algún pedido habían pagado contra reembolso:

```

select *
from Customer
where 'COD' in (
      select distinct PaymentMethod
      from Orders
      where Orders.CustNo = Customer.CustNo)

```

puede escribirse, en primer lugar, mediante una subconsulta no correlacionada:

```

select *
from Customer
where Customer.CustNo in (
      select Orders.CustNo
      from Orders
      where PaymentMethod = 'COD')

```

pero también se puede expresar en forma “plana”:

```

select distinct Customer.CustNo, Customer.Company
from Customer, Orders
where Customer.CustNo = Orders.CustNo and
Orders.PaymentMethod = 'COD'

```

Por el contrario, las consultas que utilizan el operador **not in** y, por lo tanto sus equivalentes con **not exists**, no tienen equivalente plano, con lo que sabemos hasta el momento. Para poder aplanarlas hay que utilizar *encuentros externos*.

## Encuentros externos

El problema de los encuentros naturales es que cuando relacionamos dos tablas, digamos *customer* y *orders*, solamente mostramos las filas que tienen una columna en común. No hay forma de mostrar los clientes que *no* tienen un pedido con su código ... y solamente esos. En realidad, se puede utilizar la operación de *diferencia* entre conjuntos para lograr este objetivo, como veremos en breve. Se pueden evaluar todos los clientes, y a ese conjunto restarle el de los clientes que sí tienen pedidos. Pero esta operación, por lo general, se implementa de forma menos eficiente que la alternativa que mostraremos a continuación.

¿Cómo funciona un encuentro natural? Una posible implementación consistiría en recorrer mediante un bucle la primera tabla, supongamos que sea *customer*. Para cada fila de esa tabla tomaríamos su columna *CustNo* y buscaríamos, posiblemente con un índice, las filas correspondientes de *orders* que contengan ese mismo valor en la columna del mismo nombre. ¿Qué pasa si no hay ninguna fila en *orders* que satisfaga esta condición? Si se trata de un encuentro natural, común y corriente, no se mostrarán los datos de ese cliente. Pero si se trata de la extensión de esta operación, conocida como *encuentro externo* (*outer join*), se muestra aunque sea una vez la fila correspondiente al cliente. Un encuentro muestra, sin embargo, pares de filas, ¿qué valores podemos esperar en la fila de pedidos? En ese caso, se considera que todas las columnas de la tabla de pedidos tienen valores nulos. Si tuviéramos estas dos tablas:

Customers		Orders	
<i>CustNo</i>	<i>Company</i>	<i>OrderNo</i>	<i>CustNo</i>
1510	Ocean Paradise	1025	1510
1666	Marteens' Diving Academy	1139	1510

el resultado de un encuentro externo como el que hemos descrito, de acuerdo a la columna *CustNo*, sería el siguiente:

Customer.CustNo	Company	OrderNo	Orders.CustNo
1510	Ocean Paradise	1025	1510
1510	Ocean Paradise	1139	1510
1666	Marteens' Diving Academy	null	null

Con este resultado en la mano, es fácil descubrir quién es el tacaño que no nos ha pedido nada todavía, dejando solamente las filas que tengan valores nulos para alguna de las columnas de la segunda tabla.

Este encuentro externo que hemos explicado es, en realidad, un encuentro externo *por la izquierda*, pues la primera tabla tendrá todas sus filas en el resultado final, aunque no exista fila correspondiente en la segunda. Naturalmente, también existe un encuentro externo *por la derecha* y un encuentro externo *simétrico*.

El problema de este tipo de operaciones es que su inclusión en SQL fue bastante tardía. Esto trajo como consecuencia que distintos fabricantes utilizaran sintaxis propias para la operación. En el estándar ANSI para SQL del año 87 no hay referencias a esta instrucción, pero sí la hay en el estándar del 92. Utilizando esta sintaxis, que es la permitida por el SQL local, la consulta que queremos se escribe del siguiente modo:

```
select *
from   Customer left outer join Orders
      on (Customer.CustNo = Orders.CustNo)
where  Orders.OrderNo is null
```

Observe que se ha extendido la sintaxis de la cláusula **from**.

En Oracle, el encuentro externo por la izquierda se escribe así:

```
select *
from   Customer, Orders
where  Customer.CustNo (+) = Orders.CustNo and
      Orders.OrderNo is null
```

La mayoría de las aplicaciones de los encuentros externos están relacionadas con la generación de informes. Pongamos por caso que tenemos una tabla de clientes y una tabla relacionada de teléfonos, asumiendo que un cliente puede tener asociado un número de teléfono, varios o ninguno. Si queremos listar los clientes y sus números de teléfono y utilizamos un encuentro natural, aquellos clientes de los que desconocemos el teléfono no aparecerán en el listado. Es necesario entonces recurrir a un encuentro externo por la izquierda.

## La curiosa sintaxis del encuentro interno

De la misma manera en que hay una sintaxis especial para los encuentros externos, existe una forma equivalente de expresar el encuentro “normal”, o interno:

```
select Company, OrderNo
from   Customer inner join Orders
      on (Customer.CustNo = Orders.CustNo)
```

Fácilmente se comprende que la consulta anterior es equivalente a:

```

select Company, OrderNo
from Customer, Orders
where Customer.CustNo = Orders.CustNo

```

¿Por qué nos complican la vida con la sintaxis especial los señores del comité ANSI? El propósito de las cláusulas **inner** y **outer join** es la definición de “expresiones de tablas” limitadas. Hasta el momento, nuestra sintaxis solamente permitía nombres de tabla en la cláusula **from**, pero gracias a las nuevas cláusulas se pueden escribir consultas más complejas.

```

select Company, OrderNo, max(Discount)
from Customer C
      left outer join
      (Orders O inner join Items I
       on (O.OrderNo = I.OrderNo))
      on C.CustNo = O.CustNo
group by Company, OrderNo

```

Si hubiéramos omitido los paréntesis en la cláusula **from** de la consulta anterior hubiéramos perdido las filas de clientes que no han realizado pedidos. También pueden combinarse entre sí varios encuentros externos.

Mi opinión sobre la sintaxis del **inner join** ha variado en los últimos años. Antes, consideraba que era un incordio. Pero me he dado cuenta de que un encuentro escrito con esta sintaxis es más legible, porque aísla nítidamente las condiciones del encuentro de los filtros posteriores que podamos incluir en la cláusula **where**. Además, le conviene familiarizarse con ella, pues varias herramientas de Delphi (SQL Builder, el editor de consultas de Decision Cube) insistirán tozudamente en modificar sus encuentros “naturales” para que usen esta notación.

## Las instrucciones de actualización

Son tres las instrucciones de actualización de datos reconocidas en SQL: **delete**, **update** e **insert**. Estas instrucciones tienen una sintaxis relativamente simple y están limitadas, en el sentido de que solamente cambian datos en una tabla a la vez. La más sencilla de las tres es **delete**, la instrucción de borrado:

```

delete from Tabla
where Condición

```

La instrucción elimina de la tabla indicada todas las filas que se ajustan a cierta condición. En dependencia del sistema de bases de datos de que se trate, la condición de la cláusula **where** debe cumplir ciertas restricciones. Por ejemplo, aunque InterBase admite la presencia de subconsultas en la condición de selección, otros sistemas no lo permiten.

La segunda instrucción, **update**, nos sirve para modificar las filas de una tabla que satisfacen cierta condición:

```

update Tabla
set Columna = Valor [, Columna = Valor ...]
where Condición

```

Al igual que sucede con la instrucción **delete**, las posibilidades de esta instrucción dependen del sistema que la implementa. InterBase, en particular, permite actualizar columnas con valores extraídos de otras tablas; para esto utilizamos subconsultas en la cláusula **set**:

```

update Customer
set LastInvoiceDate =
    (select max(SaleDate) from Orders
     where Orders.CustNo = Customer.CustNo)

```

Por último, tenemos la instrucción **insert**, de la cual tenemos dos variantes. La primera permite insertar un solo registro, con valores constantes:

```

insert into Tabla [ (Columnas) ]
values (Valores)

```

La lista de columnas es opcional; si se omite, se asume que la instrucción utiliza todas las columnas en orden de definición. En cualquier caso, el número de columnas empleado debe coincidir con el número de valores. El objetivo de todo esto es que si no se indica un valor para alguna columna, el valor de la columna en el nuevo registro se inicializa con el valor definido por omisión; recuerde que si en la definición de la tabla no se ha indicado nada, el valor por omisión es **null**:

```

insert into Employee(EmpNo, LastName, FirstName)
values (666, "Bonaparte", "Napoleón")
/* El resto de las columnas, incluida la del salario, son nulas */

```

La segunda variante de **insert** permite utilizar como fuente de valores una expresión **select**:

```

insert into Tabla [ (Columnas) ]
InstrucciónSelect

```

Se utiliza con frecuencia para copiar datos de una tabla a otra:

```

insert into Resumen(Empresa, TotalVentas)
select Company, sum(ItemsTotal)
from Customer, Orders
where Customer.CustNo = Orders.CustNo
group by Company

```

## La semántica de la instrucción **update**

Según el estándar ANSI para SQL, durante la ejecución de la cláusula **set** de una instrucción **update** en la que se modifican varias columnas, se evalúan primero todos los valores a asignar y se guardan en variables temporales, y luego se realizan todas



las asignaciones. De este modo, el orden de las asignaciones no afecta el resultado final.

Supongamos que tenemos una tabla llamada *Tabla*, con dos columnas del mismo tipo: *A* y *B*. Para intercambiar los valores de ambas columnas puede utilizarse la siguiente instrucción:

```
update Tabla
set    A = B, B = A
```

Y realmente, así suceden las cosas en Oracle y SQL Server. ... pero no en InterBase. Si la instrucción anterior se ejecuta en InterBase, al final de la misma los valores de la columna *B* se han copiado en la columna *A*, pero no al contrario.

Este es un detalle a tener en cuenta por los programadores que intentan transportar una aplicación de una plataforma a otra. Imagino que en algún nivel del estándar se deje a la decisión del fabricante qué semántica implementar en la cláusula **set**. Y es que el ANSI SQL es demasiado permisivo para mi gusto.

## Actualizaciones parciales

En compensación, InterBase 6.5 extiende la sintaxis de **update** y **delete** con una cláusula muy útil, porque permite actualizar o eliminar los primeros registros de un conjunto resultado:

```
delete from Empleados
order by HireDate desc
rows 5
```

La instrucción anterior elimina los últimos cinco empleados que hemos contratado. Observe que la selección de los registros va ligada al uso de una cláusula de ordenación. No se trata de algo obligatorio, pero es el criterio de ordenación el que imparte un significado práctico a **rows**. Recuerde que en SQL estándar no se admite la cláusula **order by** en las sentencias de modificación y borrado.

También podrá utilizar **rows** en un **update**, a partir de InterBase 6.5.

## Vistas

Se puede aprovechar una instrucción **select** de forma tal que el conjunto de datos que define se pueda utilizar “casi” como una tabla real. Para esto, debemos definir una *vista*. La instrucción necesaria tiene la siguiente sintaxis:

```
create view NombreVista[Columnas]
as InstrucciónSelect
[with check option]
```

Por ejemplo, podemos crear una vista para trabajar con los clientes que viven en Hawaii:

```
create view Hawaianos as
select *
from Customer
where State = 'HI'
```

A partir del momento en que se ejecuta esta instrucción, el usuario de la base de datos se encuentra con una nueva tabla, *Hawaianos*, con la cual puede realizar las mismas operaciones que realizaba con las tablas “físicas”. Puede utilizar la nueva tabla en una instrucción **select**:

```
select *
from Hawaianos
where LastInvoiceDate >=
(select avg(LastInvoiceDate) from Customer)
```

En esta vista en particular, puede también eliminar insertar o actualizar registros:

```
delete from Hawaianos
where LastInvoiceDate is null;

insert into Hawaianos(CustNo, Company, State)
values (8888, 'Ian Marteens' Diving Academy, 'HI')
```

No todas las vistas permiten operaciones de actualización. Las condiciones que deben cumplir para ser actualizables, además, dependen del sistema de bases de datos en que se definan. Los sistemas más restrictivos exigen que la instrucción **select** tenga una sola tabla en la cláusula **from**, que no contenga consultas anidadas y que no haga uso de operadores tales como **group by**, **distinct**, etc.

### EXTENSIONES DE INTERBASE

Cuando estudiemos los *triggers*, veremos que InterBase permite actualizar cualquier vista, aunque teóricamente no sea actualizable, si creamos los *triggers* apropiados. Esta característica también existe en Oracle, y recientemente se ha incorporado a SQL Server 2000.

Cuando una vista permite actualizaciones se nos plantea el problema de qué hacer si se inserta un registro que no pertenece lógicamente a la vista. Por ejemplo, ¿podemos insertar dentro de la vista *Hawaianos* una empresa con sede social en la ciudad costera de Cantalapiedra<sup>10</sup>? Si lo permitiésemos, el registro recién insertado “desaparecería” inmediatamente de la vista (aunque no de la tabla base, *Customer*). El mismo conflicto se produciría al actualizar la columna *State* de un hawaiano.

---

<sup>10</sup> Cuando escribí la primera versión de este libro, no sabía que había un Cantalapiedra en España; pensé que nombre tan improbable era invento mío. Mis disculpas a los cantalapiedrenses, pues me parece que viven en ciudad sin costas.

Para controlar este comportamiento, SQL define la cláusula **with check option**. Si se especifica esta opción, no se permiten inserciones ni modificaciones que violen la condición de selección impuesta a la vista; si intentamos una operación tal, se produce un error de ejecución. Por el contrario, si no se incluye la opción en la definición de la vista, estas operaciones se permiten, pero nos encontraremos con situaciones como las descritas, en que un registro recién insertado o modificado desaparece misteriosamente por no pertenecer a la vista.



# Procedimientos almacenados y triggers

CON ESTE CAPÍTULO COMPLETAMOS la presentación de los sublenguajes de SQL, mostrando el lenguaje de definición de procedimientos de InterBase. Desgraciadamente, los lenguajes de procedimientos de los distintos sistemas de bases de datos difieren entre ellos, al no existir todavía un estándar al respecto. De los dialectos existentes, he elegido nuevamente InterBase por dos razones. La primera, y fundamental, es que es el sistema SQL que *usted* tiene a mano. La segunda es que el dialecto de InterBase para procedimientos es el que más se asemeja al propuesto en el borrador del estándar SQL-3. De cualquier manera, las diferencias entre dialectos no son demasiadas, y no le costará mucho trabajo entender el lenguaje de procedimientos de cualquier otro sistema de bases de datos.

## ¿Para qué usar procedimientos almacenados?

Un *procedimiento almacenado* (*stored procedure*) es, sencillamente, un algoritmo cuya definición reside en la base de datos, y que es ejecutado por el servidor del sistema. Aunque SQL-3 define formalmente un lenguaje de programación para procedimientos almacenados, cada uno de los sistemas de bases de datos importantes a nivel comercial implementa su propio lenguaje para estos recursos. InterBase ofrece un dialecto parecido a la propuesta de SQL-3; Oracle tiene un lenguaje llamado PL-SQL; Microsoft SQL Server ofrece el denominado Transact-SQL. No obstante, las diferencias entre estos lenguajes son mínimas, principalmente sintácticas, siendo casi idénticas las capacidades expresivas.

El uso de procedimientos almacenados ofrece las siguientes ventajas:

- Los procedimientos almacenados ayudan a mantener la consistencia de la base de datos.

Las instrucciones básicas de actualización, **update**, **insert** y **delete**, pueden combinarse arbitrariamente si dejamos que el usuario tenga acceso ilimitado a las mismas. No toda combinación de actualizaciones cumplirá con las reglas de consistencia de la base de datos. Hemos visto que algunas de estas

reglas se pueden expresar declarativamente durante la definición del esquema relacional. El mejor ejemplo son las restricciones de integridad referencial. Pero, ¿cómo expresar declarativamente que para cada artículo presente en un pedido, debe existir un registro correspondiente en la tabla de movimientos de un almacén? Una posible solución es prohibir el uso directo de las instrucciones de actualización, revocando permisos de acceso al público, y permitir la modificación de datos solamente a partir de procedimientos almacenados.

- Los procedimientos almacenados permiten superar las limitaciones del lenguaje de consultas.

SQL no es un lenguaje completo. Un problema típico en que falla es en la definición de *clausuras relacionales*. Tomemos como ejemplo una tabla con dos columnas: *Objeto* y *Parte*. Esta tabla contiene pares como los siguientes:

Objeto	Parte
Cuerpo humano	Cabeza
Cuerpo humano	Tronco
Cabeza	Ojos
Cabeza	Boca
Boca	Dientes

¿Puede el lector indicar una consulta que liste todas las partes incluidas en la cabeza? Lo que falla es la posibilidad de expresar algoritmos recursivos. Para resolver esta situación, los procedimientos almacenados pueden implementarse de forma tal que devuelvan conjuntos de datos, en vez de valores escalares. En el cuerpo de estos procedimientos se pueden realizar, entonces, llamadas recursivas.

- Los procedimientos almacenados pueden reducir el tráfico en la red.

Un procedimiento almacenado se ejecuta en el servidor, que es precisamente donde se encuentran los datos. Por lo tanto, no tenemos que explorar una tabla de arriba a abajo desde un ordenador cliente para extraer el promedio de ventas por empleado durante el mes pasado. Además, por regla general el servidor es una máquina más potente que las estaciones de trabajo, por lo que puede que ahorremos tiempo de ejecución para una petición de información. No conviene, sin embargo, abusar de esta última posibilidad, porque una de las ventajas de una red consiste en distribuir el tiempo de procesador.

- Con los procedimientos almacenados se puede ahorrar tiempo de desarrollo.

Siempre que existe una información, a alguien se le puede ocurrir un nuevo modo de aprovecharla. En un entorno cliente/servidor es típico que varias aplicaciones diferentes trabajen con las mismas bases de datos. Si centraliza-

mos en la propia base de datos la imposición de las reglas de consistencia, no tendremos que volverlas a programar de una aplicación a otra. Además, evitamos los riesgos de una mala codificación de estas reglas, con la consiguiente pérdida de consistencia.

Como todas las cosas de esta vida, los procedimientos almacenados también tienen sus inconvenientes. Ya he mencionado uno de ellos: si se centraliza todo el tratamiento de las reglas de consistencia en el servidor, corremos el riesgo de saturar los procesadores del mismo. El otro inconveniente es la poca portabilidad de las definiciones de procedimientos almacenados entre distintos sistemas de bases de datos. Si hemos desarrollado procedimientos almacenados en InterBase y queremos migrar nuestra base de datos a Oracle (o viceversa), estaremos obligados a partir “casi” de cero; algo se puede aprovechar, de todos modos.

## Cómo se utiliza un procedimiento almacenado

Un procedimiento almacenado puede utilizarse desde una aplicación cliente, desarrollada en cualquier lenguaje de programación que pueda acceder a la interfaz de programación de la base de datos, o desde las propias utilidades interactivas del sistema. En la VCL tenemos el componente *TStoredProc*, diseñado para la ejecución de procedimientos; IB Express ofrece *TIBStoredProc* y DB Express, *TSQLStoredProc*. En un capítulo posterior veremos cómo suministrar parámetros, ejecutar procedimientos y recibir información utilizando estos componentes.

En el caso de InterBase, también es posible ejecutar un procedimiento almacenado directamente desde la aplicación *Windows ISQL*, mediante la siguiente instrucción:

```
execute procedure NombreProcedimiento [ListaParámetros];
```

La misma instrucción puede utilizarse en el lenguaje de definición de procedimientos y *triggers* para llamar a un procedimiento dentro de la definición de otro. Es posible también definir procedimientos recursivos. InterBase permite hasta un máximo de 1000 llamadas recursivas por procedimiento.

## El carácter de terminación

Los procedimientos almacenados de InterBase deben necesariamente escribirse en un fichero *script* de SQL. Más tarde, este fichero debe ser ejecutado desde la utilidad *Windows ISQL* para que los procedimientos sean incorporados a la base de datos. Hemos visto las reglas generales del uso de *scripts* en InterBase en el capítulo de introducción a SQL. Ahora tenemos que estudiar una característica de estos *scripts* que anteriormente hemos tratado superficialmente: el carácter de terminación.

Por regla general, cada instrucción presente en un *script* es leída y ejecutada de forma individual y secuencial. Esto quiere decir que el intérprete de *scripts* lee del fichero

hasta que detecta el fin de instrucción, ejecuta la instrucción recuperada, y sigue así hasta llegar al final del mismo. El problema es que este proceso de extracción de instrucciones independientes se basa en la detección de un carácter especial de terminación. Por omisión, este carácter es el punto y coma; el lector habrá observado que todos los ejemplos de instrucciones SQL que deben colocarse en *scripts* han sido, hasta el momento, terminados con este carácter.

Ahora bien, al tratar con el lenguaje de procedimientos y *triggers* encontraremos instrucciones y cláusulas que deben terminar con puntos y comas. Si el intérprete de *scripts* tropieza con uno de estos puntos y comas pensará que se encuentra frente al fin de la instrucción, e intentará ejecutar lo que ha leído hasta el momento; casi siempre, una instrucción incompleta. Por lo tanto, debemos cambiar el carácter de terminación de *Windows ISQL* cuando estamos definiendo *triggers* o procedimientos almacenados. La instrucción que nos ayuda para esto es la siguiente:

```
set term Terminador
```

Como carácter de terminación podemos escoger cualquier carácter o combinación de los mismos lo suficientemente rara como para que no aparezca dentro de una instrucción del lenguaje de procedimientos. Por ejemplo, podemos utilizar el acento circunflejo:

```
set term ^;
```

Observe cómo la instrucción que cambia el carácter de terminación debe terminar ella misma con el carácter antiguo. Al finalizar la creación de todos los procedimientos que necesitamos, debemos restaurar el antiguo carácter de terminación:

```
set term ;^
```

En lo sucesivo asumiremos que el carácter de terminación ha sido cambiado al acento circunflejo.

## Procedimientos almacenados en InterBase

La sintaxis para la creación de un procedimiento almacenado en InterBase es la siguiente:

```
create procedure Nombre
[ ( ParámetrosDeEntrada ) ]
[ returns ( ParámetrosDeSalida ) ]
as CuerpoDeProcedimiento
```

Las cláusulas *ParámetrosDeEntrada* y *ParámetrosDeSalida* representan listas de declaraciones de parámetros. Los parámetros de salida pueden ser más de uno; esto significa que el procedimiento almacenado que retorna valores no se utiliza como si fuese



una función de un lenguaje de programación tradicional. El siguiente es un ejemplo de cabecera de procedimiento:

```
create procedure TotalPiezas(PiezaPrincipal char(15))
    returns (Total integer)
as
/* ... Aquí va el cuerpo ... */
```

El cuerpo del procedimiento, a su vez, se divide en dos secciones, siendo opcional la primera de ellas: la sección de declaración de variables locales, y una instrucción compuesta, **begin...end**, que agrupa las instrucciones del procedimiento. Las variables se declaran en este verboso estilo, *á la 1970*:

```
declare variable V1 integer;
declare variable V2 char(50);
```

Estas son las instrucciones permitidas por los procedimientos almacenados de InterBase:

- Asignaciones:

```
Variable = Expresión
```

Las variables pueden ser las declaradas en el propio procedimiento, parámetros de entrada o parámetros de salida.

- Llamadas a procedimientos:

```
execute procedure NombreProc [ParsEntrada]
    [returning_values ParsSalida]
```

No se admiten expresiones en los parámetros de entrada; mucho menos en los de salida.

- Condicionales:

```
if (Condición) then Instrucción [else Instrucción]
```

- Bucles controlados por condiciones:

```
while (Condición) do Instrucción
```

- Instrucciones SQL:

Cualquier instrucción de manipulación, **insert**, **update** ó **delete**, puede incluirse en un procedimiento almacenado. Estas instrucciones pueden utilizar variables locales y parámetros, siempre que estas variables estén precedidas de dos puntos, para distinguirlas de los nombres de columnas. Por ejemplo,

si *Mínimo* y *Aumento* son variables o parámetros, puede ejecutarse la siguiente instrucción:

```
update Empleados
set Salario = Salario * :Aumento
where Salario < :Mínimo;
```

Se permite el uso directo de instrucciones **select** si devuelven una sola fila; para consultas más generales se utiliza la instrucción **for** que veremos dentro de poco. Estas selecciones únicas van acompañadas por una cláusula **into** para transferir valores a variables o parámetros:

```
select Empresa
from Clientes
where Código = 1984
into :NombreEmpresa;
```

- Iteración sobre consultas:

```
for InstrucciónSelect into Variables do Instrucción
```

Esta instrucción recorre el conjunto de filas definido por la instrucción **select**. Para cada fila, transfiere los valores a las variables de la cláusula **into**, de forma similar a lo que sucede con las selecciones únicas, y ejecuta entonces la instrucción de la sección **do**.

- Lanzamiento de excepciones:

```
exception NombreDeExcepción
```

Similar a la instrucción **raise** de Delphi.

- Captura de excepciones:

```
when ListaDeErrores do Instrucción
```

Similar a la cláusula **except** de la instrucción **try/except** de Delphi. Los errores capturados pueden ser excepciones propiamente dichas o errores reportados con la variable `SQLCODE`. Estos últimos errores se producen al ejecutarse instrucciones SQL. Las instrucciones **when** deben colocarse al final de los procedimientos.

- Instrucciones de control:

```
exit;
suspend;
```

La instrucción **exit** termina la ejecución del procedimiento actual, y es similar al procedimiento *Exit* de Delphi. Por su parte, **suspend** se utiliza en pro-

cedimientos que devuelven un conjunto de filas para retornar valores a la rutina que llama a este procedimiento. Con esta última instrucción, se interrumpe temporalmente el procedimiento, hasta que la rutina que lo llama haya procesado los valores retornados.

- Instrucciones compuestas:

```
begin ListaDeInstrucciones end
```

La sintaxis de los procedimientos de InterBase es similar a la de Pascal. A diferencia de este último lenguaje, la palabra **end** no puede tener un punto y coma a continuación.

Mostraré ahora un par de procedimientos sencillos, que ejemplifiquen el uso de estas instrucciones. El siguiente procedimiento, basado en las tablas definidas en el capítulo sobre DDL, sirve para recalcular la suma total de un pedido, si se suministra el número de pedido correspondiente:

```
create procedure RecalcularTotal(NumPed int) as
declare variable Total integer;
begin
    select sum(Cantidad * PVP * (100 - Descuento) / 100)
    from Detalles, Articulos
    where Detalles.RefArticulo = Articulos.Codigo
        and Detalles.RefPedido = :NumPed
    into :Total;
    if (Total is null) then
        Total = 0;
    update Pedidos
    set Total = :Total
    where Numero = :NumPed;
end ^
```

El procedimiento consiste básicamente en una instrucción **select** que calcula la suma de los totales de todas las líneas de detalles asociadas al pedido; esta instrucción necesita mezclar datos provenientes de las líneas de detalles y de la tabla de artículos. Si el valor total es nulo, se cambia a cero. Esto puede suceder si el pedido no tiene líneas de detalles; en este caso, la instrucción **select** retorna el valor nulo. Finalmente, se localiza el pedido indicado y se le actualiza el valor a la columna *Total*, utilizando el valor depositado en la variable local del mismo nombre.

El procedimiento que definimos a continuación se basa en el anterior, y permite recalcular los totales de todas las filas almacenadas en la tabla de pedidos; de este modo ilustramos el uso de las instrucciones **for select ... do** y **execute procedure**:

```
create procedure RecalcularPedidos as
declare variable Pedido integer;
begin
    for select Numero from Pedidos into :Pedido do
        execute procedure RecalcularTotal :Pedido;
end ^
```

## Procedimientos que devuelven un conjunto de datos

Antes he mencionado la posibilidad de superar las restricciones de las expresiones **select** del modelo relacional mediante el uso de procedimientos almacenados. Un procedimiento puede diseñarse de modo que devuelva un conjunto de filas; para esto hay que utilizar la instrucción **suspend**, que transfiere el control temporalmente a la rutina que llama al procedimiento, para que ésta pueda hacer algo con los valores asignados a los parámetros de salida. Esta técnica es poco habitual en los lenguajes de programación más extendidos; si quiere encontrar algo parecido, puede desenterrar los *iteradores* del lenguaje CLU, diseñado por Barbara Liskov a mediados de los setenta.

Supongamos que necesitamos obtener la lista de los primeros veinte, treinta o mil cuatrocientos números primos. Comencemos por algo fácil, con la función que analiza un número y dice si es primo o no:

```
create procedure EsPrimo(Numero integer)
    returns (Respuesta integer) as
declare variable I integer;
begin
    I = 2;
    while (I < Numero) do
        begin
            if (cast((Numero / I) as integer) * I = Numero) then
                begin
                    Respuesta = 0;
                    exit;
                end
            I = I + 1;
        end
    Respuesta = 1;
end ^
```

Ya sé que hay implementaciones más eficientes, pero no quería complicar mucho el ejemplo. Observe, de paso, la pirueta que he tenido que realizar para ver si el número es divisible por el candidato a divisor. He utilizado el criterio del lenguaje C para las expresiones lógicas: devuelvo 1 si el número es primo, y 0 si no lo es. Recuerde que InterBase no tiene un tipo *Boolean*.

Ahora, en base al procedimiento anterior, implementamos el nuevo procedimiento *Primos*:

```
create procedure Primos(Total integer)
    returns (Primo Integer) as
declare variable I integer;
declare variable Respuesta integer;
begin
    I = 0;
    Primo = 2;
    while (I < Total) do
        begin
            execute procedure EsPrimo Primo
```

```

        returning_values Respuesta;
    if (Respuesta = 1) then
    begin
        I = I + 1;
        suspend; /* iii Nuevo !!! */
    end
    Primo = Primo + 1;
end
end ^

```

Este procedimiento puede ejecutarse en dos contextos diferentes: como un procedimiento normal, o como procedimiento de selección. Como procedimiento normal, utilizamos la instrucción **execute procedure**, como hasta ahora:

```
execute procedure Primos(100);
```

No obstante, no van a resultar tan sencillas las cosas. Esta llamada, si se realiza desde *Windows ISQL*, solamente devuelve el primer número primo (era el 2, ¿o no?). El problema es que, en ese contexto, la primera llamada a **suspend** termina completamente el algoritmo.

La segunda posibilidad es utilizar el procedimiento *como si fuera una tabla o vista*. Desde *Windows ISQL* podemos lanzar la siguiente instrucción, que nos mostrará los primeros cien números primos:

```
select * from Primos(100);
```

Por supuesto, el ejemplo anterior se refiere a una secuencia aritmética. En la práctica, un procedimiento de selección se implementa casi siempre llamando a **suspend** dentro de una instrucción **for**, que recorre las filas de una consulta.

## Recorriendo un conjunto de datos

En esta sección mostraré un par de ejemplos más complicados de procedimientos que utilizan la instrucción **for/select** de InterBase. El primero tiene que ver con un sistema de entrada de pedidos. Supongamos que queremos actualizar las existencias en el inventario después de haber grabado un pedido. Tenemos dos posibilidades, en realidad: realizar esta actualización mediante un *trigger* que se dispare cada vez que se guarda una línea de detalles, o ejecutar un procedimiento almacenado al finalizar la grabación de todas las líneas del pedido.

La primera técnica será explicada en breve, pero adelanto en estos momentos que tiene un defecto. Pongamos como ejemplo que dos usuarios diferentes están pasando por el cajero, simultáneamente. El primero saca un *pack* de Coca-Colas de la cesta de la compra, mientras el segundo pone Pepsis sobre el mostrador. Si, como es de esperar, la grabación del pedido tiene lugar mediante una transacción, al dispararse el *trigger* se han modificado las filas de estas dos marcas de bebidas, y se han bloqueado hasta el final de la transacción. Ahora, inesperadamente, el primer usuario saca Pepsis

mientras el segundo nos sorprende con Coca-Colas; son unos fanáticos de las bebidas americanas estos individuos. El problema es que el primero tiene que esperar a que el segundo termine para poder modificar la fila de las Pepsis, mientras que el segundo se halla en una situación similar.

Esta situación se denomina *abrazo mortal* (*deadlock*) y realmente no es problema alguno para InterBase, en el cual los procesos fallan inmediatamente cuando se les niega un bloqueo<sup>11</sup>. Pero puede ser un peligro en otros sistemas con distinta estrategia de espera. La solución más común consiste en que cuando un proceso necesita bloquear ciertos recursos, lo haga siempre en el mismo orden. Si nuestros dos consumidores de líquidos oscuros con burbujas hubieran facturado sus compras en orden alfabético, no se hubiera producido este conflicto. Por supuesto, esto descarta el uso de un *trigger* para actualizar el inventario, pues hay que esperar a que estén todos los productos antes de ordenar y realizar entonces la actualización. El siguiente procedimiento se encarga de implementar el algoritmo explicado:

```
create procedure ActualizarInventario(Pedido integer) as
declare variable CodArt integer;
declare variable Cant integer;
begin
    for select RefArticulo, Cantidad
      from Detalles
     where RefPedido = :Pedido
    order by RefArticulo
    into   :CodArt, :Cant do
        update Articulos
       set  Pedidos = Pedidos + :Cant
      where Codigo = :CodArt;
end ^
```

Otro ejemplo: necesitamos conocer los diez mejores clientes de nuestra tienda. Pero sólo los diez primeros, y no vale mirar hacia otro lado cuando aparezca el undécimo. Algunos sistemas SQL tienen extensiones con este propósito (**top** en SQL Server; **fetch first** en DB2), pero no InterBase. Este procedimiento, que devuelve un conjunto de datos, nos servirá de ayuda:

```
create procedure MejoresClientes(Rango integer)
returns (Codigo int, Nombre varchar(30), Total int) as
begin
    for select Codigo, Nombre, sum(Total)
      from Clientes, Pedidos
     where Clientes.Codigo = Pedidos.Cliente
    order by 3 desc
    into   :Codigo, :Nombre, :Total do
        begin
            suspend;
            Rango = Rango - 1;
            if (Rango = 0) then
                exit;
        end
    end
```

---

<sup>11</sup> Realmente, es el BDE quien configura a InterBase de este modo. En la versión 5.0.1.24 se introduce el nuevo parámetro *WAIT ON LOCKS*, para modificar este comportamiento.

```
end ^
```

Entonces podremos realizar consultas como la siguiente:

```
select *
from MejoresClientes(10)
```

## Triggers, o disparadores

Uno de los recursos más interesantes de los sistemas de bases de datos relacionales son los *triggers*, o disparadores; en adelante, utilizaré preferentemente la palabra inglesa original. Se trata de un tipo de procedimiento almacenado que se activa automáticamente al efectuar operaciones de modificación sobre ciertas tablas de la base de datos.

La sintaxis de la declaración de un *trigger* es la siguiente:

```
create trigger NombreTrigger for Tabla [active | inactive]
{before | after} {delete | insert | update}
[position Posición]
as CuerpoDeProcedimiento
```

El cuerpo de procedimiento tiene la misma sintaxis que los cuerpos de los procedimientos almacenados. Las restantes cláusulas del encabezamiento de esta instrucción tienen el siguiente significado:

Cláusula	Significado
<i>NombreTrigger</i>	El nombre que se le va a asignar al <i>trigger</i>
<i>Tabla</i>	El nombre de la tabla a la cual está asociado
<b>active</b>   <b>inactive</b>	Puede crearse inactivo, y activarse después
<b>before</b>   <b>after</b>	Se activa antes o después de la operación
<b>delete</b>   <b>insert</b>   <b>update</b>	Qué operación provoca el disparo del <i>trigger</i>
<b>position</b>	Orden de disparo para la misma operación

A diferencia de otros sistemas de bases de datos, los *triggers* de InterBase se definen para una sola operación sobre una sola tabla. Si queremos compartir código para eventos de actualización de una o varias tablas, podemos situar este código en un procedimiento almacenado y llamarlo desde los diferentes *triggers* definidos.

Un parámetro interesante es el especificado por **position**. Para una operación sobre una tabla pueden definirse varios *triggers*. El número indicado en **position** determina el orden en que se disparan los diferentes sucesos; mientras más bajo sea el número, mayor será la prioridad. Si dos *triggers* han sido definidos con la misma prioridad, el orden de disparo entre ellos será aleatorio.

Hay una instrucción similar que permite modificar algunos parámetros de la definición de un *trigger*, como su orden de disparo, si está activo o no, o incluso su propio cuerpo:

```
alter trigger NombreTrigger [active | inactive]
    [{before | after} {delete | insert | update}]
    [position Posición]
    [as CuerpoProcedimiento]
```

Podemos eliminar completamente la definición de un *trigger* de la base de datos mediante la instrucción:

```
drop trigger NombreTrigger
```

## Las variables *new* y *old*

Dentro del cuerpo de un *trigger* pueden utilizarse las variables predefinidas *new* y *old*. Estas variables hacen referencia a los valores nuevos y anteriores de las filas involucradas en la operación que dispara el *trigger*. Por ejemplo, en una operación de modificación **update**, *old* se refiere a los valores de la fila antes de la modificación y *new* a los valores después de modificados. Para una inserción, solamente tiene sentido la variable *new*, mientras que para un borrado, solamente tiene sentido *old*.

El siguiente *trigger* hace uso de la variable *new*, para acceder a los valores del nuevo registro después de una inserción:

```
create trigger UltimaFactura for Pedidos
    active after insert position 0 as
declare variable UltimaFecha date;
begin
    select UltimoPedido
    from Clientes
    where Codigo = new.RefCliente
    into :UltimaFecha;
    if (UltimaFecha < new.FechaVenta) then
        update Clientes
        set UltimoPedido = new.FechaVenta
        where Codigo = new.RefCliente;
end ^
```

Este *trigger* sirve de contraejemplo a un error muy frecuente en la programación SQL. La primera instrucción busca una fila particular de la tabla de clientes y, una vez encontrada, extrae el valor de la columna *UltimoPedido* para asignarlo a la variable local *UltimaFecha*. El error consiste en pensar que esta instrucción, a la vez, deja a la fila encontrada como “fila activa”. El lenguaje de *triggers* y procedimientos almacenados de InterBase, y la mayor parte de los restantes sistemas, no utiliza “filas activas”. Es por eso que en la instrucción **update** hay que incluir una cláusula **where** para volver a localizar el registro del cliente. De no incluirse esta cláusula, cambiaríamos la fecha para *todos* los clientes.

Es posible cambiar el valor de una columna correspondiente a la variable *new*, pero solamente si el *trigger* se define “antes” de la operación de modificación. En cualquier caso, el nuevo valor de la columna se hace efectivo después de que la operación tenga lugar.



## Más ejemplos de *triggers*

Para mostrar el uso de *triggers*, las variables *new* y *old* y los procedimientos almacenados, mostraré cómo se puede actualizar automáticamente el inventario de artículos y el total almacenado en la tabla de pedidos en la medida en que se realizan actualizaciones en la tabla que contiene las líneas de detalles.

Necesitaremos un par de procedimientos auxiliares para lograr una implementación más modular. Uno de estos procedimientos, *RecalcularTotal*, debe actualizar el total de venta de un pedido determinado, y ya lo hemos programado antes. Repito aquí su código, para mayor comodidad:

```
create procedure RecalcularTotal(NumPed int) as
declare variable Total integer;
begin
    select sum(Cantidad * PVP * (100 - Descuento) / 100)
    from Detalles, Articulos
    where Detalles.RefArticulo = Articulos.Codigo
        and Detalles.RefPedido = :NumPed
    into :Total;
    if (Total is null) then
        Total = 0;
    update Pedidos
    set Total = :Total
    where Numero = :NumPed;
end ^
```

El otro procedimiento debe modificar el inventario de artículos. Su implementación es muy simple:

```
create procedure ActInventario(CodArt integer, Cant Integer) as
begin
    update Articulos
    set Pedidos = Pedidos + :Cant
    where Codigo = :CodArt;
end ^
```

Ahora le toca el turno a los *triggers*. Los más sencillos son los relacionados con la inserción y borrado; en el primero utilizaremos la variable *new*, y en el segundo, *old*.

```
create trigger NuevoDetalle for Detalles
    active after insert position 1 as
begin
    execute procedure RecalcularTotal new.RefPedido;
    execute procedure ActInventario
        new.RefArticulo, new.Cantidad;
end ^

create trigger EliminarDetalle for Detalles
    active after delete position 1 as
declare variable Decremento integer;
begin
    Decremento = - old.Cantidad;
    execute procedure RecalcularTotal old.RefPedido;
```

```

        execute procedure ActInventario
            old.RefArticulo, :Decremento;
    end ^

```

Es curiosa la forma en que se pasan los parámetros a los procedimientos almacenados. Tome nota, en particular, de que hemos utilizado una variable local, *Decremento*, en el *trigger* de eliminación. Esto es así porque no se puede pasar expresiones como parámetros a los procedimientos almacenados, ni siquiera para los parámetros de entrada.

Finalmente, nos queda el *trigger* de modificación:

```

create trigger ModificarDetalle for Detalles
    active after update position 1 as
declare variable Decremento integer;
begin
    execute procedure RecalcularTotal new.RefPedido;
    if (new.RefArticulo <> old.RefArticulo) then
    begin
        Decremento = -old.Cantidad;
        execute procedure ActInventario
            old.RefArticulo, :Decremento;
        execute procedure ActInventario
            new.RefArticulo, new.Cantidad;
    end
    else
    begin
        Decremento = new.Cantidad - old.Cantidad;
        execute procedure ActInventario
            new.RefArticulo, :Decremento;
    end
end ^

```

Observe cómo comparamos el valor del código del artículo antes y después de la operación. Si solamente se ha producido un cambio en la cantidad vendida, tenemos que actualizar un solo registro de inventario; en caso contrario, tenemos que actualizar dos registros. No hemos tenido en cuenta la posibilidad de modificar el pedido al cual pertenece la línea de detalles. Suponemos que esta operación no va a permitirse, por carecer de sentido, en las aplicaciones clientes.

## **Triggers y vistas**

Al presentar las vistas, explicamos que sólo algunas de ellas eran actualizables. Las vistas que son de sólo lectura casi siempre entran en esta categoría por una de tres razones:

- 1 No se puede establecer matemáticamente una semántica apropiada para su actualización. En buen cristiano, hay vistas para las que no tiene sentido decir: “elimina esta fila”, “crea un nuevo registro”, etc.
- 2 Existe más de una interpretación plausible para la actualización, e InterBase no puede elegir entre ellas.

- 3 Existe una sola interpretación plausible, pero InterBase cree que sería demasiado costosa de implementar.

Con una consulta no actualizable por el primer o tercer motivo, hay muy poco que hacer. En el caso contrario, InterBase nos permite que demos nuestra visión sobre qué significa actualizar la vista, y el mecanismo ofrecido son los *triggers*. Tomemos como punto de partida las siguientes tablas simplificadas:

```
create table PAISES (
    IDPais      integer      not null,
    Pais        varchar(30)  not null,

    primary key (IDPais)
);

create table CLIENTES (
    IDCliente   integer      not null,
    Nombre      varchar(35)  not null,
    IDPais      integer      not null,

    primary key (IDCliente),
    foreign key (IDPais) references PAISES(IDPais)
);
```

Sería útil disponer de una vista como la siguiente:

```
create view CLIENTES_V as
select c.*, p.Pais
from   CLIENTES c inner join PAISES p on c.IDPais = p.IDPais;
```

¿Qué significa eliminar una fila de *CLIENTES\_V*? Es fácil ver que la intención del usuario es eliminar el cliente; InterBase no puede estar completamente seguro de que no signifique eliminar el registro del país. Nosotros identificamos enseguida la relación un país, muchos clientes, pero tenga presente que la restricción **foreign key** solamente *sugiere* la existencia de la relación; podría, por ejemplo, tratarse de una relación uno/uno opcional.

Para lograr que la vista presentada sea actualizable, debemos definir *triggers* similares a los siguientes:

```
create trigger BI_CLIENTES_V for CLIENTES_V
active before insert as
declare variable i integer;
begin
select IDPais
from   PAISES
where  IDPais = new.IDPais
into   :i;
if (i is null) then
insert into PAISES (IDPais, Pais)
values (new.IDPais, new.Pais);
insert into CLIENTES (IDCliente, Nombre, IDPais)
values (new.IDCliente, new.Nombre, new.IDPais);
end ^
```

```

create trigger BU_CLIENTES_V for CLIENTES_V
active before update as
begin
    update CLIENTES
    set     Nombre = new.Nombre,
           IDPais = new.Pais
    where  IDCliente = old.IDCliente;
end ^

create trigger BD_CLIENTES_V for CLIENTES_V
active before delete as
begin
    delete from CLIENTES
    where  IDCliente = old.IDCliente;
end ^

```

Tome nota de que es incluso posible crear el registro del país “al vuelo”, de ser necesario. En el *trigger* de actualización debe observar que no he tenido en cuenta la posibilidad de cambiar el identificador del cliente.

### ADVERTENCIA

Si la vista sobre la que se definen los triggers es actualizable, InterBase no desconecta el mecanismo de actualización original, y nos advierte en la documentación que podemos encontrar interacciones y resultados inesperados.

## Generadores

Los *generadores* (*generators*) son un recurso de InterBase para poder disponer de valores secuenciales, que pueden utilizarse, entre otras cosas, para garantizar la unicidad de las claves artificiales. Un generador se crea, del mismo modo que los procedimientos almacenados y *triggers*, en un fichero *script* de SQL. El siguiente ejemplo muestra cómo crear un generador:

```
create generator CodigoEmpleado;
```

Un generador define una variable interna persistente, cuyo tipo es un entero de 32 bits. Aunque esta variable se inicializa automáticamente a 0, tenemos una instrucción para cambiar el valor de un generador:

```
set generator CodigoEmpleado to 1000;
```

Por el contrario, no existe una instrucción específica que nos permita eliminar un generador. Esta operación debemos realizarla directamente en la tabla del sistema que contiene las definiciones y valores de todos los generadores:

```
delete from rdb$generators
where rdb$generator_name = 'CODIGOEMPLEADO'
```

Para utilizar un generador necesitamos la función *gen\_id*. Esta función utiliza dos parámetros. El primero es el nombre del generador, y el segundo debe ser la cantidad en la que se incrementa o decrementa la memoria del generador. La función retorna

entonces el valor ya actualizado. Utilizaremos el generador anterior para suministrar automáticamente un código de empleado si la instrucción **insert** no lo hace:

```
create trigger NuevoEmpleado for Empleados
  active before insert
as
begin
  if (new.Codigo is null) then
    new.Codigo = gen_id(CodigoEmpleado, 1);
  end ^
```

Al preguntar primeramente si el código del nuevo empleado es nulo, estamos permitiendo la posibilidad de asignar manualmente un código de empleado durante la inserción.

Los programas escritos en Delphi con el BDE tienen problemas cuando se asigna la clave primaria de una fila dentro de un *trigger* utilizando un generador, pues el registro recién insertado “desaparece” según el punto de vista de la tabla. Este problema se presenta sólo cuando estamos navegando simultáneamente sobre la tabla.

Para no tener que abandonar los generadores, una de las soluciones consiste en crear un procedimiento almacenado que obtenga el próximo valor del generador, y utilizar este valor para asignarlo a la clave primaria en el evento *BeforePost* de la tabla. En el lado del servidor se programaría algo parecido a lo siguiente:

```
create procedure ProximoCodigo returns (Cod integer) as
begin
  Cod = gen_id(CodigoEmpleado);
end ^
```

En la aplicación crearíamos un componente *spProximoCodigo*, de la clase *TStoredProc*, y lo aprovecharíamos de esta forma en uno de los eventos *BeforePost* o *OnNewRecord* de la tabla de clientes:

```
procedure TmodDatos.tbClientesBeforePost (DataSet: TDataSet);
begin
  spProximoCodigo.ExecProc;
  tbClientesCodigo.Value :=
    spProximoCodigo.ParamByName('COD').AsInteger;
end;
```

De todos modos, si la tabla cumple determinados requisitos, podemos ahorrarnos trabajo en la aplicación y seguir asignando la clave primaria en el *trigger*. Las condiciones necesarias son las siguientes:

- 1 La tabla no debe tener columnas con valores **default**. Así evitamos que el BDE tenga que releer la fila después de su creación.
- 2 Debe existir un índice único, o casi único, sobre alguna columna alternativa a la clave primaria. La columna de este índice se utilizará entonces como criterio de ordenación para la navegación.

- 3 El valor de la clave primaria no nos importa realmente, como sucede con las claves artificiales.

Las tablas de referencia que abundan en toda base de datos son un buen ejemplo de la clase de tablas anterior. Por ejemplo, si necesitamos una tabla para los diversos valores del estado civil, probablemente la definamos de este modo:

```
create table EstadoCivil (
    Codigo          integer not null primary key,
    Descripcion     varchar(15) not null unique,
    EsperanzaVida   integer not null
);

create generator EstadoCivilGen;

set term ^;

create trigger BIEstadoCivil for EstadoCivil
    active before insert as
begin
    Codigo = gen_id(EstadoCivilGen, 1);
end ^
```

En Delphi asociaremos una tabla o consulta a la tabla anterior, pero ordenaremos las filas por su descripción, y ocultaremos el campo *Codigo*, que será asignado automáticamente en el servidor. Recuerde, en cualquier caso, que los problemas con la asignación de claves primarias en el servidor son realmente problemas de la navegación con el BDE, y nada tienen que ver con InterBase, en sí. Cuando estudiemos DataSnap, además, veremos cómo estos problemas se resuelven con elegancia en este sistema.

#### NOTA IMPORTANTE

En cualquier caso, si necesita valores únicos y *consecutivos* en alguna columna de una tabla, no utilice generadores (ni secuencias de Oracle, o identidades de MS SQL Server). El motivo es que los generadores no se bloquean durante las transacciones. Usted pide un valor dentro de una transacción, y le es concedido; todavía no ha terminado su transacción. A continuación, otro usuario pide el siguiente valor, y sus deseos se cumplen. Pero entonces usted aborta la transacción, por el motivo que sea. La consecuencia: se pierde el valor que recibió, y se produce un "hueco" en la secuencia.

## Excepciones

Sin embargo, todavía no contamos con medios para detener una operación SQL; esta operación sería necesaria para simular imperativamente las restricciones a la propagación de cambios en cascada, en la integridad referencial. Lo que nos falta es poder lanzar excepciones desde un *trigger* o procedimiento almacenado. Las excepciones de InterBase se crean asociando una cadena de mensaje a un identificador:

```
create exception CLIENTE_CON_PEDIDOS
    "No se puede modificar este cliente"
```

Es necesario confirmar la transacción actual para poder utilizar una excepción recién creada. Existen también instrucciones para modificar el mensaje asociado a una excepción (**alter exception**), y para eliminar la definición de una excepción de la base de datos (**drop exception**).

Una excepción se lanza desde un procedimiento almacenado o *trigger* mediante la instrucción **exception**:

```
create trigger CheckDetails for Clientes
  active before delete
  position 0 as
declare variable Numero int;
begin
  select count(*)
  from Pedidos
  where RefCliente = old.Codigo
  into :Numero;
  if (:Numero > 0) then
    exception CLIENTE_CON_PEDIDOS;
end ^
```

Las excepciones de InterBase determinan que cualquier cambio realizado dentro del cuerpo del *trigger* o procedimiento almacenado, sea directa o indirectamente, se anule automáticamente. De esta forma puede programarse algo parecido a las transacciones anidadas de otros sistemas de bases de datos.

Si la instrucción **exception** es similar a la instrucción **raise** de Delphi, el equivalente más cercano a **try/except** es la instrucción **when** de InterBase. Esta instrucción tiene tres formas diferentes. La primera intercepta las excepciones lanzadas con **exception**:

```
when exception NombreExcepción do
  BloqueInstrucciones;
```

Con la segunda variante, se detectan los errores producidos por las instrucciones SQL:

```
when sqlcode Numero do
  BloqueInstrucciones;
```

Los números de error de SQL aparecen documentados en la ayuda en línea y en el manual *Language Reference* de InterBase. A grandes rasgos, la ejecución correcta de una instrucción devuelve un código igual a 0, cualquier valor negativo es un error propiamente dicho (-803, por ejemplo, es un intento de violación de una clave primaria), y los valores positivos son advertencias. En particular, 100 es el valor que se devuelve cuando una selección única no encuentra el registro buscado. Este convenio es parte del estándar de SQL, aunque los códigos de error concreto varíen de un sistema a otro.

La tercera forma de la instrucción **when** es la siguiente:

```

when gdscode Numero do
    BloqueInstrucciones;

```

En este caso, se están interceptando los mismos errores que con **sqlcode**, pero se utilizan los códigos internos de InterBase, que ofrecen más detalles sobre la causa. Por ejemplo, los valores 335544349 y 35544665 corresponden a -803, la violación de unicidad, pero el primero se produce cuando se inserta un valor duplicado en cualquier índice único, mientras que el segundo se reserva para las violaciones específicas de clave primaria o alternativa.

En cualquier caso, las instrucciones **when** deben ser las últimas del bloque en que se incluyen, y pueden colocarse varias simultáneamente, para atender varios casos:

```

begin
    /* Instrucciones */
    /* ... */
    when sqlcode -803 do
        Resultado = "Violación de unicidad";
    when exception CLIENTE_CON_PEDIDOS do
        Resultado = "Elimine primero los pedidos realizados";
end

```

La Tercera Regla de Martens sigue siendo aplicable a estas instrucciones: no detenga la propagación de una excepción, a no ser que tenga una solución a su causa.

## Alertadores de eventos

Los alertadores de eventos (*event alerter*s) son un recurso único, por el momento, de InterBase. Los procedimientos almacenados y *triggers* de InterBase pueden utilizar la instrucción siguiente:

```

post_event NombreDeEvento

```

El nombre de evento puede ser una constante de cadena o una variable del mismo tipo. Cuando se produce un evento, InterBase avisa a todos los clientes interesados de la ocurrencia del mismo.

Los alertadores de eventos son un recurso muy potente. Sitúese en un entorno cliente/servidor donde se producen con frecuencia cambios en una base de datos. Las estaciones de trabajo normalmente no reciben aviso de estos cambios, y los usuarios deben actualizar periódica y frecuentemente sus pantallas para reflejar los cambios realizados por otros usuarios, pues en caso contrario puede suceder que alguien tome una decisión equivocada en base a lo que está viendo en pantalla. Sin embargo, refrescar la pantalla toma tiempo, pues hay que traer cierta cantidad de información desde el servidor de bases de datos, y las estaciones de trabajo realizan esta operación periódicamente, colapsando la red. El personal de la empresa se aburre en los tiempos de espera, la moral se resquebraja y la empresa se sitúa al borde del caos...



Entonces aparece Usted, un experto programador de Delphi e InterBase, y añade *triggers* a discreción a la base de datos, como éste:

```
create trigger AlertarCambioBolsa for Cotizaciones
    active after update position 10
as
begin
    post_event "CambioCotizacion";
end ^
```

Observe que se ha definido una prioridad baja para el orden de disparo del *trigger*. Hay que aplicar la misma técnica para cada una de las operaciones de actualización de la tabla de cotizaciones.

Luego, en el módulo de datos de la aplicación que se ejecuta en las estaciones de trabajo, hay que añadir el componente *TIBEventAlerter*, que se encuentra en la página *Samples* de la Paleta de Componentes. Este componente tiene las siguientes propiedades, métodos y eventos:

Nombre	Tipo	Propósito
<i>Events</i>	Propiedad	Los nombres de eventos que nos interesan.
<i>Registered</i>	Propiedad	Debe ser <i>True</i> para notificar, en tiempo de diseño, nuestro interés en los eventos almacenados en la propiedad anterior.
<i>Database</i>	Propiedad	La base de datos a la cual nos conectaremos.
<i>RegisterEvents</i>	Método	Notifica a la base de datos nuestro interés por los eventos de la propiedad <i>Events</i> .
<i>UnRegisterEvents</i>	Método	El inverso del método anterior.
<i>OnEvent.Alert</i>	Evento	Se dispara cada vez que se produce el evento.

En nuestro caso, podemos editar la propiedad *Events* y teclear la cadena *CambioCotizacion*, que es el nombre del evento que necesitamos. Conectamos la propiedad *Database* del componente a nuestro componente de bases de datos y activamos la propiedad *Registered*. Luego creamos un manejador para el evento *OnEvent.Alert* similar a éste:

```
procedure TForm1.IBEventAlerter1EventAlert(Sender: TObject;
    EventName: string; EventCount: Integer;
    var CancelAlerts: Boolean);
begin
    tbCotizaciones.Refresh;
end;
```

Cada vez que se modifique el contenido de la tabla *Cotizaciones*, el servidor de InterBase lanzará el evento identificado por la cadena *CambioCotizacion*, y este evento será recibido por todas las aplicaciones interesadas. Cada aplicación realizará consecuentemente la actualización visual de la tabla en cuestión.

Esta historia termina previsiblemente. La legión de usuarios del sistema lo aclama con fervor, su jefe le duplica el salario, usted se casa ... o se compra un perro ... o ... Bueno, se me ha complicado un poco el guión; póngale usted su final preferido.

#### NOTA

El componente *TIBEventAlerter* se ha quedado obsoleto, en las últimas versiones de Delphi. Cuando estudiemos InterBase Express, en el capítulo 29, presentaremos otro componente que realiza la misma función: *TIBEvents*. Quiero advertirle también que el sistema de eventos ha estado plagado de *bugs* inexplicables a lo largo de su historia. Antes de apostar fuerte por él, compruebe que funcionan correctamente con su versión de InterBase.

## Funciones de usuario en InterBase

Para finalizar el capítulo, mostraré un ejemplo de cómo utilizar DLLs para extender la funcionalidad de un servidor de InterBase. Los servidores de InterBase basados en Windows admiten la creación de *funciones definidas por el usuario* (*User Defined Functions*, ó *UDF*). Estas funciones se definen en DLLs que se deben registrar en el servidor, y pueden ser ejecutadas desde consultas SQL, *triggers* y procedimientos almacenados.

Los pasos para crear una función de usuario son los siguientes:

- Programe la DLL, exportando las funciones deseadas.
- Copie la DLL resultante al directorio *bin* del servidor de InterBase. Si se trata de un servidor local, o si tenemos acceso al disco duro del servidor remoto, esto puede realizarse cambiando el directorio de salida en las opciones del proyecto.
- Utilice la instrucción **declare external function** de InterBase para registrar la función en la base de datos correspondiente. Para facilitar el uso de la extensión programada, puede acompañar a la DLL con las declaraciones correspondientes almacenadas en un *script* SQL.

Para ilustrar la técnica, crearemos una función que devuelva el nombre del día de la semana de una fecha determinada. La declaración de la función, en la sintaxis de InterBase, será la siguiente:

```
declare external function DiaSemana (DATE)
returns cstring(15)
entry_point "DiaSemana" module_name "MisUdfs.dll";
```

Aunque podemos comenzar declarando la función, pues InterBase cargará la DLL sólo cuando sea necesario, es preferible comenzar creando la DLL, así que cree un nuevo proyecto DLL, con el nombre *MisUdfs*.

Las funciones de usuario de InterBase deben implementarse con la directiva **cdecl**. Hay que tener en cuenta que todos los parámetros se pasan por referencia; incluso los valores de retorno de las funciones se pasan por referencia (se devuelve un puntero), si no se especifica la opción **by value** en la declaración de la función. La co-

Correspondencia entre tipos de datos de InterBase y Delphi es sencilla: **int** equivale a *Integer*, **smallint** a *Smallint*, las cadenas de caracteres se pasan como punteros a caracteres *PChar*, y así sucesivamente. En particular, las fechas se pasan en un tipo de registro con la siguiente declaración:

```
type
  TIBDate = record
    Days, Frac: Integer;
  end;
```

*Days* es la cantidad de días transcurridos a partir de una fecha determinada por InterBase, el 17 de noviembre de 1858. *Frac* es la cantidad de diezmilésimas de segundos transcurridas desde las doce de la noche. Con esta información en nuestras manos, es fácil programar la función *DiaSemana*:

```
function DiaSemana(var fecha: TIBDate): PChar; cdecl; export;
const
  Dias: array [0..6] of PChar = ('Miércoles', 'Jueves',
    'Viernes', 'Sábado', 'Domingo', 'Lunes', 'Martes');
begin
  Result := Dias[Fecha.Days mod 7];
end;
```

Para saber qué día de la semana corresponde al “día de la creación” de InterBase, tuvimos que realizar un proceso sencillo de prueba y error; parece que para alguien en este mundo los miércoles son importantes.

#### NOTA

Recuerde que a partir de InterBase 6 ya existen funciones predefinidas para extraer componentes de los valores de tipo fecha.

Una vez compilado el proyecto, asegúrese que la DLL generada está presente en el directorio *bin* del servidor de InterBase. Active entonces la utilidad *WISQL*, conéctese a una base de datos que contenga tablas con fechas, teclee la instrucción **declare external function** que hemos mostrado anteriormente y ejecútela. A continuación, pruebe el resultado, con una consulta como la siguiente:

```
select DiaSemana(SaleDate), SaleDate,
       cast("Now" as date), DiaSemana("Now")
from   Orders
```

Tenga en cuenta que, una vez que el servidor cargue la DLL, ésta quedará en memoria hasta que el servidor se desconecte. De este modo, para sustituir la DLL (para añadir funciones o corregir errores) debe primero detener al servidor y volver a iniciarlo posteriormente.

Hay que tener cuidado con las funciones que devuelven cadenas de caracteres generadas por la DLL. Con toda seguridad necesitarán un *buffer* para devolver la cadena, que debe ser suministrado por la DLL. No se puede utilizar una variable global con este propósito debido a la arquitectura multihilos: todas las conexiones de clientes

comparten un mismo proceso en el servidor, y si varias de ellas utilizan una misma UDF, accediendo a la función desde distintos hilos, es muy probable que terminemos sobrescribiendo el contenido de la variable global.

Por ejemplo, ésta es la implementación en Delphi de una función de usuario para convertir cadenas a minúsculas:

```
function Lower(S: PChar): PChar; cdecl; export;
var
    Len: Integer;
begin
    Len := StrLength(S);
    Result := SysGetMem(Len + 1);
    StrCopy(Result, S);
    CharLowerBuff(Result, Len);
end;
```

Si queremos utilizar esta función desde InterBase, debemos declararla mediante la siguiente instrucción:

```
declare external function lower cstring(256)
returns cstring (256) free_it
entry_point "Lower" module_name "MisUdfs.dll"
```

Observe el uso de la nueva palabra reservada **free\_it**, para indicar que la función reserva memoria que debe ser liberada por el servidor.

### ADVERTENCIA

Las UDFs pueden ser necesarias, y en ocasiones inevitables. Pero es una mala idea depender de ellas. ¿Demostración? Suponga que tiene un servidor de InterBase ejecutándose en Windows 2000, y que un buen día su empresa decide que hay que pasar el servidor a Solaris. ¿Cómo recompila las UDFs para que funcionen en el nuevo sistema operativo?

# Transacciones

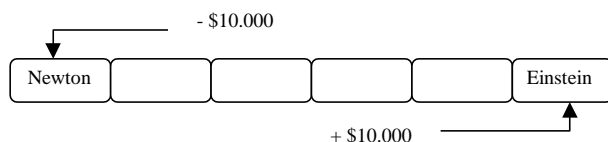
**T**ODO O NADA, Y QUE NO ME ENTERE YO que andas cerca. Parece una amenaza, pero no lo es. La frase anterior puede resumir el comportamiento egoísta deseable para las *transacciones*: el mecanismo que ofrecen las bases de datos para garantizar la coherencia de su contenido, especialmente cuando varias aplicaciones acceden simultáneamente al mismo.

En este capítulo explicamos la teoría general de las transacciones, desde el punto de vista de las bases de datos SQL y de escritorio, y cómo son implementadas por los diferentes sistemas. El programador acostumbrado a trabajar con bases de datos de escritorio, asocia la solución de los problemas de concurrencia con la palabra mágica “bloqueos”. Como veremos, esto es sólo parte de la verdad, y en ocasiones, ni siquiera es verdad. Los sistemas profesionales de bases de datos utilizan los bloqueos como un posible mecanismo de implementación del control de concurrencia a bajo nivel. Y el programador debe trabajar y pensar en *transacciones*, como forma de asegurar la consistencia de sus operaciones en la base de datos.

## ¿Por qué necesitamos transacciones?

Por omisión, cuando realizamos modificaciones en tablas a través de cualquiera de las interfaces de Delphi, excepto InterBase Express, cada operación individual es independiente de las operaciones que le preceden y de las que siguen a continuación. El fallo de una de ellas no afecta a las demás. Sin embargo, existen ocasiones en que nos interesa ligar la suerte de varias operaciones consecutivas sobre una base de datos.

El ejemplo clásico es la transferencia bancaria: hay que restar del saldo de un registro y aumentar en la misma cantidad el saldo de otro. No podemos permitir que, una vez actualizado el primer registro nos encontremos que alguien está trabajando con el segundo registro, y se nos quede el dinero de la transferencia en el limbo. Y no es solución regresar al primer registro y reingresar la cantidad extraída, pues puede que otro usuario haya comenzado a editar este primer registro después de haberlo abandonado nosotros. En este caso nos veríamos como un jugador de béisbol atrapado entre dos bases.



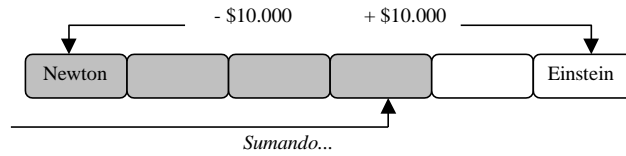
O considere una subida salarial a cierto grupo de empleados de una empresa. En este caso, es mucho más difícil dar marcha atrás a la operación si se produce un error a mediados de la misma. Normalmente, los programadores que vienen del mundo de las bases de datos locales atacan estos problemas blandiendo bloqueos a diestra y siniestra. En el caso de la transferencia, un bloqueo sobre la cuenta destino y la cuenta origen y ¡venga transferencia! En el caso de la subida masiva, un bloqueo sobre la tabla completa, y ¡pobre del que intente acceder a la tabla mientras tanto! Es cierto que los bloqueos son una de las muchas maneras de resolver los problemas de acceso concurrente (aunque no la mejor). Pero una actualización puede fallar por muchos más motivos que por un bloqueo denegado; una violación de alguna restricción de validez puede dejarnos a mitad de una operación larga de actualización sin saber cómo retroceder.

La forma de salir de éste y de otros atolladeros similares es utilizar el concepto de *transacción*. Una transacción es una secuencia de operaciones de lectura y escritura durante las cuales se puede ver la base de datos como un todo consistente y, si se realizan actualizaciones, dejarla en un estado consistente. Estoy consciente de que la oración anterior parece extraída de un libro de filosofía, por lo cual dedicaré los próximos párrafos a despejar la niebla.

En primer lugar: “ver la base de datos como un todo consistente”. Nada que ver con la ecología ni con la Tabla Esmeralda. Con esto quiero decir que, durante todo el intervalo que está activa la transacción, los valores leídos y no modificados por la misma permanecen estables, y si al principio de la misma satisfacían las reglas de integridad, siguen cumpliéndolas durante todo el tiempo de vida de la transacción.

¿Elemental? No tanto. Suponga que una transacción quiere sumar los saldos de las cuentas depositadas en nuestro banco, y comienza a recorrer la tabla pertinente. Suponga también que las filas están almacenadas por orden alfabético de acuerdo al apellido, y que la transacción se encuentra ahora mismo analizando la cuenta de cierto sujeto de apellido Marteens.

En ese preciso momento, llega un tal Albert Einstein y quiere transferir diez mil dólares a la cuenta de un tal Isaac Newton (gente importante los clientes de este banco). La fila de Mr. Einstein se actualiza, decrementando su saldo; esta fila ya ha sido leída por la transacción que suma. Luego, la fila de Mr. Newton incrementa su saldo en la cantidad correspondiente. Y esta fila no ha sido leída aún por la transacción sumadora. Por lo tanto, esta transacción al final reportará diez mil dólares de más en el saldo total almacenado; diez mil dólares inexistentes, que es lo más triste del asunto.



En segundo lugar: “si se realizan actualizaciones, dejar la base de datos en un estado consistente”. La condición anterior es necesaria, desde un punto de vista estricto, para el cumplimiento de esta condición, pues no se puede pretender realizar una actualización que satisfaga las reglas de consistencia si la transacción puede partir de un estado no consistente. Pero implica más que esto. En particular, se necesita garantizar que toda la secuencia de operaciones consideradas dentro de una transacción se ejecute; si la transacción aborta a mitad de camino, los cambios efectuados deben poder deshacerse automáticamente. Y esto vale también para el caso especial en que el gato de la chica del piso de abajo entre por la puerta y se electrocute con el cable de alimentación del servidor. Después de retirar a la víctima y reiniciar el servidor, la base de datos no debe acusar recibo de las transacciones inconclusas: el espectáculo debe continuar<sup>12</sup>.

## El ácido sabor de las transacciones

A veces, en la literatura anglosajona, se dice que las transacciones tienen propiedades “ácidas”, por las siglas ACID: *Atomicity*, *Consistency*, *Isolation* y *Durability*. O, forzando un poco la traducción para conservar las iniciales: atomicidad, consistencia, independencia y durabilidad. La atomicidad no se refiere al carácter explosivo de las transacciones, sino al hecho de que deben ser *indivisibles*; se realizan todas las operaciones, o no se realiza ninguna. Consistencia quiere decir, precisamente, que una transacción debe llevar la base de datos de un estado consistente a otro. Independencia, porque para ser consistentes debemos imaginar que somos los únicos que tenemos acceso a la base de datos en un instante dado; las demás transacciones deben ser invisibles para nosotros. Y la durabilidad se refiere a que cuando una transacción se confirma, los cambios solamente pueden deshacerse mediante otra transacción.

Para que el sistema de gestión de base de datos reconozca una transacción tenemos que marcar sus límites: cuándo comienza y cuándo termina, además de cómo termina. Todos los sistemas SQL ofrecen, con mínimas variantes sintácticas, las siguientes instrucciones para marcar el principio y el fin de una transacción:

<sup>12</sup> Nota del censor: Me estoy dando cuenta de que en este libro se maltrata y abusa de los animales. Antes fue el perro de Codd; ahora, el gato de la vecina... Aunque os parezca mentira, una vez tuve un gato llamado Pink Floyd (aunque sólo respondía por Pinky), y un buen día se le ocurrió morder el cable de alimentación de la tele. Lo curioso es que, a pesar de que se quedó tieso, un oportuno masaje cardíaco lo devolvió a este mundo. Claro, era un gato nuevo y aún no había consumido sus siete vidas.

```

start transaction
commit work
rollback work

```

La primera de ellas señala el principio de una transacción, mientras que las dos últimas marcan el fin de la transacción. La instrucción **commit work** señala un final exitoso: los cambios se graban definitivamente; **rollback work** indica la intención del usuario de deshacer todos los cambios realizados desde la llamada a **start transaction**. Solamente puede activarse una transacción por base de datos en cada sesión. Dos usuarios diferentes, sin embargo, pueden tener concurrentemente transacciones activas.

## Transacciones implícitas y explícitas

Existen dos formas diferentes en las que una aplicación puede utilizar las transacciones. En la sección anterior he mencionado las instrucciones necesarias para marcar el principio y fin de una transacción, en lenguaje SQL. Cada una de las interfaces de acceso a datos de Delphi ofrece métodos equivalentes, generalmente asociados a los componentes de conexión: *TDatabase*, para el BDE, *TADOConnection*, para ADO, *TSQLConnection* para DB Express... El programador tiene la posibilidad de controlar explícitamente cuándo comienzan o acaban sus transacciones. Pero también pueden crearse y cerrarse transacciones sin la intervención clara del programador, y es una característica que puede implementarse tanto a nivel del servidor SQL como en la interfaz de acceso: las transacciones *implícitas*.

Tomemos como ejemplo el funcionamiento del BDE trabajando con InterBase. Si el programador no ha iniciado una transacción desde su programa, cada modificación que la aplicación intente realizar desde la estación de trabajo será englobada en una transacción para esta única operación. ¿La razón? Lo que la aplicación cliente considera una simple actualización, puede significar varias actualizaciones en realidad, si existen *triggers* asociados a la acción de modificación, como vimos en el capítulo anterior. Además, la aplicación puede ejecutar procedimientos almacenados que modifiquen varias filas de la base de datos, y es lógico desear que este conjunto de modificaciones se aplique de forma atómica: todo o nada.

Si está trabajando con InterBase a través del BDE, quizás le interese modificar el parámetro *DRIVER FLAGS* en el controlador SQL Link. Si asigna 4096 a este parámetro, al confirmar y reiniciar una transacción, se aprovecha el "contexto de transacción" existente, con lo cual se acelera la operación. Estos parámetros serán estudiados en el capítulo sobre el Motor de Datos de Borland.

Ahora bien, incluso fijando la interfaz de acceso (supongamos que es el BDE), cada servidor implementa estas transacciones implícitas de forma diferente, y es aconsejable que probemos cada combinación de servidor/interfaz antes de confiar la integridad de nuestros datos a este comportamiento por omisión.



La prueba que realizaremos es muy sencilla, y nos podemos valer indistintamente del Database Explorer, extrae sus datos a través del BDE, o de la Consola de InterBase. Utilizaré una base de datos de InterBase que viene con los ejemplos de Delphi; pero usted puede usar cualquier otra base de datos SQL equivalente. Mi base de datos está almacenada en el siguiente fichero:

*C:\Archivos de programa\Archivos comunes\Borland Shared\Data\mastsql.gdb*

Primero nos conectamos a la base de datos y creamos el siguiente procedimiento almacenado:

```
create procedure Transferencia(Ordenante int, Beneficiario int,
    Cantidad double precision) as
begin
    update Employee
    set    Salary = Salary + :Cantidad
    where EmpNo = :Beneficiario;
    update Employee
    set    Salary = Salary - :Cantidad
    where EmpNo = :Ordenante;
end ^
```

Evidentemente, no se trata de una transferencia bancaria “real” (le estamos bajando el salario a un empleado para subírselo a otro) pero nos bastará para lo que pretendemos demostrar. Ahora modifiquemos las reglas sobre la tabla de empleados, de modo que un empleado no pueda tener un salario negativo (¡ya quisieran algunos jefes!):

```
alter table Employee
add constraint SalarioPositivo
check (Salary > 0);
```

Intente ahora una “transferencia de salario” que deje con salario negativo al primer empleado. Por ejemplo, puede ejecutar la siguiente instrucción desde el propio *Windows ISQL*, si la base de datos de ejemplos no ha sido aún modificada:

```
execute procedure Transferencia(2, 4, 100000)
```

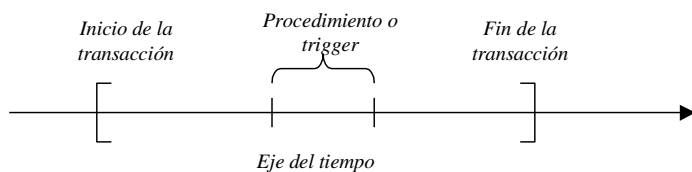
Naturalmente, como el empleado número 2 no tiene ese salario, la segunda instrucción **update** del procedimiento falla, al no cumplirse la restricción sobre el salario positivo. Pero lo importante es que si listamos los empleados, ¡nos encontraremos que el registro del empleado número 4 no ha sido modificado, a pesar de que aparentemente la primera instrucción se pudo ejecutar exitosamente!

Tome nota de la siguiente característica de InterBase: cada ejecución de un procedimiento o de un trigger inicia una mini-transacción. Si ya hay alguna transacción explícita se trata del único caso en que InterBase permite alguna forma de *transacción anidada*. Si durante la ejecución del procedimiento o trigger se produce una excepción, todas las acciones realizadas en la mini-transacción se anulan. Observe, sin embargo que si ya hay una transacción iniciada, no se deshacen automáticamente las

acciones previas a la ejecución del procedimiento. Asegúrese de iniciar una transacción limpia, y en primer lugar, ejecute alguna tontería como la siguiente:

```
update EMPLOYEE
set   FirstName = 'Kimberley'
where FirstName = 'Kim'
```

Acto seguido, ejecute el mismo procedimiento *Transferencia*. Cuando se produzca el inevitable error, compruebe que todas las *Kim* siguen llamándose *Kimberley*, pero la primera actualización de *Transferencia* ha sido cancelada. El siguiente diagrama puede ayudarnos a aclarar la idea:



¿Y el resto de los servidores? He realizado la prueba con Oracle 8 y con Microsoft SQL Server 7 y 2000. En el CD-ROM que acompaña al libro se incluyen *scripts* sencillos para que el lector repita el experimento, si lo desea. Resulta que Oracle se comporta de forma similar a InterBase, ¡pero SQL Server no! En este sistema, la primera instrucción se ejecuta sin problemas, pero cuando la segunda falla, las modificaciones realizadas por la primera permanecen en la base de datos.

Hechos de tal calibre merecen que extraigamos una moraleja. Aunque en teoría Delphi trata de forma similar a todos los servidores de datos, en la práctica encontramos diferencias como la que acabamos de presentar, que no hace imposible la programación independiente del formato de datos, pero que nos obliga a prestar suma atención a estos detalles.

## Niveles de aislamiento de transacciones

Las propiedades atómicas de las transacciones deben cumplirse tanto en sistemas multipuestos como en sistemas con un solo usuario. Cuando es posible el acceso simultáneo a una base de datos por varios usuarios o aplicaciones (que pueden residir en la misma máquina), hay que tener en cuenta la forma en que estas transacciones se comportan colectivamente. Para poder garantizar la consistencia de la base de datos cuando varias transacciones se ejecutan concurrentemente sobre la misma, deben respetarse las siguientes condiciones:

- Una transacción no debe poder leer datos grabados por otra transacción mientras ésta no haya finalizado.
- Los datos leídos por una transacción deben mantenerse constantes hasta que la transacción que los ha leído finalice.

La primera condición es evidente: no podemos tomar una decisión en base a un dato que ha sido colocado tentativamente, que no sabemos aún si viola o no las reglas de consistencia de la base de datos. Solamente las transacciones transforman la base de datos de un estado consistente a otro; pero sólo aquellas transacciones que terminan exitosamente. ¿Elemental, no? Pues ni Paradox ni dBase aíslan a una transacción de cambios no confirmados efectuados desde otros puestos. Comprenderá que esto deja mucho margen para el desastre...

En cuanto al segundo requerimiento, ya hemos hablado acerca de él cuando contábamos la historia de la transferencia bancaria de Albert Einstein. La condición que estamos imponiendo se denomina frecuentemente *lecturas repetibles*, y está motivada por la necesidad de partir de un estado consistente para poder llegar sensatamente a otro estado similar. La violación de esta condición da lugar a situaciones en que la ejecución consecutiva de dos transacciones da un resultado diferente a la ejecución concurrente de las mismas. A esto se le llama, en la jerga académica, el criterio de *serializabilidad* (mi procesador de textos protesta por la palabra, pero yo sé Informática y él no).

En realidad, el criterio de serializabilidad implica también que se prohíba la aparición de *filas fantasmas*: un registro insertado por otra transacción no puede aparecer en el campo de visión de una transacción ya iniciada. Pero la mayoría de los sistemas con los que trataremos logran la serializabilidad a la par que las lecturas repetibles.

Considere una aplicación que lee un registro de una tabla para reservar asientos en un vuelo. Esta aplicación se ejecuta concurrentemente en dos puestos diferentes. Llego a uno de los puestos y facturo mi equipaje; frente al otro terminal se sitúa (¡oh, sorpresa!) Pamela Anderson<sup>13</sup>. Antes de comenzar nuestras respectivas transacciones, el registro que almacena la última plaza disponible del vuelo a Tahití contiene el valor 1 (hemos madrugado). Mi transacción lee este valor en la memoria de mi ordenador. Pam hace lo mismo en el suyo. Me quedo atontado mirando a la chica, por lo cual ella graba un 2 en el registro, termina la transacción y se marcha. Regreso a la triste realidad y pulso la tecla para terminar mi transacción. Como había leído un 1 de la base de datos y no hay transacciones en este momento que estén bloqueando el registro, grabo un *dos*, suspiro y me voy. Al montar en el avión descubro que Pam y yo viajamos en el mismo asiento, y uno de los dos tiene que ir sobre las piernas del otro. Esta es una situación embarazosa; para Pamela, claro está.

¿Demasiada verbosa la explicación? Intentémoslo otra vez, pero con menos poesía. Hay una aplicación que ejecuta el siguiente algoritmo:

```
Leer Valor
Valor := Valor + 1
Escribir Valor
```

<sup>13</sup> No sea excesivamente cruel; escribí este párrafo hace mucho tiempo. *Sic transit gloria mundi*.

Está claro que si ejecutamos este algoritmo dos veces, una a continuación de la otra, como resultado, la variable *Valor* debe incrementarse en dos. Pero cuando la ejecución no es secuencial, sino concurrente, las instrucciones elementales que componen el algoritmo pueden entremezclarse. Una de las muchas formas en que puede ocurrir esta mezcla es la siguiente:

<pre>Leer Valor (1) (...espera...) Valor := Valor + 1 (2) (...espera...) (...espera...) Escribir Valor (2)</pre>	<pre>--- Leer Valor (1) (...espera...) Valor := Valor + 1 (2) Escribir Valor (2) ---</pre>
--	--

El resultado final almacenado en *Valor* sería 2, en vez del correcto 3. El problema se presenta cuando la segunda transacción escribe el valor. La primera sigue asumiendo que este valor es 1, pues fue lo que leyó al principio. Si en este momento, el algoritmo relejera la variable, se encontraría con que su contenido ha cambiado. Piense en la vida real: usted saca su billetera, cuenta sus riquezas y se entera de que tiene dos billetes de 100 euros. Un minuto más tarde va a pagar en una tienda, saca la billetera y descubre que uno de los billetes se ha evaporado en la zona crepuscular.

En conclusión, todo sistema de bases de datos debería implementar las transacciones de forma tal que se cumplan las dos condiciones antes expuestas. Pero una implementación tal es algo costosa, y en algunas situaciones se puede prescindir de alguna de las dos condiciones, sobre todo la segunda (y no lo digo por la señorita Anderson). En el estándar del 92 de SQL se definen tres niveles de aislamiento de transacciones, en dependencia de si se cumplen las dos condiciones, si no se cumplen las lecturas repetibles, o si no se cumple ninguna de las condiciones.

Del mismo modo que los métodos de activación y fin de transacciones se implementan como métodos de los componentes de conexión en Delphi, estos mismos componentes son también responsables de configurar el nivel de aislamiento. Por ejemplo, el componente *TDatabase* del BDE prevé tres niveles de aislamiento, que se configuran en la propiedad *TransIsolation*:

Constante	Nivel de aislamiento
<i>tiDirtyRead</i>	Lee cambios sin confirmar
<i>tiReadCommitted</i>	Lee solamente cambios confirmados
<i>tiRepeatableRead</i>	Los valores leídos no cambian durante la transacción

El valor por omisión de *TransIsolation* es *tiReadCommitted*. Aunque el valor almacenado en esta propiedad indique determinado nivel de aislamiento, es prerrogativa del sistema de bases de datos subyacente el aceptar ese nivel o forzar un nivel de aislamiento superior. Por ejemplo, no es posible (ni necesario o conveniente) utilizar el nivel *tiDirtyRead* sobre bases de datos de InterBase. Si una de estas bases de datos se configura para el nivel *tiDirtyRead*, InterBase establece la conexión mediante el nivel *tiReadCommitted*. Sin embargo, la implementación actual de las transacciones locales

sobre tablas Paradox y dBase solamente admite el nivel de aislamiento *tiDirtyRead*; cualquier otro nivel es aceptado, pero si intentamos iniciar una transacción sobre la base de datos, se nos comunicará el problema.

## Registros de transacciones y bloqueos

¿Cómo logran los sistemas de gestión de bases de datos que dos transacciones concurrentes no se estorben entre sí? La mayor parte de los sistemas, cuya arquitectura está basada en el arquetípico System R, utilizan técnicas basadas en bloqueos. Cuando estudiemos el BDE, descubriremos que Paradox y dBase utilizan también bloqueos para garantizar el acceso exclusivo a registros, implementando un control de concurrencia pesimista. Sin embargo, aunque también se trata de “bloqueos”, el significado de los mismos es bastante diferente al que tienen en los sistemas SQL. Ahora veremos solamente cómo se adapta este mecanismo de sincronización a la implementación de transacciones.

En primer lugar, ¿cómo evitar las “lecturas sucias”? Existen dos técnicas básicas. La más sencilla consiste en “marcar” el registro que modifica una transacción como “sucio”, hasta que la transacción confirme o anule sus grabaciones. A esta marca es a lo que se le llama “bloqueo”. Si otra transacción intenta leer el valor del registro, se le hace esperar hasta que desaparezca la marca sobre el registro. Por supuesto, esta política se basa en un comportamiento “decente” por parte de las transacciones que modifican registros: no deben dejarse cambios sin confirmar o anular por períodos de tiempo prolongados.

Ahora bien, ¿cómo es que la transacción que realiza el cambio restaura el valor original del registro si se decide su anulación? Lo más frecuente, en los sistemas inspirados por System R, es encontrar implementaciones basadas en *registros de transacciones* (*transaction logs*<sup>14</sup>). Estos registros de transacciones son ficheros en los cuales se graban secuencialmente las operaciones necesarias para deshacer las transacciones no terminadas. Algunos guardan en el *log* el valor original; otros, por el contrario, guardan el nuevo valor no confirmado, y lo transfieren a la base de datos solamente al confirmarse la transacción. En la teoría de bases de datos, hablaríamos de *undo* y *redo logs* (*registros para deshacer o rehacer*). Cada técnica tiene sus ventajas y desventajas: por ejemplo, si se utiliza un registro de rehacer y alguien corta la corriente, la base de datos no queda afectada, y al volver a encender el sistema podemos seguir trabajando sobre un estado consistente. Sin embargo, la aplicación de los cambios durante la confirmación es sumamente peligrosa, y el implementador debe tomar precauciones extraordinarias.

---

<sup>14</sup> La palabra inglesa *log* quiere decir literalmente “leño”, pero en este contexto se refiere al “cuaderno de bitácora”, en que los capitanes de navío anotaban las incidencias del viaje. En la gloriosa época de la piratería, una de las principales anotaciones era la velocidad de la nave. Para medirla, se arrojaba un *leño* al agua por la proa, y se calculaba el tiempo que tardaba en llegar a la popa. De no haber un madero a mano, se arrojaba al cocinero, con el inconveniente de que el pateo de éste podía distorsionar los resultados.

Esto nos da una idea para otra política de acceso: si una aplicación, al intentar leer un registro, encuentra que ha sido modificado, puede ir a buscar el valor original. Si utilizamos un *redo log*, el valor es el que se encuentra dentro del propio registro. En caso contrario, hay que buscarlo en el registro de transacciones. Por supuesto, esta técnica es superior a la anterior, pues ofrece mejores tiempos de acceso incluso en caso de modificaciones frecuentes.

Paradox y dBase implementan un *undo log*, pero cuando una aplicación lee un registro modificado por otra transacción, no se toma la molestia de buscar el valor original en el *log file*. Por este motivo es que aparecen lecturas sucias.

Todos los bloqueos impuestos por una transacción, por supuesto, son liberados al terminar ésta, ya sea confirmando o anulando.

El fichero con el registro de transacciones es una estructura difícil de mantener. Muchos sistemas lo utilizan para la recuperación de bases de datos estropeadas por fallos físicos. Suponga que la última copia de seguridad de su base de datos se realizó el domingo, a medianoche. El lunes por la tarde ocurre lo impensable: encima de su oficina van a montar un bingo, inician las obras de remodelado y las vibraciones de la maquinaria estropean físicamente su disco duro. ¿No se lo cree? Pues a mí me pasó. Supongamos que hemos tenido suerte, y solamente se ha perdido el fichero que contiene la base de datos, pero que el fichero con el *log* está intacto. Entonces, a partir de la última copia y de las transacciones registradas en el fichero, pueden restablecerse los cambios ocurridos durante la jornada del lunes.

Para no tentar a la suerte, los cursos para administradores de bases de datos recomiendan que los ficheros de datos y de transacciones estén situados en discos físicos diferentes. No sólo aumenta la seguridad del sistema, sino que además mejora su eficiencia. Medite en que cada modificación tiene que escribir en ambas estructuras, y que si hay dos controladores físicos para atender a las operaciones de grabación, mejor que mejor.

## Lecturas repetibles mediante bloqueos

Las técnicas expuestas en la sección anterior no garantizan la repetibilidad de las lecturas dentro de una transacción. ¿Recuerda el ejemplo en que dos transacciones incrementaban una misma variable? Estas eran las instrucciones finales de la secuencia conflictiva:

(...espera...)	<b>Escribir</b> Valor (2)
<b>Escribir</b> Valor (2)	---

El problema es que la segunda transacción (la primera en escribir) libera el bloqueo en cuanto termina, lo que sucede antes de que la primera transacción intente su grabación. Eso... o que ambas transacciones intentan bloquear demasiado tarde. En realidad, hay una solución draconiana a nuestro alcance: todas las transacciones de-

ben anticipar qué registros van a necesitar, para bloquearlos directamente al iniciarse. Claro está, los bloqueos también se liberan en masa al terminar la transacción. A esta estrategia se le denomina *bloqueo en dos fases* (*two-phase locking*) en los libros de texto, pero no es recomendable en la práctica. En primer lugar, la previsión de las modificaciones no es siempre posible mediante algoritmos. Y en segundo lugar, aún en los casos en que es posible, limita demasiado las posibilidades de concurrencia de las transacciones.

Para una solución intermedia necesitaremos, al menos, dos tipos de bloqueos diferentes: de *lectura* y de *escritura* (*read locks/write locks*). La siguiente tabla aparece en casi todos los libros de teoría de bases de datos, y muestra la compatibilidad entre estos tipos de bloqueos:

	Lectura	Escritura
Lectura	Concedido	Denegado
Escritura	Denegado	Denegado

En la nueva estrategia, cuando una transacción va a leer un registro, coloca primero un bloqueo de lectura sobre el mismo. De acuerdo a la tabla anterior, la única posibilidad de que este bloqueo le sea denegado es que el registro esté bloqueado en modo de escritura por otra transacción. Y esto es necesario que sea así para evitar las lecturas sucias. Una vez concedido el bloqueo, las restantes transacciones activas pueden leer los valores de este registro, pues se les pueden conceder otros bloqueos de lectura sobre el mismo. Pero no pueden modificar el valor leído, pues lo impide el bloqueo impuesto por la primera transacción. De este modo se garantizan las lecturas repetibles.

Quizás sea necesaria una aclaración: la contención por bloqueos se aplica entre transacciones diferentes. Una transacción puede colocar un bloqueo de lectura sobre un registro y, si necesita posteriormente escribir sobre el mismo, promover el bloqueo a uno de escritura sin problema alguno. Del mismo modo, un bloqueo de escritura impuesto por una transacción no le impide a la misma realizar otra escritura sobre el registro más adelante.

Por supuesto, todo este mecanismo se complica en la práctica, pues hay que tener en cuenta la existencia de distintos niveles de bloqueos: a nivel de registro, a nivel de página y a nivel de tabla. Estos niveles de bloqueos están motivados por la necesidad de mantener dentro de un tamaño razonable la tabla de bloqueos concedidos por el sistema. Si tenemos un número elevado de bloqueos, el tiempo de concesión o negación de un nuevo bloqueo estará determinado por el tiempo de búsqueda dentro de esta tabla. ¿Recuerda el ejemplo de la transferencia bancaria de Albert Einstein versus la suma de los saldos? La aplicación que suma los saldos de las cuentas debe imponer bloqueos de lectura a cada uno de los registros que va leyendo. Cuando la transacción termine habrá pedido tantos bloqueos como registros tiene la tabla, y esta cantidad puede ser respetable. En la práctica, cuando el sistema detecta que una transacción

posee cierta cantidad de bloqueos sobre una misma tabla, trata de promover de nivel a los bloqueos, transformando la multitud de bloqueos de registros en un único bloqueo a nivel de tabla.

Sin embargo, esto puede afectar la capacidad del sistema para hacer frente a múltiples transacciones concurrentes. Tal como hemos explicado en el ejemplo anterior, la transacción del físico alemán debe fallar, pues el registro de su cuenta bancaria ya ha sido bloqueado por la transacción que suma. No obstante, una transferencia entre Isaac Newton y Erwin Schrödinger debe realizarse sin problemas, pues cuando la transacción sumadora va por el registro de Mr. Marteens, los otros dos registros están libres de restricciones. Si, por el contrario, hubiéramos comenzado pidiendo un bloqueo de lectura a nivel de tabla, esta última transacción habría sido rechazada por el sistema.

Se pueden implementar también otros tipos de bloqueo además de los clásicos de lectura y escritura. En particular, las políticas de bloqueo sobre índices representados mediante árboles balanceados son bastante complejas, si se intenta maximizar el acceso concurrente a los datos.

## **Variaciones sobre el tema de bloqueos**

¿Qué sucede cuando una transacción pide un bloqueo sobre un registro y encuentra que está ocupado temporalmente? Si consultamos los libros clásicos sobre teoría de bases de datos, veremos que casi todos asumen que la aplicación debe esperar a que el sistema pueda concederle el bloqueo. El hecho es, sin embargo, que la mayoría de estos autores se formaron y medraron en el oscuro período en que los ordenadores se programaban mediante tarjetas perforadas, preferiblemente a mano. Para ejecutar una aplicación, había que rellenar montones de formularios para el Administrador del Centro de Cálculo, y lo más probable es que el dichoso programa se ejecutara mientras nosotros no estábamos presentes. Por lo tanto, no se podía contar con la intervención del usuario para resolver el conflicto de intereses entre aplicaciones que luchaban por un mismo registro.

La otra solución extrema al problema del conflicto al bloquear es devolver inmediatamente un error a la aplicación cliente. Aunque parezca mentira a primera vista, ésta es la solución más flexible, pues el programador puede decidir si la aplicación debe esperar y reintentar, o si debe anular inmediatamente la transacción y dedicarse a otra cosa. Y lo más sensato que puede hacer, en esta época de la Informática Interactiva, es pasarle la patata caliente al usuario para que decida.

¿Cómo se comporta cada sistema específico de los que vamos a tratar? Depende, en muchos casos, de cómo está configurada la interfaz de acceso. Por ejemplo, si utilizamos el BDE, Oracle espera indefinidamente a que el bloqueo conflictivo se libere. SQL Server e Informix permiten configurar en el BDE el tiempo que debe esperar una transacción antes de dar por fallido el intento de bloqueo. Para Informix, hay



que ajustar el parámetro *LOCK MODE*; con SQL Server, es necesario configurar *LOCK MODE*. Si utilizamos ADO con SQL Server, tenemos varias propiedades para configurar el tiempo de espera, como *CommandTimeout*, que se encuentra en el componente de conexión *TADOConnection*, y en uno de los conjuntos de datos, en el componente *TADODataSet*.

En cuanto a InterBase, si la versión del SQL Link es anterior a la 5.0.1.23, se genera una excepción inmediatamente al encontrar un registro bloqueado (no se alarme, espere a leer el resto del capítulo). En realidad, el API de bajo nivel de InterBase puede también soportar directamente el modo de espera, y esto se ha implementado en las versiones más recientes del SQL Link, por medio de un parámetro llamado *WAIT ON LOCKS*. Si utilizamos DB Express, tenemos también un parámetro *WaitOnLocks* en el controlador de InterBase.

Otro problema que se presenta en relación con los bloqueos es conocido como *abrazo mortal*, o *deadlock*. Hay dos niños en una guardería, y una moto y una escopeta de juguete. El primero balbucea: “yo quiero la moto”; y el segundo: “yo quiero la escopeta”. Pero si el objetivo de estos precoces chavales es ir de *easy riders*, lo van a tener complicado, pues no van a obtener la otra mitad del atuendo. Uno de ellos tendrá que renunciar, por lo que será necesaria la actuación de la domadora de la guardería. Eso, por supuesto, si nuestros niños son de los que esperan indefinidamente a que se libere el “bloqueo” sobre el juguete deseado. Si son de los que se rinden a la primera (¡es lo preferible en este caso!), en cuanto el primero solicita la escopeta y ve que está en uso, cambia de juego y deja al segundo en libertad para ir por las carreteras violando la ley.

Existe una sencilla receta para minimizar la aparición de abrazos mortales, y que es de particular importancia en aquellos sistemas que hacen esperar a las aplicaciones por los bloqueos que solicitan:

*“Obligue a las transacciones a bloquear siempre en el mismo orden”*

Por ejemplo, “moto” y “escopeta”. Enséñeles el alfabeto a los niños, para que cuando pidan varios juguetes, lo hagan en orden alfabético (¿no es mucho pedir para un *easy rider*?). Otro ejemplo, extraído del así llamado mundo real: en un sistema de entrada de pedidos hay que actualizar las existencias en almacén de los artículos contenidos en un pedido. Entonces, lo más sensato es ordenar las actualizaciones de acuerdo al código o al nombre del artículo. Así se evitan abrazos mortales entre pedidos que venden C++ Builder/Delphi, y Delphi/C++ Builder.

## El jardín de los senderos que se bifurcan

InterBase resuelve los problemas de concurrencia con una técnica diferente a las utilizadas por los demás sistemas de bases de datos. Si le atrae la ciencia ficción, los

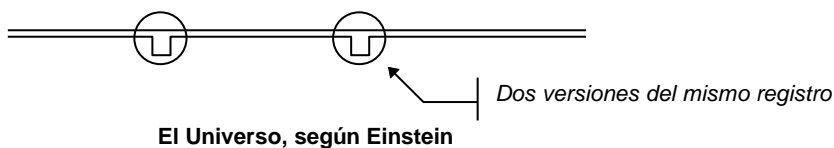
viajes en el tiempo y la teoría de los mundos paralelos, le gustará también la siguiente explicación.

Volvamos al ejemplo de las maniobras financieras de la comunidad internacional de físicos, suponiendo esta vez que el sistema de base de datos empleado es InterBase. Este día, Mr. Marteens, que no es físico ni matemático, sino banquero (por lo tanto, un hombre feliz), llega a su banco temprano en la mañana. Para Marteens levantarse a las nueve de la mañana es madrugar, y es a esa hora que inicia una transacción para conocer cuán rico es. Recuerde esta hora: las nueve de la mañana.

A las nueve y diez minutos se presenta Albert Einstein en una de las ventanas de la entidad a mover los famosos diez mil dólares de su cuenta a la cuenta de Newton. Si Ian Marteens no hubiese sido programador en una vida anterior y hubiera escogido para el sistema informático de su banco un sistema de gestión implementado con bloqueos, Einstein no podría efectuar su operación hasta las 9:30, la hora en que profetizamos que terminará la aplicación del banquero. Sin embargo, esta vez Albert logra efectuar su primera operación: extraer el dinero de su cuenta personal. Nos lo imaginamos pasándose la mano por la melena, en gesto de asombro: ¿qué ha sucedido?

Bueno, mi querido físico, ha sucedido que el Universo de datos almacenados se ha dividido en dos mundos diferentes: el mundo de Marteens, que corresponde a los datos existentes a las nueve de la mañana, y el mundo de Einstein, que acusa todavía un faltante de \$10.000, pues la transacción no ha terminado. Para que esto pueda ocurrir, deben existir dos versiones diferentes de la cuenta bancaria de Einstein, una en cada Universo. La versión del físico es todavía una versión tentativa; si la señora Einstein introduce la tarjeta en un cajero automático para averiguar el saldo de la cuenta de su marido, no tendrá acceso a la nueva versión, y no se enterará de las locuras inversionistas de su cónyuge. En este punto insistiremos más adelante.

#### El Universo, según Marteens



#### El Universo, según Einstein

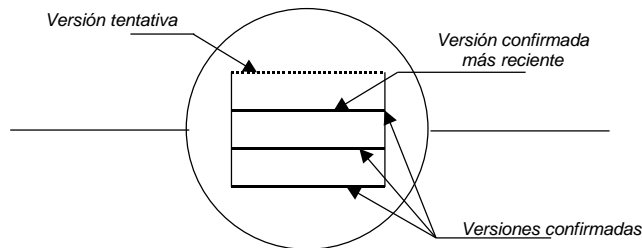
Algo parecido sucederá cuando Einstein modifique la cuenta de Newton, incrementándola en la cantidad extraída. Esta vez también se creará una nueva versión que no será visible para la transacción de las 9:00. *Ni siquiera cuando Einstein confirme la transacción.* La idea es que cada transacción solamente ve el mundo tal como era en el momento en que se inicia. Es como si cada transacción sacara una copia local de los datos que va a utilizar. De hecho, hay algunos sistemas que utilizan técnicas parecidas de *replicación*, para garantizar las lecturas repetibles. InterBase *no* hace esto. InterBase

saca copias de la parte de la base de datos afectada por actualizaciones concurrentes; de esta manera, se mantiene la base de datos dentro de un tamaño razonable.

Los problemas de este enfoque se producen cuando los Universos deben volver a sincronizarse. Por ejemplo, ¿qué sucede cuando Einstein confirma su transacción? Nada: siguen existiendo dos versiones de su cuenta. La más reciente es la modificada, y si alguna transacción comienza después de que esta confirmación ocurra, la versión que verá es la grabada por Einstein. La versión de las 9:00 existe solamente porque hay una transacción que la necesita hasta las 9:30; a partir de ese momento, pierde su razón de ser y desaparece.

Pero no siempre las cosas son tan fáciles. Mientras Einstein realizaba su transferencia, Newton, que hacía las compras en el supermercado (hay algunos que abren muy temprano), intentaba pagar con su tarjeta. Iniciaba una transacción, durante la cual extraía dinero de su cuenta; Newton no puede ver, por supuesto, los cambios realizados por Einstein, al no estar confirmada la transacción. En este caso, Newton no puede modificar el registro de su cuenta, pues InterBase solamente permite una versión sin confirmar por cada registro.

Es útil, por lo tanto, distinguir entre versiones “tentativas”, que pertenecen a transacciones sin confirmar, y versiones “definitivas”, que pertenecen a transacciones ya confirmadas:



Solamente puede haber una versión tentativa por registro. Esta restricción actúa, desde el punto de vista de las aplicaciones clientes, exactamente igual que un bloqueo de escritura a nivel de registro. Cuando InterBase detecta que alguien intenta crear una versión de un registro que ya tiene una versión tentativa, lanza el siguiente mensaje de error; extraño y confuso, según mi humilde opinión:

*“A deadlock was detected”*

Sin embargo, sí se permiten varias versiones confirmadas de un mismo registro. Si una aplicación modifica un registro, y hay otra transacción activa que ha leído el mismo registro, la versión antigua se conserva hasta que la transacción desfasada culmina su ciclo de vida. Evidentemente, se pueden acumular versiones obsoletas, pero en cualquier caso, cuando se conecta una nueva transacción, siempre recibe la versión confirmada más reciente.

Con las versiones de registros de InterBase sucede igual que con las creencias humanas: no importa si se ha demostrado que tal o más cual creencia es falsa. Siempre que alguna pobre alma tenga fe, el objeto en que se cree “existe”. Y si duda de mi palabra, pregúntele al Sr. Berkeley.

## ¿Bloqueos o versiones?

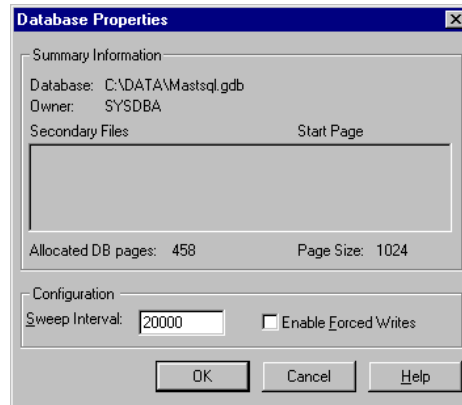
En comparación con las técnicas de aislamiento basadas en la contención (bloqueos), la técnica empleada por InterBase, conocida como *Arquitectura Multigeneracional*, se puede calificar de optimista. Cuando se trata de optimismo y pesimismo en relación con la implementación del aislamiento entre transacciones, las ventajas de una estrategia optimista son realmente abrumadoras. Los sistemas basados en bloqueos han sido diseñados y optimizados con la mente puesta en un tipo de transacciones conocidas como OLTP, de las siglas inglesas *OnLine Transaction Processing* (procesamiento de transacciones en línea). Estas transacciones se caracterizan por su breve duración, y por realizar preferentemente escrituras. Como las transacciones proceden por ráfagas, y cada una involucra a unos pocos registros, la posibilidad de un conflicto entre un par de ellas es pequeña. Como ejemplo práctico, piense en cómo funciona una base de datos que alimenta a una red de cajeros automáticos. Evidentemente, la técnica de bloqueos a nivel de registro funciona estupendamente bajo estas suposiciones. Y lo que también es de notar, la técnica optimista también da la talla en este caso.

Sin embargo, existen otros tipos de transacciones que estropean la fiesta. Son las utilizadas por los sistemas denominados *DSS: Decision Support Systems*, o sistemas de ayuda para las decisiones. El ejemplo de la transacción que suma los saldos, y que hemos utilizado a lo largo del capítulo, es un claro ejemplar de esta especie. También las aplicaciones que presentan gráficos, estadísticas, que imprimen largos informes... Estas transacciones se caracterizan por un tiempo de vida relativamente prolongado, y por preferir las operaciones de lectura.

¿Otro ejemplo importante?, el proceso que realiza la copia de seguridad de la base de datos. ¡La transacción iniciada por este proceso debe tener garantizadas las lecturas repetibles, o podemos quedarnos con una copia inconsistente de la base de datos!

En un sistema basado en bloqueos las transacciones OLTP y DSS tienen una difícil coexistencia. En la práctica, un sistema de este tipo debe “desconectar” la base de datos para poder efectuar la copia de seguridad (“Su operación no puede efectuarse en estos momentos”, me dice la verde pantalla de mi cajero). De hecho, uno de los objetivos de técnicas como la replicación es el poder aislar físicamente a las aplicaciones de estos dos tipos entre sí. Sin embargo, InterBase no tiene ninguna dificultad para permitir el uso consistente y simultáneo de ambos tipos de transacciones. Esta clase de consideraciones condiciona muchas veces el rendimiento de un sistema de bases de datos.

En contraste con los sistemas basados en bloqueos, que necesitan un *log file* para el control de atomicidad, en una arquitectura como la de InterBase, las propias versiones de las filas modificadas son la única estructura necesaria para garantizar la atomicidad. En realidad, hasta que una transacción finaliza, las versiones modificadas representan datos “tentativos”, no incorporados a la estructura principal de la base de datos. Si la base de datos falla durante una transacción, basta con reiniciar el sistema para tener la base de datos en un estado estable, el estado del que nunca salió. Las versiones tentativas se convierten en “basura”, que se puede recoger y reciclar.



Y este es el inconveniente, un inconveniente menor en mi opinión, de la arquitectura multigeneracional: la necesidad de efectuar de vez en cuando una operación de recogida de basura (*garbage collection*). Esta operación se activa periódicamente en InterBase cada cierto número elevado de transacciones, y puede coexistir con otras transacciones que estén ejecutándose simultáneamente. También se puede programar la operación para efectuarla en momentos de poco tráfico en el sistema; la recogida de basura consume, naturalmente, tiempo de procesamiento en el servidor.



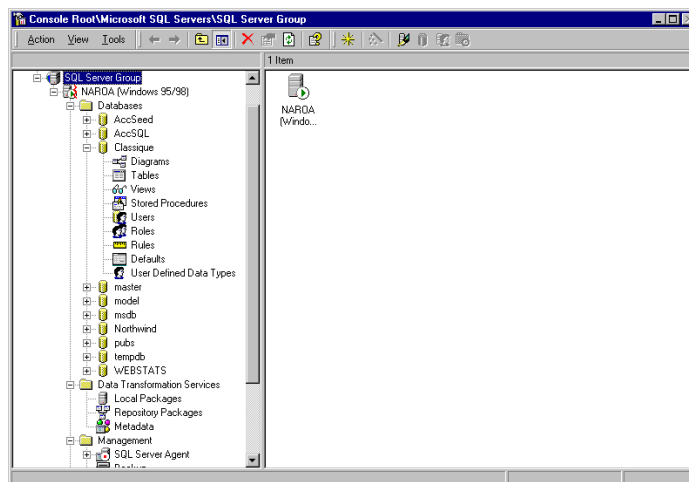
# Microsoft SQL Server

**H**ABÍA UNA VEZ UN PATITO MUY FEO, tan feo que casi nadie podía adivinar el cisne en que podía convertirse. Seamos honestos y contemos bien la historia: era un pato de familia rica. Cuando Papá Pato comprobó resignado el poco éxito que tenía su crío con las patas, decidió invertir una considerable suma de dinero para cambiarle el aspecto. Y lo logró. Porque da igual pato blanco o pato negro: lo importante es que cace ratones (de pato a gato hay una sola letra).

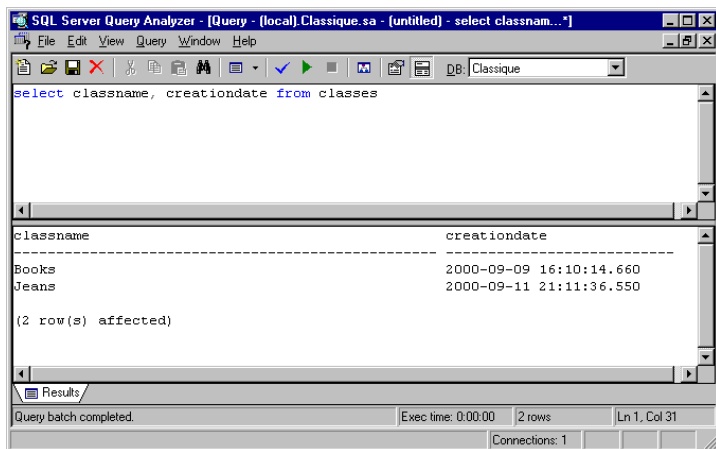
En el presente capítulo describiremos principalmente las versiones 7 y 2000 de SQL Server, pero también trataremos un poco, más que nada por inercia, de la versión 6.5, anterior a la cirugía estética.

## Herramientas de desarrollo en el cliente

La herramienta adecuada para diseñar y administrar bases de datos de MS SQL Server se llama *SQL Enterprise Manager*. Normalmente, esta aplicación se instala en el servidor de datos, pero podemos también instalarla en el cliente. Con el Enterprise Manager podemos crear bases de datos, usuarios, administrar contraseñas, e incluso gestionar otros servidores de forma remota.



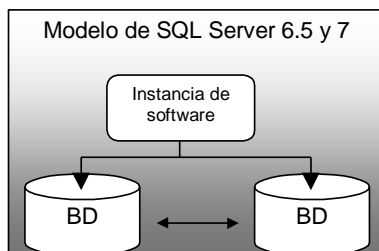
Si lo que necesitamos es ejecutar consultas individuales o todo un *script* con instrucciones, el arma adecuada es el *SQL Query Tool*, que puede ejecutarse como aplicación independiente, o desde un comando de menú de *SQL Enterprise Manager*, como se muestra en la siguiente imagen. Dentro del menú *File* de esta aplicación encontraremos un comando para que ejecutemos nuestros *scripts* SQL.



Hay que reconocer que las herramientas de administración de MS SQL Server clasifican entre las amigables con el usuario. Esto se acentúa a partir de la versión 7, en la cual pegas una patada al servidor y saltan cinco o seis asistentes (*wizards*) para adivinar qué es lo que quieres realmente hacer.

## Un servidor, muchas bases de datos

Las dos palabras más utilizadas en este libro son: *servidor* y *bases de datos*. Parece trivial, pero su significado exacto varía de acuerdo al sistema de bases de datos. En general, con *servidor* nos referiremos al software que se encuentra en ejecución en un ordenador determinado, mientras utilizaremos *base de datos* para referirnos a una entidad lógica representada mediante ficheros que es gestionada por el software mencionado.



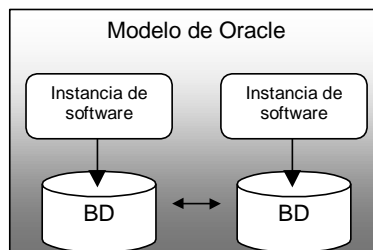
En SQL Server 7, un mismo servidor puede atender directamente varias bases de datos ubicadas físicamente en el propio ordenador. Por lo tanto, una conexión a una base de datos de SQL Server debe especificar dos parámetros: el nombre del servi-



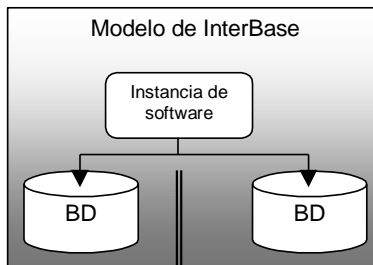
dor y el nombre de la base de datos. Observe la flecha bidireccional que he colocado entre las dos bases de datos. La he añadido para indicar que, al ser gestionadas por una misma instancia de la aplicación servidora, dos bases de datos de SQL Server pueden intercambiar datos directamente entre sí. Se puede, por ejemplo, insertar datos provenientes de una de ellas en la otra sin necesidad de programar algo a medida, y también podemos relacionar sus datos en una misma consulta. Más adelante veremos cómo.

Por supuesto, SQL Server va más allá de la simple comunicación entre dos bases de datos ubicadas en el mismo servidor. También existe la comunicación entre servidores diferentes, como ya mencioné en el capítulo anterior, al tratar sobre el Coordinador de Transacciones Distribuidas.

¿Qué sucede con otros sistemas? El modelo de Oracle, por ejemplo, permite que un servidor gestione una sola base de datos. Dentro de la misma máquina pueden existir y funcionar simultáneamente dos bases de datos ... pero con el coste añadido de tener dos *instancias* del software servidor activas a la vez, una por cada base. La consecuencia es que para conectarnos a una base de datos de Oracle basta con que digamos el nombre del servidor.



La comunicación directa entre bases de datos se logra “enlazando” servidores explícitamente. El sistema es menos directo que el utilizado por MS SQL, pero es igual de sencillo.



El concepto de servidor en InterBase, por el contrario, es similar al de SQL Server: un mismo servidor puede manejar simultáneamente varias bases de datos. Dos bases de datos en un mismo servidor de InterBase, sin embargo, tienen menos posibilidad de comunicación directa que en Microsoft SQL Server. Si necesitamos trasvasar da-

tos de una base de datos a otra, necesitaremos algún tipo de almacenamiento externo, o tendremos que escribir un par de líneas de código en algún lenguaje de propósito general.

#### NOTA

SQL Server 2000 permite ejecutar más de una instancia del servidor en una misma máquina. El sistema de múltiples instancias mantiene todas las ventajas del modelo anterior, pero hace posible además ejecutar dos versiones diferentes de SQL Server en un mismo ordenador. Por ejemplo, puede instalar SQL Server 2000 sin eliminar una instalación existente de la versión 6.5 (debe haberse actualizado con el *Service Pack 5*). O puede ejecutar dos instancias de la versión 2000, una en español y la otra en inglés.

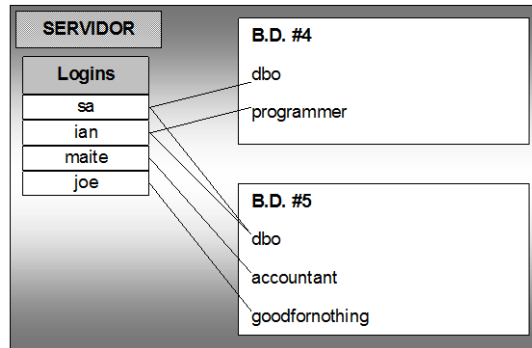
## Dos niveles de identificación

Ya es hora de explicar cómo funciona la seguridad en SQL Server. Lo primero que tenemos que comprender es que, desde una perspectiva práctica, se trata de un sistema organizado en dos niveles fundamentales. El primero de estos niveles tiene que ver con el inicio de sesión, es decir, con el establecimiento de una conexión con el servidor. En este nivel, SQL Server nos exige que proporcionemos un nombre de *login* (traducido como *inicio de sesión* en la versión del producto en castellano) cada vez que intentemos acceder a un servidor para cualquier propósito. Un poco más adelante estudiaremos en qué consisten estos *logins*, y cómo podemos integrarlos con las cuentas del propio Windows NT.

Pero los *logins* se asocian realmente a permisos sobre la conexión, no a permisos sobre operaciones en bases de datos concretas. En este segundo nivel, SQL Server nos obliga a definir *usuarios* (*users*) que varían para cada una de las bases de datos gestionada por un mismo servidor. Existe, por lo tanto, una especie de matriz cuyas dimensiones son, por una parte, los *logins*, y por la otra las bases de datos, y en cada celda se almacena un nombre de usuario.

Por ejemplo, puede que en cierto servidor deba conectarme como *ian*; no hablaré de contraseñas en este momento. Hay cinco bases de datos en este servidor, y el administrador no me da derechos de conexión para tres de ellas. Siguiendo mi símil de la matriz, esto lo lograría dejando tres celdas en blanco.

En una de las dos bases de datos restantes, al inicio de sesión *ian* le corresponde el usuario *programador*. Este usuario tiene determinados derechos sobre ciertas tablas, procedimientos, vistas y demás objetos de la base de datos. Si *maite* es otro inicio de sesión reconocido por el sistema, ya no puede ser reconocido como *programador* por esa misma base de datos.



Se deduce entonces que no es buena idea darle nombre a los usuarios de acuerdo al papel que van a jugar en la base de datos. Para este propósito existe una tercera clase de objetos en SQL Server: los *roles*, que traduciré como *perfiles*. Existen perfiles predefinidos que se relacionan con el sistema o con operaciones generales sobre bases de datos, y podemos también crear los nuestros. Sirven para agrupar permisos y un usuario puede pertenecer a varios si así lo decidimos. Pero no juegan papel alguno en la identificación y autenticación de usuarios.

Sin embargo, he sido yo quien ha creado la quinta base de datos. El servidor me asocia automáticamente con el usuario especial *dbo* (*database owner*, o propietario de la base de datos) de esa base de datos concreta. Así disfruto de todos los permisos imaginables sobre cualquier objeto que se cree en *mí* base de datos. Incluso los que sean creados por *maite*, si es que le permito utilizarla.

De todos modos, el administrador de la base de datos, que se identifica con el inicio de sesión especial *sa*, goza de los mismos privilegios que yo sobre todos los objetos gestionados por su servidor. Lo que sucede en realidad es que el inicio de sesión *sa* se identifica automáticamente con el usuario *dbo* en cualquier base de datos. Se trata, por supuesto, de una excepción a la regla antes mencionada que exige que cada inicio de sesión dentro de una base de datos dada se asocie a un usuario diferente.

## Dos modelos de seguridad

Pero seguimos teniendo una asignatura pendiente. ¿Cómo se identifica una persona ante SQL Server cuando quiere iniciar una sesión? Existen dos formas:

- SQL Server gestiona su propia lista de *logins*, o inicios de sesión. La validación de un inicio de sesión la realiza SQL Server comprobando la contraseña que debe proporcionar el usuario obligatoriamente al intentar una conexión con el servidor.
- SQL Server confía la verificación de identidades a Windows NT/2000. Supone que el sistema operativo ya ha comprobado que el usuario es quien dice ser al

arrancar el ordenador. Por lo tanto, SQL Server utiliza para su lista de *logins* a un conjunto de usuarios del sistema operativo que debemos autorizar previamente.

Al último modelo, se le conoce como *modelo de seguridad integrada*. El primero es el *modelo de seguridad mixto*, porque puede coexistir de todos modos con el segundo. Solamente en las versiones anteriores a la 7 existía la posibilidad de trabajar en exclusiva con *logins* de SQL Server.

#### **NOTA IMPORTANTE**

Si quiere utilizar el modelo de seguridad integrada no es obligatorio, pero sí *muy recomendable*, que su red NT soporte el modelo de dominios. Aunque es posible tener seguridad integrada sobre una red basada en grupos de trabajo de NT, las interacciones entre las cuentas de usuario del sistema operativo y los inicios de sesión que reconocerá SQL Server se complican bastante. Además, una red con dominios bien implementados es generalmente más eficiente que cualquier otra arquitectura de red.

El modelo de seguridad de un servidor puede modificarse, como hemos adelantado hace poco, en el diálogo de las propiedades del servidor, en la página *Security*. Cuando elegimos el modo de seguridad integrada, los nombres de los inicios de sesión corresponden a los nombres de cuentas del sistema operativo *más* el nombre del dominio en el cual se ha definido cada una. El administrador de sistema (*sa*) debe indicar explícitamente a SQL Server a cuáles usuarios de Windows NT/2000 les va a permitir iniciar sesiones en ese servidor.

Como esta tarea puede ser tediosa si existiesen muchas cuentas en un dominio dado, SQL Server también permite autorizar el acceso a grupos completos de usuarios. Aquí “grupo” se refiere a los grupos de usuarios de Windows NT. Para no autorizar el acceso a usuarios puntuales de un grupo autorizado, existe la posibilidad adicional de denegar explícitamente el acceso a cuentas individuales, o incluso a grupos completos.

## **Adiós a los dispositivos**

SQL Server surgió gracias a una adaptación al entorno PC del código del sistema de bases de datos Sybase. Este último es capaz de ejecutarse sobre una amplia variedad de sistemas operativos, por lo cual su código fuente debe evitar dependencias respecto al sistema de archivos nativo. Para lograrlo, las bases de datos de Sybase no se definen directamente sobre ficheros físicos, sino sobre un concepto lógico denominado *dispositivo* ó *device*, y que cada sistema operativo es libre de implementar o adaptar según sus necesidades.

Hasta la versión 6.5, SQL Server siguió utilizando dispositivos como estructuras que contenían las verdaderas bases de datos, aunque el único sistema operativo soportado fuese el omnipresente Windows NT. Esta decisión, sumada a la pereza de no añadir nueva funcionalidad a los dispositivos, fue la causa de varias severas limitaciones en el producto de Microsoft. La más importante: los dispositivos tenían un tamaño está-

tico, y sólo podía modificarse mediante la acción humana directa. Si una base de datos crecía desmesuradamente y no había un Administrador de Sistemas cerca, los usuarios podían perder su trabajo. Y lo mismo sucedía con el registro de transacciones, una estructura a la que volveremos en breve.

Gracias a B.G., Microsoft ha decidido eliminar completamente los dispositivos de la versión 7, y definir las bases de datos directamente sobre ficheros del sistema operativo. Así que, de ahora en adelante, no volveré a mencionar las palabras *dispositivo* y *device*. Para nosotros, ya no existen.

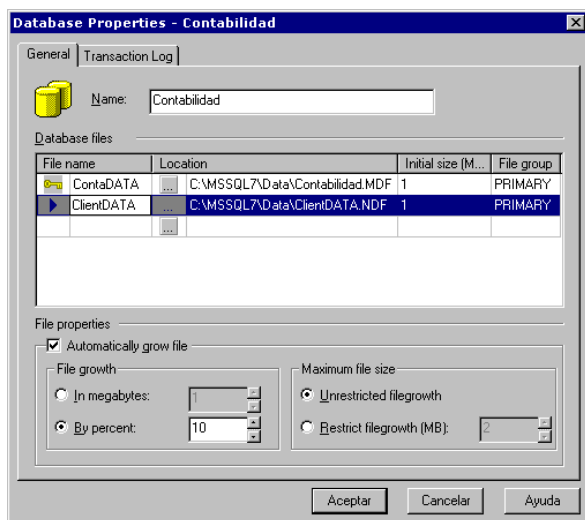
## Los ficheros de una base de datos

Una base de datos de SQL Server está constituida por uno o más ficheros. Como mínimo, se necesita un fichero de datos primario y, aunque el mismo fichero primario puede albergarlo, es conveniente dedicar un fichero separado para el denominado *registro de transacciones*, o *log file*. Pero es posible también utilizar más ficheros para datos o para el registro de transacciones; podemos incluso añadirlos a la base de datos después de que ésta ha sido creada.

Cada fichero de SQL Server 7 tiene las siguientes propiedades:

- **NAME:** contiene un nombre lógico para el fichero.
- **FILENAME:** el nombre del fichero físico. Es el único parámetro que no es opcional, naturalmente.
- **SIZE:** el tamaño inicial del fichero. Más adelante explicaré las unidades empleadas para indicar el tamaño de un fichero.
- **MAXSIZE:** como cada fichero puede crecer según sea necesario, podemos indicar un tamaño máximo. Además de especificar una cifra concreta, podemos utilizar la opción **UNLIMITED**.
- **FILEGROWTH:** el tamaño o porcentaje en el cual crece el fichero cada vez que se expande automáticamente.

Para crear la base de datos podemos utilizar alguna de las herramientas gráficas que ofrece Microsoft, o utilizar instrucciones en modo texto para que sean ejecutadas por el motor de SQL. La siguiente figura muestra el diálogo de creación de bases de datos de *SQL Enterprise Manager*. Personalmente, encuentro más interesante escribir a mano las instrucciones de creación. ¿Motivo?, pues que posiblemente tenga que ejecutarlas varias veces durante el desarrollo y la distribución. Además, así queda documentada inmediatamente la estructura de la base de datos.



En vez de presentar de golpe la sintaxis para la creación de una base de datos, mostraré ejemplos sencillos, como el siguiente:

```
create database Contabilidad
on primary (name = Datos,
    filename = 'c:\conta\contadat.mdf',
    size = 10MB, maxsize = 1000MB,
    filegrowth = 20%)
log on (name = LogFile,
    filename = 'd:\conta\contalog.ldf')
```

Para especificar el tamaño inicial y el máximo de un fichero podemos utilizar los sufijos *KB* y *MB* para indicar kilobytes y megabytes. Para el factor de crecimiento de un fichero se puede especificar una cantidad absoluta, con *KB* y *MB*, pero también es correcto utilizar el sufijo *%* para un porcentaje. En cualquiera de los casos, si se omite la unidad de medida, se asume *MB*.

En el listado anterior, por ejemplo, pedimos 10 megas para el tamaño inicial del fichero de datos, en vez de 1MB, el valor por omisión. De este modo, evitamos reestructuraciones costosas durante la carga inicial de datos.

El hecho de que los ficheros de una base de datos crezcan automáticamente no sirve de excusa para un error frecuente durante las cargas masivas de datos. Si vamos a alimentar una base de datos con un número elevado de registros, la técnica que nos ahorrará más tiempo consiste simplemente en modificar el tamaño de los ficheros existentes para evitar el crecimiento dinámico.

## Particiones y el registro de transacciones

La posibilidad de utilizar distintos ficheros, posiblemente situados en diferentes discos físicos, es una de las características que distinguen a un sistema de bases de datos

profesional de uno para aficionados. El objetivo de esta técnica, conocida como *particionamiento*, es aprovechar el paralelismo que permite la existencias de controladores físicos por separado para cada unidad.

Como veremos, son muchas las puertas que nos abre el particionamiento, por lo cual debemos escoger con mucho cuidado si tenemos la posibilidad. Supongamos que tenemos dos discos duros en nuestro servidor, una situación bastante común. ¿Cómo repartimos los ficheros? Este es un caso de manual: el fichero de datos debe ir en uno de los discos, y el fichero del registro de transacciones (*log file*) en el otro.

El registro de transacciones es una de las estructuras de datos característica de los sistemas relacionales inspirados en el jurásico System R de IBM. Es como un auditor con mala leche: cada vez que se modifica cualquier registro de la base de datos, nuestro ceñudo vigilante realiza un apunte en el registro de transacciones. De esta forma, SQL Server puede devolver la base de datos a su estado original si se aborta una transacción o se cae el sistema. También se puede utilizar este fichero para realizar copias de seguridad diferenciales, llevando a la cinta solamente los cambios realizados desde la última copia.

Si los datos y el registro de transacciones están en discos diferentes, la carga de las lecturas y escrituras se distribuye equitativamente entre ambos discos. Y si se estropea el disco de datos, podemos restaurar la base de datos a partir de la última copia de seguridad, repitiendo las actualizaciones registradas en el *log file*.

## Grupos de ficheros

Pero las técnicas de particionamiento no terminan aquí. SQL Server permite agrupar los ficheros de datos en grupos. Además, a estos grupos se le asignan nombres lógicos, que pueden utilizarse en las instrucciones de creación de tablas e índices para indicar en qué espacio físico deben residir sus datos.

Cuando utilizamos la sintaxis simplificada de creación de bases de datos, como en nuestro primer ejemplo, que no contiene aparentemente indicación alguna sobre grupos de ficheros, estamos realmente utilizando el grupo por omisión, denominado *primary*. El siguiente listado nos enseña cómo crear una base de datos con un grupo adicional al primario, y cómo situar una tabla en dicho grupo.

```
create database CONTABILIDAD
on primary
( name = Datos_1,
  filename = 'c:\mssql7\data\contal.mdf')
filegroup DatosEstaticos
( name = Datos_2,
  filename = 'd:\mssql7\data\conta2.ndf')
log on
( name = LogConta,
  filename = 'd:\mssql7\data\contalog.ldf')
go
```

```

use CONTABILIDAD
go

create table Provincias (
    Codigo    tinyint not null primary key,
    Nombre    varchar(15) not null unique,
    check     (Codigo between 0 and 100),
    check     (Nombre <> '')
) on DatosEstaticos -- ;;ESTA PARTE ES LA QUE NOS INTERESA!!!
go

```

¿Por qué he creado un grupo para los datos estáticos? He seguido asumiendo que tenemos solamente dos discos duros. Al situar el registro de transacciones en el segundo disco hemos mejorado el tiempo de las operaciones de escritura, pero no hemos avanzado nada en lo que concierne a las consultas, que serán mucho más frecuentes.

Por todo lo anterior, he creado un grupo *DatosEstaticos* para que contenga aquellas tablas de nuestra aplicación que son poco propensas al cambio: la tabla de códigos postales y provincias, las de conversión de moneda; incluso los datos de proveedores o bancos con los que trabajamos, si forman un conjunto estable. Cuando realicemos un encuentro (*join*) entre alguna de las sedentarias tablas de *DatosEstaticos* y las ajustadas tablas del grupo primario, volveremos a dividir las operaciones de entrada y salida en partes iguales, entre los dos discos del servidor.

## Varios ficheros en el mismo grupo

Ya sabemos cómo colocar tablas en distintos discos físicos utilizando distintos grupos de ficheros. Pero cada grupo puede contener más de un fichero, como podemos observar a continuación.

```

create database CONTABILIDAD
on primary
( name = Datos_1,
  filename = 'c:\mssql7\data\contal.mdf')
filegroup DatosEstaticos
( name = Datos_2_1,
  filename = 'd:\mssql7\data\conta21.ndf')
( name = Datos_2_2,
  filename = 'd:\mssql7\data\conta22.ndf')
log on
( name = LogConta,
  filename = 'd:\mssql7\data\contalog.ldf')

```

Se nos puede ocurrir con toda lógica: ¿para qué necesitamos entonces varios ficheros dentro de un mismo grupo? La respuesta está en el algoritmo que utilizan los ficheros de un mismo grupo para crecer:

- Si es necesario expandir un grupo de ficheros, posiblemente porque una tabla o índice ha crecido, el espacio necesario se distribuye proporcionalmente entre los ficheros del grupo.



Por lo tanto, ésta es una forma bastante primitiva de particionamiento “vertical” de una tabla. Los registros se repartirán sistemáticamente entre los diferentes ficheros del grupo al que pertenece la tabla, siguiendo un criterio pseudo aleatorio. Puestos a pedir, sería preferible dividir los registros de acuerdo a una condición lógica (clientes de distintas provincias, por ejemplo), pero “algo” es siempre mejor que “nada”.

¿Tiene sentido tomarse el trabajo de definir grupos de ficheros, en cualquiera de sus variantes, si solamente tenemos uno o dos discos duros? En mi opinión, sí. Cuando la base de datos es para uso de un solo cliente (quizás nosotros mismos), debemos ser optimistas y pensar que los Reyes Magos pueden regalarnos un segundo disco. Si vamos a instalar la aplicación en varios lugares, es conveniente diseñar un sistema flexible de particiones. Puede que en algunas instalaciones debamos conformarnos con un simple disco en el servidor, pero puede que en otras tengamos todos los medios de hardware que se nos ocurra pedir.

## Algunas consideraciones físicas

Los ficheros de datos de SQL Server 7 se dividen en páginas físicas de 8KB de longitud. Se trata de un avance importante respecto a la versión anterior, en la que las páginas tenían solamente 2KB. La primera ventaja: ahora se admiten registros más grandes. En la versión 6.5 el tamaño de un registro, sin contar imágenes y documentos largos, estaba limitado a cerca de 1700 bytes, una cantidad evidentemente menor que el antiguo tamaño de página. La última aplicación que he escrito para SQL Server 6.5, por ejemplo, trabaja sobre un esquema relacional (bastante mal diseñado, hay que reconocerlo) en el que los datos sobre productos están “esparcidos” a lo ancho de 6 tablas distintas. Si se hubiesen agrupado en una sola tabla, el ancho de registro habría superado el máximo permitido. Por lo tanto, hay que efectuar agónicos encuentros naturales entre tablas para restaurar esa información.

Se notará otra mejora en rendimiento gracias a los índices. El factor principal para medir la eficiencia de un índice es el número de niveles de profundidad del árbol implementado. El número de niveles, a su vez, depende de la cantidad de claves que quepan en un nodo. Y esta cantidad está determinada fundamentalmente por el ancho de página. Índices menos profundos, índices más rápidos.

¿Por qué Microsoft ha tardado tanto en aumentar el ancho de página? Sencillamente porque hasta la versión 6.5 el bloqueo más selectivo que implementaba era a nivel de página. Si usted modificaba un registro, el sistema lo bloqueaba... a él y a los demás registros que tuviesen la desgracia de compartir la misma página. En este escenario, mientras mayor fuese el tamaño de página, más angustioso sería el comportamiento del sistema frente a los bloqueos.

Por último, debo aclarar que, aunque existe un parámetro *FILEGROWTH* para indicar el factor de crecimiento de un fichero de datos, SQL Server siempre redondea la

cantidad al próximo múltiplo de 64KB, que corresponde al espacio ocupado por 8 páginas. En la jerga interna del producto, a estos grupos de 8 páginas se les denomina *extensiones*.

## Tipos básicos

Seamos organizados, y echemos un vistazo primero a los tipos de datos básicos que ofrece la versión 7 de SQL Server. Son los siguientes:

- 1 Tipos numéricos enteros.  
Incluye los tipos *integer*, *smallint*, *tinyint* y *bit*. Ocupan respectivamente 4 bytes, 2 bytes, 1 byte y 1 bit. SQL Server 2000, además, introduce *bigint*, para representar enteros de 64 bits.
- 2 Tipos numéricos reales.  
Están los tipos flotantes, *float* y *real*, que equivalen al *Double* y al *Single* de Delphi, y están los tipos “exactos”: *numeric* y *decimal*, que son equivalentes.
- 3 Tipos de moneda  
Los tipos *money* y *smallmoney* son una clase especial de tipos numéricos. El primero es similar al tipo *Currency* de Delphi: se representa como un entero de 8 bytes (un *bigint*), pero se reservan cuatro dígitos para los valores decimales. En contraste, *smallmoney* se representa sobre 4 bytes, aunque sigue manteniendo cuatro dígitos decimales.
- 4 Cadenas alfanuméricas.  
Los tipos *char* y *varchar* almacenan cadenas de longitud fija y variable. Si necesitamos cadenas Unicode, debemos recurrir a *nchar*, *nvarchar* y *ntext*; la *n* vale por *national*. En total, una columna de estos tipos puede ocupar hasta 8000 bytes (4000 solamente si es Unicode).
- 5 Fecha y hora.  
No existen representaciones por separados para fechas y horas, sino que siempre se representan juntas. El tipo *datetime* es el de mayor capacidad, y ocupa 8 bytes. Su rango de representación va desde el 1 de enero de 1753 hasta el 31 de diciembre de 9999. El tipo *smalldatetime* ocupa solamente 4 bytes, pero abarca un intervalo menor: desde el 1 de enero de 1900 hasta el 6 de junio de 2079. La precisión de *datetime* permite representar milisegundos, mientras que con *smalldatetime* sólo podemos representar segundos.
- 6 Tipos binarios y de longitud variable.  
Los tipos *binary* y *varbinary* almacenan valores sin interpretar de hasta 255 bytes de longitud. Para representaciones de mayor tamaño debemos recurrir a *text* e *image*.
- 7 Rarezas y miscelánea.  
El tipo *uniqueidentifier* permite almacenar identificadores únicos (GUID), con el formato de los identificadores COM. Si se desea obtener un número único a nivel de toda la base de datos, se puede utilizar *timestamp*. Por último, el tipo *cursor* permite almacenar una referencia a un cursor. Todos estos tipos han sido introducidos en la versión 7.

Aunque el tipo *varchar* de SQL Server permite almacenar cadenas de caracteres de longitud variable, en la versión 6.5, los registros que contenían estos campos ocupaban siempre el tamaño máximo. La versión 7 corrigió este derrochador comportamiento.

## Tipos de datos definidos por el programador

Para no variar, MS SQL Server complica y distorsiona la creación de tipos de datos por el programador. En vez de utilizar dominios, o algún mecanismo elegante de definición, se utiliza un procedimiento almacenado, *sp\_addtype*, para crear nuevos tipos de datos:

```
sp_addtype      telefono, 'char(9)', null
```

Estas son todas las posibilidades de *sp\_addtype*: especificar el nombre del nuevo tipo, indicar a qué tipo predefinido es equivalente, y decir si el tipo admite valores nulos o no. ¿Existe alguna forma de asociar una restricción a estos tipos de datos? Sí, pero la técnica empleada es digna de Forrest Gump. Primero hay que crear una *regla*:

```
create rule solo_numeros as
    @valor not like '%[^0-9]%'
```

Luego, hay que *asociar* la regla al tipo de datos:

```
sp_bindrule     solo_numeros, telefono
```

Existen también instrucciones **create default** y *sp\_bindefault* para asociar valores por omisión a tipos de datos creados por el programador.

### DIGAMOS ALGO BUENO...

No todo es negativo con los tipos de datos creados por el usuario en SQL Server. A diferencia de lo que sucede en InterBase, podemos utilizar los nombres de tipos que creamos en las declaraciones de parámetros y variables locales de procedimientos almacenados.

## Creación de tablas y atributos de columnas

La sintaxis para la creación de tablas, a grandes rasgos, es similar a la de InterBase, con la lógica adición de una cláusula opcional para indicar un segmento donde colocar la tabla:

```
create table Clientes (
    Codigo      int not null primary key,
    Nombre      varchar(30) not null unique,
    Direccion   varchar(35) null,
)
on SegmentoMaestro
```

La cláusula **on** se refiere, a partir de SQL Server 7, a un grupo de ficheros definido durante la creación de la base de datos o posteriormente.

Sin embargo, hay diferencias notables en los atributos que se especifican en la definición de columnas. Por ejemplo, en la instrucción anterior he indicado explícitamente que la columna *Direccion* admite valores nulos. ¿Por qué? ¿Acaso no es éste el comportamiento asumido por omisión en SQL estándar? Sí, pero Microsoft no está de acuerdo con esta filosofía, y asume por omisión que una columna no puede recibir nulos. Peor aún: la opción '*ANSI null default*', del procedimiento predefinido de configuración *sp\_dboption*, puede influir en qué tipo de comportamiento se asume. ¿Le cuesta tanto a esta compañía respetar un estándar?

Uno de los recursos específicos de SQL Server es la definición de columnas con el atributo **identity**. Por ejemplo:

```
create table Colores (
    Codigo          integer identity(0,1) primary key,
    Descripcion     varchar(30) not null unique
)
```

En la tabla anterior, el valor del código del color es generado por el sistema, partiendo de cero, y con incrementos de uno en uno. ¿Es recomendable utilizar **identity** para las claves primarias? Todo depende de la interfaz de acceso SQL que estemos utilizando. Por ejemplo, el BDE tuvo siempre problemas con este tipo de campos. Sin embargo, ADO los maneja correctamente. Y con Midas o DataSnap, no hay problema alguno en utilizar ese atributo.

En este aspecto, SQL Server también se aparta de InterBase y Oracle, que ofrecen *generadores* y *secuencias*, respectivamente, como ayuda para crear campos con valores en secuencia. La diferencia consiste en que no podemos saber cuál valor se asignará a un registro hasta que no hagamos la inserción. Sin embargo, a partir de ese momento es muy sencillo averiguar el número que nos ha correspondido, gracias a la variable global *@@identity*:

```
select @@identity
```

No se preocupe, que no han desaparecido líneas en la instrucción anterior. En SQL Server se puede omitir la cláusula **from** de un **select**. La sentencia en cuestión devuelve el último valor de identidad asignado por *nuestra conexión*. Lo explicaré con un ejemplo:

- 1 Hay una tabla *CLIENTES*, y su clave primaria *IDCliente* ha sido declarada con el atributo **identity**.
- 2 Hay dos personas conectadas a la base de datos: usted y yo. Usted inserta un nuevo cliente y, aunque todavía no lo sabe, SQL Server le ha asignado la clave 1234.
- 3 Entonces, desde la segunda conexión, soy yo quien añade otro cliente, el 1235.

- 4 Usted pregunta por el valor de *@@identity*, y SQL Server le responderá acertadamente que es 1234, porque el 1235 se asignó a través de otra conexión.

Las cláusulas **check** de SQL Server permiten solamente expresiones que involucran a las columnas de la fila actual de la tabla, que es lo que manda el estándar (¡por una vez!). Esta vez podemos decir algo a favor: las expresiones escalares de SQL Server son más potentes que las de InterBase, y permiten diseñar validaciones consecuentemente más complejas:

```
create table Clientes (
    /* ... */
    Telefono      char(11),
    check (Telefono like
          ' [0-9] [0-9] [0-9] - [0-9] [0-9] - [0-9] [0-9] [0-9] [0-9] ' or
          Telefono like
          ' [0-9] [0-9] [0-9] - [0-9] [0-9] [0-9] - [0-9] [0-9] [0-9] [0-9] ')
)
```

## Tablas temporales

Y ahora, estimadísimo lector, tengo el placer de presentarle una de las características de SQL Server que ya quisieran para sí otras bases de datos (sin mencionar nombres): las *tablas temporales*. SQL Server permite que creamos tablas utilizando para sus nombres identificadores que comienzan con una almohadilla (#) o dos (##).

En primer lugar, estas tablas no se crean en la base de datos activa, sino que siempre van a parar a la base de datos *tempdb*. Es un detalle a tener en cuenta, que puede afectar a los tipos de datos que se pueden utilizar para sus columnas. En el siguiente ejemplo, la creación de la tabla *#temporal* produce un error:

```
use MiBaseDatos
go

execute sp_addtype T_IDENTIFIER, integer, 'not null'
go

create table #temporal (
    ident      T_IDENTIFIER,
    equiv      integer
)
go
```

El problema consiste en que el tipo de datos *T\_IDENTIFIER* se ha definido (correctamente) dentro de la base de datos *MiBaseDatos*. Pero aunque no se indique nada al respecto, *#temporal* va a parar a *tempdb*, que no sabe qué es un *T\_IDENTIFIER*. La solución, como puede imaginar, es abandonar la idea de utilizar nuestro tipo de datos y utilizar el equivalente predefinido por SQL Server.

Pero la característica principal de una tabla temporal es su tiempo de vida y su “alcance”, es decir, qué conexiones pueden ver la tabla y trabajar con ella. Hay dos tipos

de tablas temporales: aquellas cuyo nombre comienza con una sola almohadilla, y aquellas que utilizan dos. Para una tabla que utiliza una sola almohadilla:

- Solamente es visible desde la conexión que la crea.
- Otra conexión puede crear una tabla temporal con el mismo nombre, pero físicamente corresponden a objetos diferentes.
- Su vida termina, como máximo, cuando la conexión que la ha creado se cierra. Naturalmente, podemos ejecutar antes una instrucción **drop table**.

Por otra parte, si la tabla se crea con dos almohadillas consecutivas al inicio de su nombre:

- Puede ser vista desde otras conexiones.
- Esas otras conexiones pueden trabajar con la tabla, realizando búsquedas y actualizaciones sobre la misma.
- La tabla se destruye inmediata y automáticamente cuando no quedan conexiones abiertas que la utilicen.

El carácter “local” del primer tipo de tablas temporales es posible gracias a que SQL Server añade un sufijo numérico relacionado con la conexión al nombre de la tabla que ha indicado el usuario, y utiliza ese nuevo nombre para el objeto físico que se crea. Por lo tanto, no hay posibilidad alguna de colisión entre nombres de tablas temporales locales.

Esto también garantiza que la tabla se “destruya” aunque suceda lo que suceda con la conexión. Si la conexión se cierra y vuelve a abrir, el número de sesión asignado por el sistema será diferentes, y aunque SQL Server tuviese algún fallo interno y no se destruyese físicamente la tabla temporal, sería imposible volver a utilizar el mismo objeto físico. Por último, recuerde que *tempdb* se crea desde cero cada vez que se inicia el servicio de SQL Server.

## Integridad referencial

Las versiones de este producto anteriores a la 2000 realizaban una pobre implementación de las restricciones de integridad referencial. La queja principal era la carencia de *acciones referenciales*, similares a las de InterBase, para permitir la propagación de borrados y actualizaciones en cascada. El problema se agravaba por culpa de las limitaciones de los *triggers* de SQL Server que, como explicaré más adelante, solamente se disparan al finalizar la operación.

Por fortuna, SQL Server 2000 implementó las cláusulas de propagación, aunque con algunas limitaciones respecto al estándar de SQL 3, porque sólo se admiten las cláusulas **no action** y **cascade**. Para simular el comportamiento de **set null** y **set default**, que sí se implementan en InterBase, tendríamos que desactivar las declara-

ciones de integridad referencial, y hacer las comprobaciones pertinentes desde *triggers*.

Es muy importante tener en cuenta que SQL Server, a diferencia de InterBase, no crea índices automáticamente para verificar el cumplimiento de las restricciones de integridad referencial. Por ejemplo:

```
create table DIRECCIONES (                /* SQL Server 2000 */
    IDDireccion integer      not null identity,
    IDCliente   integer      not null,
    Direccion1  varchar(35)  not null,
    Direccion2  varchar(35)  null,
    CP          numeric(5)   not null,
    Ciudad      varchar(35)  not null,
    IDPais      integer      not null,

    primary key (IDDireccion),
    foreign key (IDCliente) references CLIENTES(IDCliente)
        on delete cascade,
    foreign key (IDPais)   references PAISES(IDPais)
        on delete no action
)
go
```

InterBase, en la tabla anterior, crearía por iniciativa propia un índice sobre la columna *IDCliente*, y otro sobre *IDPais*. El índice sobre *IDCliente* sería muy útil, porque nos ayudaría a encontrar rápidamente las direcciones asociadas a un cliente determinado. Pero el índice sobre *IDPais* presentaría varios problemas. En primer lugar, tendría muchas claves repetidas, en el caso típico, cuando hay pocos países en juego. Tampoco tendría mucho sentido utilizarlo para localizar todas las direcciones de cierto país, porque casi sería preferible recorrer entera la tabla de direcciones e ir filtrando columnas sobre la marcha. Y aunque es cierto que el índice serviría para acelerar las comprobaciones de la integridad referencial que se disparan al eliminar un país, esta última operación es tan poco frecuente que quizás no merezca la pena tener que cargar con el mantenimiento de un índice por este único motivo.

Pero hay más. Esta es una declaración típica de una tabla de detalles de pedidos:

```
create table DETALLES (
    IDPedido    integer not null,
    IDLinea     integer not null,
    /* ... más columnas ... */

    primary key (IDPedido, IDLinea),
    foreign key (IDPedido) references PEDIDOS(IDPedido)
        on delete cascade,
    /* ... más restricciones ... */
)
go
```

Estamos en presencia de una tabla con clave primaria compuesta, y el sistema crea automáticamente un índice sobre esas dos columnas. No tendría sentido crear un índice adicional sobre *IDPedido* a secas, porque esa columna forma parte del prefijo

del índice de la clave primaria; este índice sirve perfectamente para optimizar todas las verificaciones necesarias para obligar el cumplimiento de las restricciones de integridad referencial.

## Indices

La sintaxis para la creación de índices en MS SQL Server es básicamente la siguiente:

```
create [unique] [clustered | nonclustered] index nombreIndice
on tabla ( columnas )
[with opciones]
[on grupoFicheros]
```

Tenemos la posibilidad de definir *índices agrupados* (*clustered indexes*). Cuando una tabla tiene un índice agrupado, sus registros se almacenan ordenados físicamente en las páginas de datos de la base de datos. Esto implica que solamente puede haber un índice agrupado por tabla. Es muy eficiente recorrer una tabla utilizando uno de estos índices. En cambio, puede ser relativamente costoso mantener el índice agrupado cuando se producen actualizaciones.

Como sabemos, la definición de una clave primaria o única genera automáticamente un índice. Si queremos que éste sea el índice agrupado de la tabla, podemos indicarlo de la siguiente manera:

```
create table Clientes (
    Codigo          int not null,
    Nombre          varchar(30) not null,
    /* ... */
    primary key (Codigo),
    unique clustered (Nombre)
)
```

Los índices de MS SQL Server, a diferencia de los de InterBase y Oracle, pueden ignorar la distinción entre mayúsculas y minúsculas. Sin embargo, este comportamiento debe establecerse durante la instalación del servidor, y no puede modificarse más adelante.

## Procedimientos almacenados

He aquí la sintaxis de la creación de procedimientos almacenados en Transact-SQL:

```
create procedure Nombre[;Numero] [Parámetros]
[for replication|with recompile [with encryption]]
as Instrucciones
```

Por ejemplo:

```
create procedure ProximoCodigo @cod int output as
begin
    select @cod = ProxCod
```



```

from    Numeros holdlock
update Numeros
set     ProxCod = ProxCod + 1
end

```

*ProximoCodigo* es el típico procedimiento almacenado que extrae un valor numérico de una tabla de contadores y lo devuelve al cliente que lo ha solicitado. En primer lugar, vemos que los parámetros y variables de Transact SQL deben ir precedidos obligatoriamente por un signo @. Esto ya es una molestia, porque el convenio de nombres de parámetros es diferente al de Oracle e InterBase. Si desarrollamos una aplicación que deba trabajar indistintamente con cualquiera de estos servidores, habrá que considerar el caso especial en que el servidor sea MS SQL, pues los nombres de parámetros se almacenan estáticamente en el fichero *dflm*.

Además de ciertas curiosidades sintácticas, como que las instrucciones no necesitan ir separadas por puntos y comas, lo que más llama la atención en el procedimiento anterior es el uso de la opción **holdlock** en la cláusula **from** de la primera instrucción. Esta opción fuerza a SQL Server a mantener un bloqueo de lectura sobre la fila seleccionada, al menos hasta que finalice la transacción actual, y permite evitar tener que configurar las transacciones para el nivel de lecturas repetibles.

Vemos también que Transact SQL no utiliza la cláusula **into** de InterBase y Oracle para asignar los resultados de un **select** a variables o parámetros, sino que incorpora asignaciones en la propia cláusula **select**. De hecho, el que no exista un separador de instrucciones nos obliga a anteponer la palabra reservada **select** delante de una simple asignación de variables:

```

declare @i integer           /* Declaramos una variable local */
select @i = @@rowcount       /* Le asignamos una global */
if (@i > 255)                /* Preguntamos por su valor */
    /* ... */

```

Es característico de MS SQL Server y Sybase la posibilidad de programar procedimientos que devuelvan un conjunto de datos. En tal caso, el cuerpo del procedimiento debe consistir en una sentencia **select**, que puede contener parámetros. Sin embargo, la misma funcionalidad se logra desde Delphi con consultas paramétricas, que no comprometen además la portabilidad de la aplicación.

## Cursores

SQL Server no implementa la sentencia **for/do** de InterBase. En realidad, esa instrucción es única para InterBase, y casi todos los demás servidores ofrecen *cursores* como mecanismo de recorrido sobre tablas. Un cursor se define asociando un nombre a una instrucción SQL, como en este ejemplo:

```

declare QueHaHechoEsteTio cursor for
select Fecha, Total from Pedidos
where RefEmpleado = (select Codigo from Empleados
                     where Nombre = @Nombre)

```

Observe que estamos utilizando una variable, *@Nombre*, en la definición del cursor. Se supone que esa variable está disponible en el lugar donde se declara el cursor. Cuando se realiza la declaración no suenan trompetas en el cielo ni tiembla el disco duro; es solamente eso, una declaración. Cuando sí ocurre algo es al ejecutarse la siguiente instrucción:

```
open QueHaHechoEsteTio
```

Ahora se abre el cursor y queda preparado para su recorrido, que se realiza de acuerdo al siguiente esquema:

```
declare @Fecha datetime, @Total integer
fetch from QueHaHechoEsteTio into @Fecha, @Total
while (@@fetch_status = 0)
begin
    /* Hacer algo con las variables recuperadas */
    fetch from QueHaHechoEsteTio into @Fecha, @Total
end
```

La variable global *@@fetch\_status* es de vital importancia para el algoritmo, pues deja de valer cero en cuanto el cursor llega a su fin. Tome nota también de que hay que ejecutar un **fetch** también antes de entrar en el bucle **while**.

Una vez que se ha terminado el trabajo con el cursor, es necesario cerrarlo por medio de la instrucción **close** y, en ocasiones, liberar las estructuras asociadas, mediante la instrucción **deallocate**:

```
close QueHaHechoEsteTio
deallocate QueHaHechoEsteTio
```

La diferencia entre **close** y **deallocate** es que después de ejecutar la primera, aún podemos reabrir el cursor. En cambio, después de ejecutar la segunda, tendríamos que volver a declarar el cursor con **declare**, antes de poder utilizarlo.

El siguiente ejemplo, un poco más complicado, cumple la misma función que un procedimiento almacenado de mismo nombre que hemos desarrollado en el capítulo sobre InterBase, y que estaba basado en la instrucción **for...do**. Su objetivo es recorrer ordenadamente todas las líneas de detalles de un pedido, y actualizar consecuentemente las existencias en el inventario:

```
create procedure ActualizarInventario @Pedido integer as
begin
    declare dets cursor for
        select RefArticulo, Cantidad
        from Detalles
        where RefPedido = @Pedido
        order by RefArticulo
    declare @CodArt integer, @Cant integer

    open dets
    fetch next from dets into @CodArt, @Cant
    while (@@fetch_status = 0)
```

```

begin
    update Articulos
    set     Pedidos = Pedidos + @Cant
    where  Codigo = @CodArt
    fetch next from dets into @CodArt, @Cant
end
close dets
deallocate dets
end

```

Microsoft ofrece cursores bidireccionales en el servidor, y están muy orgullosos de ellos. Vale, los cursores bidireccionales están muy bien. Felicidades. Lástima que el BDE no los pueda aprovechar (es culpa del BDE). Y que Delphi sea tan bueno que no merezca la pena cambiar de herramienta de desarrollo.

## Triggers en Transact-SQL

Los *triggers* en Transact SQL son muy diferentes a los de InterBase y a los de la mayoría de los servidores existentes. Esta es la sintaxis general de la operación de creación de *triggers*:

```

create trigger NombreTrigger on Tabla
[with encryption]
for {insert,update,delete}
[with append] [not for replication]
as InstruccionSQL

```

En primer lugar, cada tabla solamente admite hasta tres *triggers*: uno para cada una de las operaciones **insert**, **update** y **delete**. Sin embargo, un mismo *trigger* puede dispararse para dos operaciones diferentes sobre la misma tabla. Esto es útil, por ejemplo, en *triggers* que validan datos en inserciones y modificaciones.

SQL Server 7 ha corregido esta situación, y permite definir más de un *trigger* por evento. Incluso si hemos activado la compatibilidad con la versión 6.5 por medio del procedimiento *sp\_dbcmplevel*, la cláusula **with append** indica que el nuevo *trigger* se debe añadir a la lista de *triggers* existentes para el evento, en vez de sustituir a uno ya creado. De no estar activa la compatibilidad, dicha cláusula no tiene efecto alguno.

Pero la principal diferencia consiste en el momento en que se disparan. Un *trigger* decente debe dispararse antes o después de una operación sobre *cada fila*. Los de Transact-SQL, en contraste, se disparan solamente *después* de una instrucción, que puede afectar a *una* o *más* filas. Por ejemplo, si ejecutamos la siguiente instrucción, el posible *trigger* asociado se disparará únicamente cuando se hayan borrado todos los registros correspondientes:

```

delete from Clientes
where Planeta <> "Tierra"

```

El siguiente ejemplo muestra como mover los registros borrados a una tabla de copia de seguridad:

```

create trigger GuardarBorrados
on Clientes
for delete as
insert into CopiaSeguridad select * from deleted

```

Como se puede ver, para este tipo de *trigger* no valen las variables *old* y *new*. Se utilizan en cambio las tablas *inserted* y *deleted*:

	insert	delete	update
<i>inserted</i>	Sí	No	Sí
<i>deleted</i>	No	Sí	Sí

Estas tablas se almacenan en la memoria del servidor. Si durante el procesamiento del *trigger* se realizan modificaciones secundarias en la tabla base, no vuelve a activarse el *trigger*, por razones lógicas.

Como es fácil de comprender, es más difícil trabajar con *inserted* y *deleted* que con las variables *new* y *old*. El siguiente *trigger* modifica las existencias de una tabla de inventarios cada vez que se crea una línea de pedido:

```

create trigger NuevoDetalle on Detalles for insert as
begin
    if @@RowCount = 1
        update Articulos
        set Pedidos = Pedidos + Cantidad
        from Inserted
        where Articulos.Codigo = Inserted.RefArticulo
    else
        update Articulos
        set Pedidos = Pedidos +
            (select sum(Cantidad)
             from Inserted
             where Inserted.RefArticulo=Articulos.Codigo)
        where Codigo in
            (select RefArticulo
             from Inserted)
end

```

La variable global predefinida *@@RowCount* indica cuántas filas han sido afectadas por la última operación. Al preguntar por el valor de la misma en la primera instrucción del *trigger* estamos asegurándonos de que el valor obtenido corresponde a la instrucción que desencadenó su ejecución. Observe también la sintaxis peculiar de la primera de las instrucciones **update**. La instrucción en cuestión es equivalente a la siguiente:

```

update Articulos
set Pedidos = Pedidos + Cantidad
from Inserted
where Articulos.Codigo =
    (select RefArticulo
     from Inserted) /* Singleton select! */

```

¿Hasta qué punto nos afecta la mala conducta de los *triggers* de Transact-SQL? La verdad es que muy poco, si utilizamos principalmente los métodos de tablas del BDE para actualizar datos. El hecho es que los métodos de actualización del BDE siempre modifican una sola fila por instrucción, por lo que @@RowCount siempre será uno para estas operaciones. En este *trigger*, un poco más complejo, se asume implícitamente que las inserciones de pedidos tienen lugar de una en una:

```
create trigger NuevoPedido on Pedidos for insert as
begin
    declare @UltimaFecha datetime, @FechaVenta datetime,
            @Num int, @CodPed int, @CodCli int

    select @CodPed =Codigo, @CodCli = RefCliente,
           @FechaVenta = FechaVenta
    from inserted
    select @Num = ProximoNumero
    from Numeros holdlock
    update Numeros
    set ProximoNumero = ProximoNumero + 1
    update Pedidos
    set Numero = @Num
    where Codigo = @CodPed
    select @UltimaFecha = UltimoPedido
    from Clientes
    where Codigo = @CodCli
    if (@UltimaFecha < @FechaVenta)
        update Clientes
        set UltimoPedido = @FechaVenta
        where Codigo = @CodCli
end
```

Observe cómo hemos vuelto a utilizar **holdlock** para garantizar que no hayan huecos en la secuencia de valores asignados al número de pedido.

## Integridad referencial mediante *triggers*

Es bastante complicado intentar añadir borrados o actualizaciones en cascada a las restricciones de integridad referencial de SQL Server 7. Como los *triggers* se disparan al final de la operación, antes de su ejecución se verifican las restricciones de integridad en general. Por lo tanto, si queremos implementar un borrado en cascada tenemos que eliminar la restricción que hemos puesto antes, y asumir también la verificación de la misma.

Partamos de la relación existente entre cabeceras de pedidos y líneas de detalles, y supongamos que no hemos declarado la cláusula **foreign key** en la declaración de esta última tabla. El siguiente *trigger* se encargaría de comprobar que no se inserte un detalle que no corresponda a un pedido existente, y que tampoco se pueda modificar posteriormente esta referencia a un valor incorrecto:

```
create trigger VerificarPedido on Detalles for update, insert as
    if exists(select * from Inserted
              where Inserted.RefPedido not in
```

```

                                (select Codigo from Pedidos))
begin
    raiserror('Código de pedido incorrecto', 16, 1)
    rollback tran
end

```

Aquí estamos introduciendo el procedimiento *raiserror* (sí, con una sola *'e'*), que sustituye a las excepciones de InterBase. El primer argumento es el mensaje de error. El segundo es la severidad; si es 10 o menor, no se produce realmente un error. En cuanto al tercero (un código de estado), no tiene importancia para SQL Server en estos momentos. A continuación de la llamada a esta instrucción, se deshacen los cambios efectuados hasta el momento y se interrumpe el flujo de ejecución. Recuerde que esto en InterBase ocurría automáticamente.

La documentación de SQL Server recomienda como posible alternativa que el *trigger* solamente deshaga los cambios incorrectos, en vez de anular todas las modificaciones. Pero, en mi humilde opinión, esta técnica puede ser peligrosa. Prefiero considerar atómicas a todas las operaciones lanzadas desde el cliente: que se ejecute todo o nada.

El *trigger* que propaga los borrados es sencillo:

```

create trigger BorrarPedido on Pedidos for delete as
delete from Detalles
where RefPedido in (select Codigo from Deleted)

```

Sin embargo, el que detecta la modificación de la clave primaria de los pedidos es sumamente complicado. Cuando se produce esta modificación, nos encontramos de repente con dos tablas, *inserted* y *deleted*. Si solamente se ha modificado un registro, podemos establecer fácilmente la conexión entre las filas de ambas tablas, para saber qué nuevo valor corresponde a qué viejo valor. Pero si se han modificado varias filas, esto es imposible en general. Así que vamos a prohibir las modificaciones en la tabla maestra:

```

create trigger ModificarPedido on Pedidos for update as
if update (Codigo)
begin
    raiserror('No se puede modificar la clave primaria',
            16, 1)
    rollback tran
end

```

Le dejo al lector que implemente la propagación de la modificación en el caso especial en que ésta afecta solamente a una fila de la tabla maestra.

## Triggers anidados y triggers recursivos

¿Qué sucede si durante la ejecución de un *trigger* se modifica alguna otra tabla, y esa otra tabla tiene también un *trigger* asociado? Todo depende de cómo esté configurado el servidor. Ejecute el siguiente comando en el programa *Query Analyzer*:

```
sp_configure 'nested triggers'
go
```

El procedimiento devolverá el valor actual de dicha opción, que puede ser *0* ó *1*. Si está activa, el *trigger* de la tabla afectada secundariamente también se dispara. El número de niveles de anidamiento puede llegar a un máximo de 16. Para cambiar el valor de la opción, teclee lo siguiente:

```
-- Activar los triggers anidados
sp_dbconfigure 'nested triggers', '1'
go
```

Recuerde que la opción *nested triggers* afecta a todas las bases de datos de un mismo servidor, así que tenga mucho cuidado con lo que hace.

Sin embargo, es muy diferente lo que sucede cuando la tabla modificada por el *trigger* es la propia tabla para la cual se define éste. Una de las consecuencias de que los *triggers* de SQL Server se disparen después de terminada la operación, es que si queremos modificar automáticamente el valor de alguna columna estamos obligados a ejecutar una instrucción adicional sobre la tabla: ya no tenemos una conveniente variable de correlación *new*, que nos permite realizar este cambio antes de la operación:

```
create trigger MantenerVersionMayusculas on Clientes
for insert, update as
if update (Nombre)
update Clientes
set NombreMay = upper(Nombre)
where Codigo in (select Codigo from inserted)
```

Si modificamos el nombre de uno o más clientes, la versión en mayúsculas del nombre de cliente debe actualizarse mediante una segunda instrucción **update**. ¿Y ahora qué, se vuelve a disparar el *trigger*? No si la versión de SQL Server es la 6.5. Pero si estamos trabajando con la 7, volvemos a depender del estado de una opción de la base de datos que, esta vez, se modifica mediante *sp\_dboption*:

```
-- Activar los triggers recursivos
sp_dboption 'facturacion', 'recursive triggers', 'true'
```

En cualquier caso, si el *trigger* vuelve a ejecutarse de forma recursiva no pasa nada malo en el ejemplo anterior, pues en la segunda ejecución la columna modificada es *NombreMay*, en vez de *Nombre*. Pero hay que tener mucho cuidado con otros casos, en que puede producirse una recursión potencialmente infinita. SQL Server 7 también limita a 16 los niveles de anidamiento de un *trigger* recursivo.



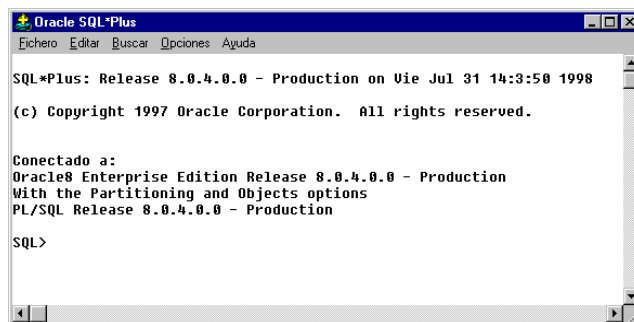


**L**AS REGLAS DEL JUEGO ESTÁN CAMBIANDO poco a poco, mientras el mundo gira (y mi guitarra solloza). Oracle ha sido el primero de los grandes sistemas relacionales en incluir extensiones orientadas a objetos significativas. Realmente, Oracle siempre ha destacado por su labor innovadora en relación con su modelo de datos y el lenguaje de programación en el servidor. De hecho, PL/SQL puede tomarse perfectamente como referencia para el estudio de *triggers*, procedimientos almacenados, tipos de datos, etc.

Por supuesto, no puedo cubrir todo Oracle en un solo capítulo. Por ejemplo, evitaré en lo posible los temas de configuración y administración. Tampoco entraremos en la programación de *packages* y otras técnicas particulares de este sistema, por entender que hacen difícil la posterior migración a otras bases de datos. Sin embargo, sí veremos algo acerca de las extensiones de objetos, debido al soporte que Delphi ofrece para las mismas.

## Sobreviviendo a SQL\*Plus

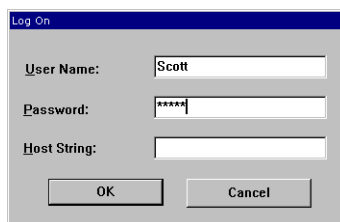
Oracle ofrece varias utilidades mediante las cuales pueden crearse tablas, procedimientos y tipos en sus bases de datos. La más utilizada se llama *SQL\*Plus*, y permite la ejecución de comandos aislados y de *scripts*. La siguiente imagen muestra a SQL\*Plus en funcionamiento:



Como podemos apreciar, es una utilidad basada en el modo texto, como en los viejos tiempos de UNIX, MSDOS y similares. En lógica consecuencia, SQL\*Plus tiene

fama de incómoda e insoportable, por lo que tenemos que aprovechar al máximo las pocas facilidades que nos ofrece.

Para comenzar, pongamos por caso que nos hemos conectado a la base de datos de ejemplos que trae Oracle como el usuario *Scott* con su contraseña *tiger*. Esta conexión se realiza mediante el cuadro de diálogo que presenta inicialmente SQL\*Plus:



Al tratarse, en mi caso, de Personal Oracle, puedo dejar vacío el campo *Host String*. Si desea conectarse a un servidor remoto, aquí debe indicar un nombre de “servicio” creado con SQL\*Net Easy Configuration. Más adelante podemos utilizar estas instrucciones para conectarnos como otro usuario, o para desconectarnos:

```
connect system/manager;  
disconnect;
```

*System* es el nombre del administrador de bases de datos, y *manager* es su contraseña inicial.

En principio, podemos teclear cualquier sentencia SQL en respuesta a la petición (*prompt*) de SQL\*Plus, terminándola con un punto y coma. La instrucción puede ocupar varias líneas; en tal caso, en el editor aparecen los números de líneas en la medida en que vamos creándolas. Al final, SQL\*Plus detecta el punto y coma, para ejecutar entonces la sentencia:

```
SQL> create table prueba (  
1      Id      integer,  
2      Nombre  varchar(30)  
3  );
```

¿Y qué pasa si hemos metido garrafalmente la extremidad inferior al teclear? Nada, que siempre hay una segunda oportunidad. Teclee *edit*, y aparecerá el Bloc de Notas con el texto de la última sentencia introducida, para que pueda ser corregida. En realidad, el texto se guarda en un fichero temporal, de nombre *afiedt.buf*, y hay que utilizar el siguiente comando para ejecutar el contenido de este fichero, una vez corregido y salvado:

```
SQL> @afiedt.buf
```

El signo @ sirve para ejecutar *scripts* con sentencias PL/SQL. En el ejemplo anterior, no hemos tenido que indicar el directorio donde se encuentra el fichero, pues éste se ha creado en el directorio raíz de Oracle, *c:\orawin95\bin* en mi instalación.

Un comportamiento curioso de SQL\*Plus, y que me ha ocasionado no pocos quebraderos de cabeza, es que no permite líneas en blanco dentro de una instrucción. A mí me gusta, por ejemplo, dejar una línea en blanco dentro de la sentencia **create table** para separar la definición de columnas de la definición de restricciones a nivel de tabla. SQL\*Plus me obliga a utilizar al menos un comentario en esa posición:

```
create table prueba (
    Id      integer,
    Nombre  varchar(30),
    --
    primary key (Id),
    unique  (Nombre)
);
```

Hablemos acerca de los errores. Cuando cometemos alguno gordo, de los de verdad, SQL\*Plus lo indica inmediatamente. Pero a veces se comporta solapadamente, casi siempre cuando creamos *triggers*, procedimientos almacenados y objetos similares. Obtenemos de repente este mensaje:

```
Procedure created with compilation errors.
```

No hay misterio: Oracle ha detectado un error, pero es tan listo que ha podido corregirlo él mismo (o al menos eso pretende). En cualquier caso, teclee el comando *show errors* para averiguar cuáles han sido los errores detectados en la última instrucción.

Cuando estamos creando tipos, procedimientos, triggers y otros objetos complejos en una base de datos, debemos utilizar instrucciones que terminan en punto y coma. En este caso, SQL\*Plus requiere que terminemos toda la instrucción con una línea que contenga una barra inclinada. Este carácter actúa entonces de forma similar al carácter de terminación de los *scripts* de InterBase.

## Instancias, bases de datos, usuarios

En dependencia del sistema operativo donde se ejecuta un servidor de Oracle, éste puede trabajar simultáneamente con una o más bases de datos. En el caso de Personal Oracle para Windows 95, sólo puede estar activa una base de datos a la vez. Pero las versiones completas del producto sí admiten varias bases de datos activas simultáneamente.

A cada base de datos activa, junto a los procesos que Oracle lanza para su mantenimiento y los datos de las conexiones de usuarios, se le denomina *instancia* de Oracle. Cada instancia se identifica con un nombre; en el caso de Oracle Enterprise Edition, la instancia que se asocia a la base de datos instalada por omisión se llama *ORCL*. Si

tenemos que mantener simultáneamente varias bases de datos en un mismo servidor, necesitaremos una instancia debidamente identificada para cada una de ellas.

Los administradores de bases de datos tienen el privilegio de poder crear nuevas bases de datos. La instrucción necesaria está cargada de parámetros, como corresponde a una arquitectura compleja y con una larga historia. Este es un ejemplo sencillo de creación de bases de datos, con la mayoría de los parámetros asumidos por omisión:

```
create database Prueba
  datafile 'p_datos' size 10M
  logfile group 1 ('p_log1a', 'p_log1b') size 500K
            group 2 ('p_log2a', 'p_log2b') size 500K
```

Por supuesto, Oracle ofrece herramientas gráficas para crear y modificar bases de datos. En el ejemplo anterior se ha creado una base de datos con un solo fichero de datos, de 10MB, y con dos grupos de ficheros para el registro de transacciones, cada grupo con 500KB.

Cada base de datos de Oracle tiene una lista independiente de usuarios autorizados a trabajar con ella. La siguiente instrucción sirve para crear un nuevo usuario:

```
create user Nombre identified [by Contraseña | externally]
[OpcionesDeUsuario]
```

Al igual que sucede en InterBase, se pueden definir roles, y asignarlos a usuarios existentes para simplificar la posterior administración de privilegios con las dos conocidas instrucciones **grant** y **revoke**.

## Tipos de datos

Los tipos de datos básicos de Oracle son los siguientes:

Tipo de dato	Significado
<i>varchar2(n), nvarchar2(n)</i>	Cadenas de caracteres de longitud variable
<i>char(n), nchar(n)</i>	Cadenas de caracteres de longitud fija
<i>number(p,s)</i>	Números, con escala y precisión
<i>date</i>	Fecha y hora, simultáneamente
<i>long</i>	Cadenas de caracteres de hasta 2GB
<i>raw(n), long raw</i>	Datos binarios
<i>clob, nclob</i>	Objetos de caracteres grandes
<i>rowid</i>	Posición de un registro
<i>blob</i>	Objetos binarios grandes
<i>bfile</i>	Puntero a un fichero binario externo a la base de datos
<i>mlslabel</i>	Utilizado por compatibilidad con el pasado

Notemos, en primer lugar, que los tipos de caracteres tienen dos versiones, y el nombre de una de ellas comienza con *n*. La *n* significa *national*, y los tipos que la indican deben ofrecer soporte para los conjuntos de caracteres nacionales de múltiples bytes: léase japonés, chino y todos esos idiomas que nos suenan a ídem. Además, los tipos *long* y *clob/nclob* se parecen mucho, lo mismo que *long raw* y *blob...* y es cierto, efectivamente. La diferencia consiste en que los tipos *lob* (de *large objects*) se almacenan más eficientemente y sufren menos restricciones que *long* y *long raw*.

¿Y dónde están nuestros viejos conocidos, los tipos *integer*, *numeric*, *decimal* y *varchar*? Oracle los admite, pero los asocia con alguno de sus tipos nativos. Por ejemplo, *integer* es equivalente a *number(38)*, mientras que *varchar* es lo mismo, hasta la versión 8, que *varchar2*. Sin embargo, Oracle recomienda utilizar siempre *varchar2*, pues en futuras versiones puede variar la semántica de las comparaciones de cadenas de caracteres, para satisfacer el estándar SQL-3.

Estos tipos de datos son los predefinidos por Oracle, y son además los que pueden utilizarse en las definiciones de tablas. Existen los tipos de datos definidos por el usuario, que estudiaremos más adelante, y están los tipos de datos de PL/SQL, que pueden emplearse para definir variables en memoria. Por ejemplo, el tipo *pls\_integer* permite definir enteros binarios de 4 bytes, con las operaciones nativas de la CPU de la máquina.

## Creación de tablas

Como es de suponer, la sentencia de creación de tablas de Oracle tiene montones de parámetros para configurar el almacenamiento físico de las mismas. Para tener una idea, he aquí una sintaxis abreviada de esta instrucción en Oracle 7 (la versión 8 ha añadido más cláusulas aún):

```
create table [Usuario.] NombreDeTabla (
    DefinicionesDeColumnas&Restricciones
)
[cluster NombreCluster (Columna [, Columna ...])]
[initrans entero] [maxtrans entero]
[pctfree entero] [pctused entero]
[storage almacenamiento]
[tablespace EspacioDeTabla]
[CláusulaEnable];
```

Gracias a Dios, ocurre lo típico: que podemos olvidarnos de la mayoría de las cláusulas y utilizar valores por omisión. De todos modos, hay un par de opciones de configuración que pueden interesarnos: la especificación de un *espacio de tablas* (*tablespace*) y el uso de *grupos* o *clusters*. El uso de espacios de tablas es una de las maneras de aprovechar las particiones físicas de una base de datos de Oracle. Los *clusters* de Oracle, por otra parte, permiten ordenar y agrupar físicamente los registros que tienen una misma clave, al estilo de los *clusters* de MS SQL Server. Pero también permiten colocar en una posición cercana los registros de otra tabla relacionados por esa clave.

Más adelante dedicaremos una sección a las modalidades de almacenamiento físico de tablas.

La creación de tablas en Oracle, obviando el problema de los parámetros físicos de configuración, no presenta mayores dificultades. Solamente tenga en cuenta los siguientes puntos:

- Recuerde que Oracle traduce los tipos de datos SQL a sus propios tipos nativos. Así que cuando creamos una columna de tipo *integer*, Oracle la interpreta como *number(38)*. Delphi entonces se ve obligado a tratarla mediante un componente *TFloatField*. En estos casos, es mejor definir la columna de tipo *number(10)* y activar la opción *ENABLE INTEGERS* en el BDE.
- Las cláusulas **check** no permiten expresiones que hagan referencia a otras tablas. Sí, amigo mío, solamente el humilde InterBase nos ofrece tal potencia.
- A pesar de todo, Oracle 8 no permite la modificación de campos de una restricción de integridad referencial en cascada, aunque sí permite la propagación de borrados mediante la cláusula **on delete cascade**.

## Indices en Oracle

Oracle ofrece varias opciones interesantes para la creación de índices. Por ejemplo, permite definir índices con clave invertida. Piense en una tabla en la que se insertan registros, y una de sus columnas es la fecha de inserción. Para el mantenimiento de dicha tabla, los registros se ordenan precisamente por esa columna, y existe la tendencia natural a trabajar más con los últimos registros insertados. Ahora asumamos que en nuestro sistema trabajan concurrentemente un alto número de usuarios. Casi todos estarán manipulando registros dentro del mismo rango de fechas, lo cual quiere decir que en el índice de fechas existirán un par de bloques que estarán siendo constantemente modificados. ¡Tanto disco duro para que al final los usuarios se encaprichen con un par de sectores! Pero como los programadores somos muy listos (... sí, ese que tiene usted a su lado también), hemos creado el índice sobre las fechas del siguiente modo:

```
create index FechaApunte on Apuntes(Fecha) reverse;
```

Cuando usamos la opción **reverse**, las claves se invierten *físicamente*, al nivel de bytes. Como resultado, fechas que antes eran consecutivas se distribuyen ahora de forma aleatoria, y se equilibra la presión sobre el disco duro. Alguna desventaja debe tener este sistema, y es que ahora no se puede aprovechar el índice para recuperar rápidamente los apuntes situados entre tal día y tal otro día.

Oracle permite utilizar las opciones **asc** y **desc** en la creación de índices, por compatibilidad con DB2, pero estas especificaciones son ignoradas. Pruebe, por ejemplo, a crear un índice ascendente y otro descendente para la misma columna o combinación de columnas, y verá cómo Oracle lo rechaza. Sin embargo, y a diferencia de lo que

sucede con InterBase, el motor de Oracle puede utilizar el mismo índice para optimizar la ordenación ascendente y descendente en una consulta.

Otra opción curiosa es el uso de índices por mapas de bits (*bitmap indexes*). Tenemos una tabla de clientes, y para cada cliente se almacena la provincia o el estado. Hay 50 provincias en España<sup>15</sup> y el mismo número de estados en los Estados Unidos. Pero hay un millón de clientes. Con un índice normal, cada clave de este índice debe tener un promedio de 20.000 filas asociadas, listadas de forma secuencial. ¡Demasiado gasto de espacio y tiempo de búsqueda! Un índice por mapas de bits almacena en cada clave una estructura en la que a cada fila corresponde un bit: si está en 0, la fila no pertenece a la clave, si está en 1, sí pertenece.

Hasta la versión 8, las comparaciones entre cadenas de caracteres en Oracle son sensibles a mayúsculas y minúsculas. Lo mismo sucede con los índices.

## Organización física de las tablas

Existen varias formas para organizar físicamente los registros de una tabla dentro de una base de datos. ¿Recuerda los *clusters* de MS SQL Server? El recurso de Oracle que más se le parece son las *tablas organizadas por índices* (*index-organized tables*), que se contraponen a la organización tradicional “en montón” (*heap*):

```
create table Detalles (
    Pedido number(10) not null,
    Linea number(5) not null,
    -- ... más columnas ...
    primary key (Pedido, Linea)
)
organization index;
```

Como se puede ver en el ejemplo, la cláusula **organization index** se sitúa al finalizar la declaración de la tabla, que debe contener una definición de clave primaria. En tal caso, los registros de la tabla se almacenan en los nodos terminales del índice *b-tree*, ordenados físicamente. Como resultado, la lectura de filas con claves adyacentes es muy eficiente. En la tabla de detalles anterior, por ejemplo, todas las filas de detalles de un pedido estarán situadas una a continuación de la otra.

La organización por índices es una característica de la versión 8 de Oracle. Puede combinarse además con el uso de tablas anidadas, que estudiaremos al final del capítulo. La organización tradicional puede indicarse ahora explícitamente con la cláusula **organization heap**.

Otra alternativa de almacenamiento es el uso de *clusters*, que Oracle interpreta de modo diferente a MS SQL Server. Cabeceras de pedidos y líneas de detalles comparten una misma columna: el número de pedido. Entonces podemos arreglar las cosas para que cada cabecera y sus líneas asociadas se almacenen en la misma página

<sup>15</sup> Si me he equivocado, perdonadme: yo no estudié Geografía Española en la escuela.

de la base de datos, manteniendo un solo índice sobre el número de pedido para ambas tablas. Este índice puede incluso utilizar la técnica conocida como *hash*, que permite tiempos de acceso muy pequeños. ¿La desventaja? Aunque las búsquedas son muy rápidas, cuesta entonces más trabajo el realizar una inserción, o modificar un número de pedido.

Para definir un *cluster* se utiliza la siguiente instrucción:

```
create cluster cpedidos (  
    Numero number(10)  
);
```

Note que no se especifican restricciones sobre la columna del *cluster*; esa responsabilidad corresponderá a las tablas que se añadirán más adelante. El siguiente paso es crear un índice para el *cluster*:

```
create index idx_cpedidos on cluster cpedidos;
```

Ya podemos crear tablas para que se almacenen en esta estructura:

```
create table Pedidos(  
    Numero number(10) not null primary key,  
    -- ... más definiciones ...  
)  
cluster cpedidos(Numero);  
  
create table Detalles(  
    Pedido number(10) not null,  
    Linea number(5) not null,  
    -- ... más definiciones ...  
)  
cluster cpedidos(Pedido);
```

Como podemos ver, no hace falta que coincidan los nombres de columnas en el *cluster* y en las tablas asociadas.

Desafortunadamente, existen limitaciones en el uso de *clusters*. Por ejemplo, no puede dividirse su contenido en particiones y las tablas asociadas no pueden contener columnas de tipo *lob*.

## Procedimientos almacenados en PL/SQL

Para crear procedimientos almacenados en Oracle debe utilizar la siguiente instrucción (menciono solamente las opciones básicas):

```
create [or replace] procedure Nombre [(Parámetros)] as  
    [Declaraciones]  
begin  
    Instrucciones  
end;  
/
```



Los parámetros del procedimiento pueden declararse con los modificadores **in** (se asume por omisión), **out** e **inout**, para indicar el modo de traspaso:

```
create or replace procedure ProximoCodigo(Cod out integer) as
begin
    -- ...
end;
/
```

Si ya conoce InterBase le será fácil iniciarse en PL/SQL, pues la sintaxis en ambos lenguajes es muy parecida, con las siguientes excepciones:

- Las variables locales y parámetros no van precedidas por dos puntos como en InterBase, ni siquiera cuando forman parte de una instrucción SQL.
- La cláusula **into** de una selección se coloca a continuación de la cláusula **select**, no al final de toda la instrucción como en InterBase.
- El operador de asignación en PL/SQL es el mismo de Pascal (**:=**).
- Las instrucciones de control tienen una sintaxis basada en terminadores. Por ejemplo, la instrucción **if** debe terminar con **end if**, **loop**, con **end loop**, y así sucesivamente.

Resulta interesante el uso de procedimientos anónimos en SQL\*Plus. Con esta herramienta podemos ejecutar conjuntos de instrucciones arbitrarios, siempre que tengamos la precaución de introducir las declaraciones de variables con la palabra reservada **declare**. Si no necesitamos variables locales, podemos comenzar directamente con la palabra **begin**. Por ejemplo:

```
declare
    Cod integer;
    Cant integer;
begin
    select Codigo into Cod
    from Clientes
    where Nombre = 'Ian Marteens';
    select count(*) into Cant
    from Pedidos
    where RefCliente = Cod;
    if Cant = 0 then
        dbms_output.put_line('Ian Marteens es un tacaño');
    end if;
end;
/
```

Este código asume que hemos instalado el “paquete” *dbms\_output*, que nos permite escribir en la salida de SQL\*Plus al estilo consola. Hay que ejecutar el siguiente comando de configuración antes de utilizar *put\_line*, para que la salida de caracteres se pueda visualizar:

```
set serveroutput on
```

## Consultas recursivas

Cuando expliqué los procedimientos almacenados de InterBase, mencioné que una de las justificaciones de este recurso era la posibilidad de poder realizar *clausuras transitivas*, o dicho de forma más práctica, permitir consultas recursivas. En Oracle no hay que llegar a estos extremos, pues ofrece una extensión del comando **select** para plantear directamente este tipo de consultas.

En la base de datos de ejemplo que instala Oracle, hay una tabla que representa los trabajadores de una empresa. La tabla se llama *EMP*, y pertenece al usuario *SCOTT*. Los campos que nos interesan ahora son:

Campo	Significado
<i>EMPNO</i>	Código de este empleado
<i>ENAME</i>	Nombre del empleado
<i>MGR</i>	Código del jefe de este empleado

La parte interesante de esta tabla es que el campo *MGR* hace referencia a los valores almacenados en el campo *EMPNO* de la propia tabla: una situación a todas luces recursiva. ¿Qué tal si listamos todos empleados de la compañía, acompañados por su nivel en la jerarquía? Digamos que el jefe supremo tiene nivel 1, que los que están subordinados directamente al Maharaja tienen nivel 2, y así sucesivamente. En tal caso, la consulta que necesitamos es la siguiente:

```
select level, ename
from   scott.emp
start  with mgr is null
connect by mgr = prior empno
```

Hay dos cláusulas nuevas en la instrucción anterior. Con **start with**, que actúa aproximadamente como una cláusula **where**, indicamos una consulta inicial. En nuestro ejemplo, la consulta inicial contiene una sola fila, la del privilegiado empleado que no tiene jefe. Ahora Oracle realizará repetitivamente la siguiente operación: para cada fila de la última hornada, buscará las nuevas filas que satisfacen la condición expresada en **connect by**. Quiere decir que en este paso se comparan en realidad dos filas, aunque sean de la misma tabla: la fila de la operación anterior y una nueva fila. Es por eso que existe el operador **prior**, para indicar que se trata de una columna de la fila que ya se seleccionó en la última pasada. Es decir, se añaden los empleados cuyos jefes están en el nivel anteriormente explorado. La pseudo columna **level** indica el nivel en el cuál se encuentra la fila activa.

El algoritmo anterior ha sido simplificado conscientemente para poderlo explicar más fácilmente. Pero toda simplificación es una traición a la verdad. En este caso, puede parecer que Oracle utiliza una búsqueda *primero en anchura* para localizar las filas, cuando en realidad la búsqueda se realiza *primero en profundidad*.

## Planes de optimización en Oracle

La visualización de los planes de optimización de Oracle es un buen ejemplo de aplicación de las consultas recursivas. Además, le interesará conocer la técnica para poder decidir si Oracle está ejecutando eficientemente sus instrucciones SQL o no.

Para poder averiguar los planes de optimización de Oracle, es necesario que exista una tabla, no importa su nombre, con el siguiente esquema:

```
create table plan_table (
  statement_id      varchar2(30),
  timestamp         date,
  remarks           varchar2(80),
  operation          varchar2(30),
  options           varchar2(30),
  object_node       varchar2(30),
  object_owner      varchar2(30),
  object_name       varchar2(30),
  object_instance   number,
  object_type       varchar2(30),
  search_columns    number,
  id                number,
  parent_id         number,
  position          number,
  other             long
);
```

Ahora se puede utilizar la instrucción **explain plan** para añadir filas a esta tabla. Después necesitaremos visualizar las filas añadidas:

```
explain plan set statement_id = 'ConsultaTonta'
into plan_table for
select * from emp order by empno desc;
```

Observe que se le asocia un identificador literal al plan de ejecución que se va a explicar. Este identificador literal se utiliza en la siguiente consulta, que devuelve el plan generado:

```
select lpad(' ', 2*(level-1)) ||
  operation || ' ' ||
  options || ' ' ||
  object_name || ' ' ||
  decode(id, 0, 'Coste=' || position) 'Plan'
from plan_table
start with id = 0 and statement_id = 'ConsultaTonta'
connect by parent_id = prior id and statement_id = 'ConsultaTonta';
```

*LPad* es una función predefinida de Oracle que sirve para rellenar con espacios en blanco a la izquierda. *Decode* sirve para seleccionar un valor de acuerdo a una expresión.

## Cursores

Oracle utiliza cursores para recorrer conjuntos de datos en procedimientos almacenados. Aunque la idea es similar, en general, a los cursores de SQL Server que hemos visto en el capítulo anterior, existen diferencias sintácticas menores. En el capítulo anterior habíamos definido un procedimiento almacenado *ActualizarInventario*, para recorrer todas las filas de un pedido y actualizar las existencias en la tabla *Articulos*. La siguiente es la versión correspondiente en Oracle:

```
create or replace procedure ActualizarInventario(Pedido integer) as
  cursor Dets(Ped integer) is
    select RefArticulo, Cantidad
    from Detalles
    where RefPedido = Ped
    order by RefArticulo;
  CodArt integer;
  Cant integer;

begin
  open Dets(Pedido);
  fetch Dets into CodArt, Cant;
  while Dets%found loop
    update Articulos
    set Pedidos = Pedidos + Cant
    whereCodigo = CodArt;
    fetch Dets into CodArt, Cant;
  end loop;
end;
/
```

Esta vez hemos utilizado un cursor con parámetros explícitos. Aunque también se admite que el cursor haga referencia a variables que se encuentran definidas a su alcance, el uso de parámetros hace que los algoritmos queden más claros. Si se definen parámetros, hay que pasar los mismos en el momento de la apertura del cursor con la instrucción **open**. Otra diferencia importante es la forma de detectar el final del cursor. En SQL Server habíamos recurrido a la variable global @@fetch\_status, mientras que aquí utilizamos el atributo %found del cursor. Existen más atributos para los cursores, entre ellos %notfound y %rowcount; este último devuelve el número de filas recuperadas hasta el momento.

También se puede realizar el recorrido del siguiente modo, aprovechando las instrucciones **loop** y **exit**, para ahorrarnos una llamada a **fetch**:

```
open Dets(Pedido);
loop
  fetch Dets into CodArt, Cant;
  exit when Dets%notfound;
  update Articulos
  set Pedidos = Pedidos + Cant
  whereCodigo = CodArt;
end loop;
```

Sin embargo, la forma más clara de plantear el procedimiento aprovecha una variante de la instrucción **for** que es muy similar a la de InterBase:

```
create or replace procedure ActualizarInventario(Pedido integer) as
begin
    for d in (
        select RefArticulo, Cantidad
        from Detalles
        where RefPedido = Ped
        order by RefArticulo) loop
        update Articulos
        set Pedidos = Pedidos + d.Cantidad
        whereCodigo = d.RefArticulo;
    end loop;
end;
/
```

## Triggers en PL/SQL

Los *triggers* de Oracle combinan el comportamiento de los *triggers* de InterBase y de MS SQL Server, pues pueden dispararse antes y después de la modificación sobre cada fila, o antes y después del procesamiento de un lote completo de modificaciones<sup>16</sup>. He aquí una sintaxis abreviada de la instrucción de creación de *triggers*:

```
create [or replace] trigger NombreTrigger
(before|after) Operaciones on Tabla
[[referencing Variables] for each row [when (Condicion)]]
declare
    Declaraciones
begin
    Instrucciones
end;
```

Vayamos por partes, comenzando por las *operaciones*. Sucede que un *trigger* puede dispararse para varias operaciones de actualización diferentes:

```
create or replace trigger VerificarReferencia
before insert or update of RefCliente on Pedidos
for each row
declare
    i pls_integer;
begin
    select count(*)
    into i
    from Clientes
    whereCodigo = :new.RefCliente;
    if i = 0 then
        raise_application_error(-20000,
            'No existe tal cliente');
    end if;
end;
/
```

---

<sup>16</sup> Informix es similar en este sentido.

*VerificarReferencia* se dispara cuando se inserta un nuevo pedido y cuando se actualiza el campo *RefCliente* de un pedido existente (¿por dónde van los tiros?). Además, la cláusula **for each row** indica que es un *trigger* como Dios manda, que se ejecuta fila por fila. He utilizado también la instrucción *raise\_application\_error*, para provocar una excepción cuando no exista el cliente referido por el pedido. Observe dos diferencias respecto a InterBase: la variable de correlación *new* necesita ir precedida por dos puntos, y la cláusula **into** se coloca inmediatamente después de la cláusula **select**.

La cláusula **referencing** permite renombrar las variables de correlación, *new* y *old*, en el caso en que haya conflictos con el nombre de alguna tabla:

```
referencing old as viejo new as nuevo
```

En cuanto a la cláusula **when**, sirve para que el *trigger* solamente se dispare si se cumple determinada condición. La verificación de la cláusula **when** puede también efectuarse dentro del cuerpo del *trigger*, pero es más eficiente cuando se verifica antes del disparo:

```
create or replace trigger ControlarInflacion
before update on Empleados
for each row when (new.Salario > old.Salario)
begin
    -- Esto es un comentario
end;
/
```

A semejanza de lo que sucede en InterBase, y a diferencia de lo que hay que hacer en MS SQL, Oracle no permite anular explícitamente transacciones dentro del cuerpo de un *trigger*.

## La invasión de las tablas mutantes

¿Pondría usted información redundante en una base de datos relacional? Muchos analistas se horrorizarían ante esta posibilidad. Sin embargo, veremos que a veces Oracle no nos deja otra opción. La “culpa” la tiene una regla fundamental que deben cumplir los *triggers* de PL/SQL: no pueden acceder a otras filas de la tabla que se está modificando, excepto a la fila que se modifica, mediante las variables de correlación *new* y *old*. Debo aclarar que esta regla se refiere a *triggers* a nivel de fila.

Supongamos que tenemos un par de tablas para almacenar departamentos y empleados:

```
create table Departamentos (
    ID      integer not null primary key,
    Nombre  varchar(30) not null unique
);

create table Empleados (
    ID      integer not null primary key,
    Nombre  varchar(35) not null,
```

```

Dpto    integer not null references Departamentos (ID)
);

```

Ahora queremos limitar el número de empleados por departamento, digamos que hasta 30. Nuestra primera reacción es crear un trigger en el siguiente estilo:

```

create or replace trigger LimitarEmpleados
before insert on Empleados
for each row
declare
NumeroActual integer;
begin
select count(*)
into NumeroActual
from Empleados
where Dpto = :new.Dpto;
if NumeroActual = 30 then
raise_application_error(-20001,
"Demasiados empleados");
end if;
end;
/

```

Lamentablemente, Oracle no permite este tipo de *trigger*, y lanza un error al intentar ejecutarlo. La dificultad está en la selección de la cantidad, que se realiza accediendo a la misma tabla sobre la cual se dispara el *trigger*. Oracle denomina *tabla mutante* a la tabla que va a ser modificada.

Hay varias soluciones a este problema, pero la más sencilla consiste en mantener en la tabla de departamentos la cantidad de empleados que trabajan en él. Esto implica, por supuesto, introducir una columna redundante:

```

create table Departamentos (
ID      integer not null primary key,
Nombre  varchar(30) not null unique,
Emps    integer default 0 not null
);

```

Debemos actualizar la columna *Emps* cada vez que insertemos un empleado, lo eliminemos o lo cambiemos de departamento. Este, por ejemplo, sería el *trigger* que se dispararía cada vez que insertamos un empleado:

```

create or replace trigger LimitarEmpleados
before insert on Empleados
for each row
declare
NumeroActual integer;
begin
select Emps
into NumeroActual
from Departamentos
where ID = :new.Dpto;
if NumeroActual = 30 then
raise_application_error(-20001,
"Demasiados empleados");
end if;

```

```

update Departamentos
set     Emps = Emps + 1
where   ID = :new.Dpto;

end;
/

```

Esta vez no hay problemas con la tabla mutante, pues la selección se realiza sobre la tabla de departamentos, no sobre los empleados. Aquí también tenemos que tener en cuenta otra regla de Oracle: en un *trigger* no se puede modificar la clave primaria de una tabla a la cual haga referencia la tabla mutante. *Empleados* hace referencia a *Departamentos* mediante su restricción de integridad referencial. Pero la última instrucción del *trigger* anterior solamente modifica *Emps*, que no es la clave primaria.

## Paquetes

En realidad, Oracle siempre ha tratado de adaptarse a las modas. En sus inicios, copió mucho de DB2. En estos momentos, toma prestado del mundo de Java (Oracle8i). Pero a mediados de los 80, el patrón a imitar era el lenguaje Ada. Este lenguaje surgió antes del reciente *boom* de la Programación Orientada a Objetos, pero introdujo algunas construcciones que iban en esa dirección. Una de ellas fueron los *paquetes* o *packages*, que rápidamente fueron asimilados por Oracle.

Un paquete en Oracle contiene una serie de declaraciones de tipos, variables y procedimientos, ofreciendo una forma primitiva de encapsulamiento. Una de las características más importantes de los paquetes es que sus variables públicas conservan sus valores durante todo el tiempo de existencia del paquete. Lo que es más importante ahora para nosotros: cada sesión de Oracle que utiliza un paquete recibe un espacio de memoria separado para estas variables. Por lo tanto, no podemos comunicar información desde una sesión a otra utilizando variables de paquetes. Pero podemos utilizarlas para mantener información de estado durante varias operaciones dentro de una misma sesión.

¿De qué nos sirven los paquetes a nosotros, programadores de Delphi? Un paquete no puede utilizarse directamente desde un programa desarrollado con la VCL. Eso sí, puede utilizarse internamente por los *triggers* y procedimientos almacenados de nuestra base de datos. Precisamente, utilizaré un paquete para mostrar cómo se pueden resolver algunas restricciones relacionadas con las tablas mutantes en Oracle.

Vamos a plantearnos la siguiente regla de empresa: un recién llegado a la compañía no puede tener un salario superior a la media. Si intentamos programar la restricción en un *trigger* a nivel de fila fracasaríamos, pues no podríamos consultar la media salarial de la tabla de empleados al ser ésta una tabla mutante durante la inserción. Claro, se nos puede ocurrir almacenar la media salarial en algún registro de otra tabla, pero esto sería demasiado engorroso.

Ya he mencionado que las restricciones correspondientes a tablas mutantes solamente se aplican a los *triggers* a nivel de fila. Un *trigger* a nivel de instrucción sí permite



leer valores de la tabla que se va a modificar. El problema es que seguimos necesitando el *trigger* por filas para comprobar la restricción. Si calculamos la media en un *trigger* por instrucciones, ¿cómo pasamos el valor calculado al segundo *trigger*? Muy fácil, pues utilizaremos una variable dentro de un paquete.

Primero necesitamos definir la interfaz pública del paquete:

```
create or replace package DatosGlobales as
    MediaSalarial number(15,2) := 0;
    procedure CalcularMedia;
end DatosGlobales;
/
```

Mediante el siguiente *script* suministramos un cuerpo al paquete, en el cual implementaremos el procedimiento *CalcularMedia*:

```
create or replace package body DatosGlobales as
    procedure CalcularMedia is
        begin
            select avg(sal)
            into    MediaSalarial
            from    emp;
        end;
end DatosGlobales;
/
```

Ahora creamos el *trigger* a nivel de instrucción:

```
create or replace trigger BIEmp
    before insert on Emp
begin
    DatosGlobales.CalcularMedia;
end;
/
```

En este caso tan sencillo, hubiéramos podido ahorrarnos el procedimiento, pero he preferido mostrar cómo se pueden crear. El valor almacenado en la variable del paquete se utiliza en el *trigger* a nivel de fila:

```
create or replace trigger BIEmpRow
    before insert on Emp
    for each row
begin
    if :new.Sal > DatosGlobales.MediaSalarial then
        raise_application_error(-20000,
            '¡Este es un enchufado!');
    end if;
end;
/
```

## Actualización de vistas mediante *triggers*

Preste atención a la técnica que describiré en esta sección; cuando estudiemos las actualizaciones en caché del BDE, nos encontraremos con un mecanismo similar, pero que será implementado en el lado cliente de la aplicación. Se trata de permitir actualizaciones sobre vistas que no son actualizables debido a su definición. Consideremos la siguiente vista, que condensa información a partir de la tabla de clientes:

```
create view Ciudades as
    select City, count(CustNo) Cantidad
    from Customer
    group by City;
```

La vista anterior muestra cada ciudad en la cual tenemos al menos un cliente, y la cantidad de clientes existentes en la misma. Evidentemente, la vista no es actualizable, por culpa de la cláusula **group by**. Sin embargo, puede que nos interese permitir ciertas operaciones de modificación sobre la misma. ¿Le doy un motivo? Bien, es muy frecuente que alguien escriba el nombre de una ciudad con faltas de ortografía, y que lo descubramos al navegar por la vista de *Ciudades*. Claro que podemos utilizar una instrucción SQL para corregir todos aquellos nombres de ciudad mal deletreados, pero ¿por qué no simular que actualizamos directamente la vista? Para ello, definamos el siguiente *trigger*:

```
create trigger CorregirCiudad
    instead of update on Ciudades
    for each row
begin
    update Customer
    set City = :new.City
    where City = :old.City;
end CorregirCiudad;
```

Observe la cláusula **instead of**, que es propia de Oracle. Ahora, la vista permite operaciones de modificación, ya sea desde una aplicación cliente o directamente mediante instrucciones **update**:

```
update Ciudades
set City = 'Munich'
where City = 'München';
```

No he podido resistirme a la tentación de escribir un chiste muy malo de mi cosecha. Supongamos que también definimos el siguiente *trigger*:

```
create trigger EliminarCiudad
    instead of delete on Ciudades
    for each row
begin
    delete from Customer
    where City = :old.City;
end EliminarCiudad;
```

Ahora usted puede montar en cólera divina y lanzar instrucciones como la siguiente:

```
delete from Ciudades
where City in ('Sodoma', 'Gomorra');
```

## Secuencias

Las secuencias son un recurso de programación de PL/SQL similar a los generadores de InterBase. Ofrecen un sustituto a las tradicionales tablas de contadores, con la ventaja principal de que la lectura de un valor de la secuencia no impide el acceso concurrente a la misma desde otro proceso. Cuando se utilizan registros con contadores, el acceso al contador impone un bloqueo sobre el mismo que no se libera hasta el fin de la transacción actual. Por lo tanto, el uso de secuencias acelera las operaciones en la base de datos.

Este es un ejemplo básico de definición de secuencias:

```
create sequence CodigoCliente increment by 1 starting with 1;
```

La secuencia anteriormente definida puede utilizarse ahora en un trigger del siguiente modo:

```
create or replace trigger BIClient
before insert on Clientes for each row
begin
    if :new.Codigo is null then
        select CodigoCliente.NextVal
        into :new.Codigo
        from Dual;
    end if;
end;
/
```

Aquí hemos utilizado *NextVal*, como si fuera un método aplicado a la secuencia, para obtener un valor y avanzar el contador interno. Si queremos conocer cuál es el valor actual solamente, podemos utilizar el “método” *CurrVal*. Es muy probable que le llame la atención la forma enrevesada que han escogido los diseñadores del lenguaje para obtener el valor de la secuencia. Me explico: *Dual* es una tabla predefinida por Oracle que siempre contiene solamente una fila. Uno pensaría que la siguiente instrucción funcionaría de modo más natural:

```
:new.Codigo := CodigoCliente.NextVal; -- ;;;INCORRECTO!!!
```

Pero no funciona. Sin embargo, sí funcionan instrucciones como la siguiente:

```
insert into UnaTabla(Codigo, Nombre)
values (CodigoCliente.NextVal, 'Your name here')
```

Los mismos problemas que presentan los generadores de InterBase se presentan con las secuencias de Oracle:

- No garantizan la secuencialidad de sus valores, al no bloquearse la secuencia durante una transacción.
- La asignación de la clave primaria en el servidor puede confundir al BDE, cuando se intenta releer un registro recién insertado. La forma correcta de utilizar una secuencia en una aplicación para Delphi es escribir un procedimiento almacenado que devuelva el próximo valor de la secuencia, y ejecutar éste desde el evento *OnNewRecord* ó *BeforePost* de la tabla

Tenga en cuenta que el segundo problema que acabamos de explicar se presenta únicamente cuando estamos creando registros y explorando la tabla al mismo tiempo desde un cliente. Si las inserciones transcurren durante un proceso en lote, quizás en el propio servidor, el problema de la actualización de un registro recién insertado no existe.

Como técnica alternativa a las secuencias, cuando deseamos números consecutivos sin saltos entre ellos, podemos utilizar tablas con contadores, al igual que en cualquier otro sistema de bases de datos. Sin embargo, Oracle padece un grave problema: aunque permite transacciones con lecturas repetibles, éstas tienen que ser sólo lectura. ¿Qué consecuencia trae esto? Supongamos que el valor secuencial se determina mediante las dos siguientes instrucciones:

```
select ProximoCodigo
into   :new.Codigo
from   Contadores;
update Contadores
set    ProximoCodigo = ProximoCodigo + 1;
```

Estamos asumiendo que *Contadores* es una tabla con una sola fila, y que el campo *ProximoCodigo* de esa fila contiene el siguiente código a asignar. Por supuesto, este algoritmo no puede efectuarse dentro de una transacción con lecturas repetibles de Oracle. El problema se presenta cuando dos procesos ejecutan este algoritmo simultáneamente. Ambos ejecutan la primera sentencia, y asignan el mismo valor a sus códigos. A continuación, el primer proceso actualiza la tabla y cierra la transacción. Entonces el segundo proceso puede también actualizar y terminar exitosamente, aunque ambos se llevan el mismo valor.

En el capítulo anterior vimos cómo Microsoft SQL Server utilizaba la cláusula **holdlock** en la sentencia de selección para mantener un bloqueo sobre la fila leída hasta el final de la transacción. Oracle ofrece un truco similar, pero necesitamos utilizar un cursor explícito:

```
declare
  cursor Cod is
    select ProximoCodigo
    from   Contadores
    for update;
begin
  open   Cod;
  fetch Cod into :new.Codigo;
```

```

update Contadores
set ProximoCodigo = ProximoCodigo + 1
where current of Cod;
close Cod;

end;
/

```

Observe la variante de la sentencia **update** que se ejecuta solamente para la fila activa de un cursor.

## Tipos de objetos

Ha llegado el esperado momento de ver cómo Oracle mezcla objetos con el modelo relacional. La aventura comienza con la creación de tipos de objetos:

```

create type TClientes as object (
  Nombre      varchar2(35),
  Direccion   varchar2(35),
  Telefono     number(9)
);
/

```

En realidad, he creado un objeto demasiado sencillo, pues solamente posee atributos. Un objeto típico de Oracle puede tener también métodos. Por ejemplo:

```

create type TClientes as object (
  Nombre      varchar2(35),
  Direccion   varchar2(35),
  Telefono     number(9),
  member function Prefijo return varchar2
);
/

```

Como es de suponer, la implementación de la función se realiza más adelante:

```

create or replace type body TClientes as
  member function Prefijo return varchar2 is
    C varchar2(9);
  begin
    C := to_char(Telefono);
    if substr(C, 1, 2) in ('91', '93') then
      return substr(C, 1, 2);
    else
      return substr(C, 1, 3);
    end if;
  end;
end;
/

```

Como Delphi no permite ejecutar, al menos de forma directa, métodos pertenecientes a objetos de Oracle, no voy a insistir mucho sobre el tema. Tenga en cuenta que estos “objetos” tienen una serie de limitaciones:

- No pueden heredar de otras clases.

- No existe forma de esconder los atributos. Aunque definamos métodos de acceso y transformación, de todos modos podremos modificar directamente el valor de los campos del objeto. No obstante, Oracle promete mejoras en próximas versiones.
- No se pueden definir constructores o destructores personalizados.

¿Dónde se pueden utilizar estos objetos? En primer lugar, podemos incrustar columnas de tipo objeto dentro de registros “normales”, o dentro de otros objetos. Suponiendo que existe una clase *TDireccion*, con dos líneas de dirección, código postal y población, podríamos mejorar nuestra definición de clientes de esta forma:

```
create type TClientes as object (
    Nombre          varchar2(35),
    Direccion        TDireccion,
    Telefono         number(9)
);
/
```

Sin embargo, no se me ocurre “incrustar” a un cliente dentro de otro objeto o registro (aunque algunos merecen eso y algo más). Los clientes son objetos con “vida propia”, mientras que la vida de un objeto incrustado está acotada por la del objeto que lo contiene. En compensación, puedo crear una tabla de clientes:

```
create table Clientes of TClientes;
```

En esta tabla de clientes se añaden todas aquellas restricciones que no pudimos establecer durante la definición del tipo:

```
alter table Clientes
    add constraint NombreUnico unique(Nombre);
alter table Clientes
    add constraint ValoresNoNulos check(Nombre <> '');
```

Es posible también especificar las restricciones anteriores durante la creación de la tabla:

```
create table Clientes of TClientes
    Nombre not null,
    check (Nombre <> ''),
    unique (Nombre)
);
```

Muy bien, pero usted estará echando de menos una columna con el código de cliente. Si no, ¿cómo podría un pedido indicar qué cliente lo realizó? ¡Ah, eso es seguir pensando a la antigua! Mire ahora mi definición de la tabla de pedidos:

```
create table Pedidos (
    Numero          number(10) not null primary key,
    Cliente         ref TClientes,
    Fecha           date not null,
    -- ... etcétera ...
);
```

Mis pedidos tienen un campo que es una referencia a un objeto de tipo *TCientes*. Este objeto puede residir en cualquier sitio de la base de datos, pero lo más sensato es que la referencia apunte a una de las filas de la tabla de clientes. La siguiente función obtiene la referencia a un cliente a partir de su nombre:

```
create or replace function RefCliente(N varchar2)
return ref TCientes as
  Cli ref TCientes;
begin
  select ref(C) into Cli
  from   Clientes C
  where  Nombre = N;
  return Cli;
end;
/
```

Ahora podemos insertar registros en la tabla de pedidos:

```
insert into Pedidos(Numero, Fecha, Cliente)
values (1, sysdate, RefCliente('Ian Marteens'));
```

La relación que existe entre los pedidos y los clientes es que a cada pedido corresponde a lo máximo un cliente (la referencia puede ser nula, en general). Sin embargo, también es posible representar relaciones uno/muchos: un pedido debe contener varias líneas de detalles. Comenzamos definiendo un tipo para las líneas de detalles:

```
create type TDetalle as object (
  RefArticulo    number(10),
  Cantidad       number(3),
  Precio         number(7,2),
  Descuento     number(3,1),
  member function Subtotal return number
);
```

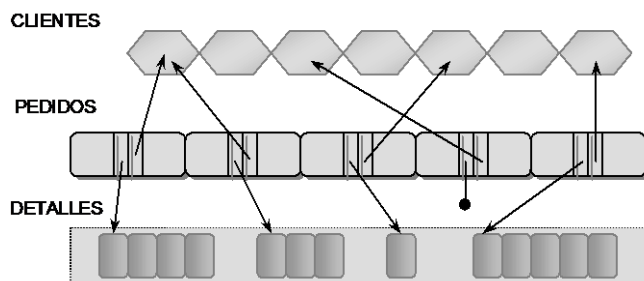
Esta vez no creamos una tabla de objetos, pero en compensación definimos un tipo de tabla anidada:

```
create type TDetalles as table of TDetalle;
```

El nuevo tipo puede utilizarse en la definición de otras tablas:

```
create table Pedidos (
  Numero      number(10) not null primary key,
  Cliente     ref TCientes,
  Fecha      date not null,
  Detalles   TDetalles
)
nested table Detalles store on TablaDetalles;
```

El siguiente diagrama puede ayudarnos a visualizar las relaciones entre los registros de clientes, pedidos y detalles:



Dos pedidos diferentes pueden hacer referencia al mismo cliente. Si eliminamos un cliente al que está apuntando un pedido, Oracle deja una referencia incorrecta, por omisión. Para evitarlo tendríamos que programar *triggers*. Por el contrario, un registro que contiene tablas anidadas es el propietario de estos objetos. Cuando borramos un pedido estamos borrando también todas las líneas de detalles asociadas. En consecuencia, a cada fila de detalles puede apuntar solamente un pedido.

Por último, Oracle permite declarar columnas vectoriales, parecidas a las tablas anidadas, pero con su tamaño máximo limitado de antemano:

```
create type VDetalles as varray(50) of TDetalle;
```

Aunque he utilizado un tipo de objeto como elemento del vector, también pueden utilizarse tipos simples, como los enteros, las cadenas de caracteres, etc.



## DB2 Universal Database

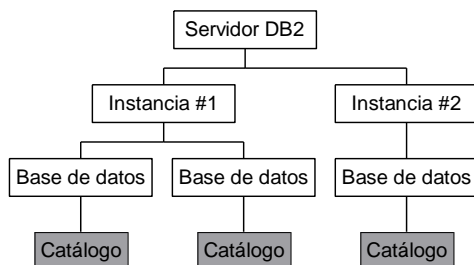
**H**ABRÍA SIDO SUFICIENTE TITULAR ESTE CAPÍTULO con un simple “DB2”. Pero mirándolo bien, eso de *Universal Database* tiene la suficiente resonancia acústica como para llamar la atención del lector más despistado. En realidad, DB2 es un sistema que me ha dejado buen sabor de boca, después de las pruebas que he realizado con él, así que no hay nada irónico en el título. Se trata de un sistema con *pedigree*: recuerde que los orígenes de SQL se remontan al *System R*, un prototipo de IBM que nunca llegó a comercializarse. Aunque para ser exactos, UDB (las siglas cariñosas que utilizan los de IBM al referirse a la Base de Datos Universal) está basada en un prototipo más adelantado, que se denominó *System R\**.

Una advertencia: el lector notará enseguida las semejanzas entre DB2 y Oracle. No se trata de una coincidencia, pues Oracle ha utilizado como patrón muchos de los avances del *System R* original y del propio DB2, posteriormente. Pero como se trata, en la mayoría de los casos, de aportaciones positivas, no tengo nada que objetar.

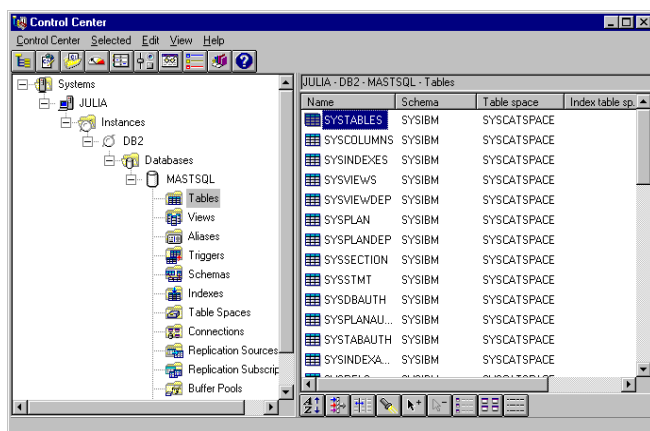
### Arquitectura y plataformas

Uno de los principales motivos para la denominación de *Universal Database* es que DB2 puede ejecutarse desde numerosas plataformas, tanto la parte cliente como la servidor. Existen versiones del servidor para Windows NT, OS/2, para varios sabores y colores de UNIX, e incluso para Windows 95. La parte cliente, conocida como *Client Application Enabler*, o *CAE*, puede instalarse también en distintos sistemas operativos. También se soporta un amplio rango de protocolos de comunicación. Es destacable la existencia de una versión “personal” de DB2, que puede ejecutarse en Windows 95.

Un servidor típico permite ejecutar simultáneamente una o más *instancias*, concepto que equivale más o menos al de un espacio de ejecución con nombre. Cada instancia, sin embargo, permite gestionar una o más bases de datos simultáneamente. Cada base de datos tiene su catálogo independiente:



La siguiente imagen muestra en acción a la utilidad *Control Center*, que se puede utilizar como punto de partida para la mayoría de las tareas de administración:



No voy a extenderme sobre las posibilidades de configuración de DB2, pues son muchas. Como cabe esperar de un sistema de gama alta, también se soporta la división de una base de datos en particiones físicas, y se implementan técnicas de replicación. Quiero destacar además que DB2 Enterprise Extended Edition permite el uso de bases de datos distribuidas, y que todas las versiones, excepto la versión personal, vienen preparadas para aprovechar equipos con varios procesadores. Un producto adicional, conocido como DataJoiner, permite extraer datos de otros tipos de servidores, como Oracle, SQL Server, Sybase e Informix, y utilizarlos como si se tratase de información en el formato nativo de DB2.

## Aislamiento de transacciones

DB2 implementa todos los niveles de aislamiento de transacciones, y el SQL Link correspondiente del BDE permite trabajar en los modos de lectura confirmada y lecturas repetibles. La implementación de las lecturas repetibles es muy similar a la de MS SQL Server: se colocan bloqueos de lectura sobre los registros leídos durante la transacción. Una transacción de lectura demasiado larga, en consecuencia, puede bloquear a muchas transacciones de escritura.

Estos son los nombres que DB2 da a los niveles de aislamiento estándar:

Según DB2	Según el estándar
<i>Uncommitted read</i>	<i>Uncommitted read (tiDirtyRead)</i>
<i>Cursor stability</i>	<i>Read Committed</i>
<i>Read stability</i>	<i>Repeatable Read</i>
<i>Repeatable read</i>	<i>Serializable</i>

Recuerde que *serializable* se diferencia de *lecturas repetibles* en que en el último modo pueden todavía aparecer registros fantasmas por culpa de las inserciones.

Una peculiaridad de DB2 consiste en que existe una instrucción para bloquear explícitamente una tabla. Naturalmente, el bloqueo se libera al terminar la transacción actual:

```
lock table NombreTabla in (share|exclusive) mode
```

## Tipos de datos

Los tipos de datos básicos de UDB son los siguientes:

Tipo de dato	Significado
<i>smallint</i>	Enteros de 16 bits
<i>integer</i>	Enteros de 32 bits
<i>decimal(p,s)</i>	Equivalente a <i>numeric</i> ; incluye precisión y escala
<i>real, float</i>	Valor flotante de 32 bits
<i>double</i>	Valor flotante de 64 bits
<i>char(n)</i>	Cadenas de longitud fija; $n \leq 254$
<i>varchar(n)</i>	Cadenas de longitud variable; $n \leq 4000$
<i>date</i>	Año, mes y día
<i>time</i>	Hora, minuto y segundo
<i>timestamp</i>	Fecha y hora; precisión en microsegundos
<i>graphic(n)</i>	Bloque binario de longitud fija ( $\leq 127$ )
<i>vargraphic(n)</i>	Bloque binario de longitud variable ( $\leq 2000$ )

Si no se indica la escala y precisión de un decimal, se asume *decimal(5,0)*. Una importante limitación es que los tipos *varchar* con tamaño mayor de 254 bytes no pueden participar en las cláusulas **order by**, **group by** y **distinct**.

A estos tipos hay que sumarle los tipos *lob* (*large objects*) que se muestran a continuación:

Tipo de dato	Significado
<i>blob</i>	Contiene datos binarios
<i>clob</i>	Contiene caracteres de un solo byte
<i>dbclob</i>	Contiene caracteres Unicode de dos bytes

Los tipos *lob* deben indicar un tamaño máximo de hasta 2GB, utilizando la siguiente sintaxis:

```
Foto          blob(5M) not logged compact,
Documento     clob(500K) logged not compact,
```

Las opción **not compact**, que es la que se asume por omisión, deja espacio libre al final de la zona reservada para el valor, previendo el crecimiento del objeto. Por ejemplo, es poco probable que la foto se sustituya, pero el documento puede ser editado. La otra opción, **not logged**, evita que las operaciones con este campo se guarden en el registro de rehacer transacciones. Así se gana en velocidad, pero si ocurre un fallo, no se pueden recuperar las transacciones ejecutadas sobre la columna desde la última copia de seguridad. Por omisión, se asume **logged**.

En cualquier caso, las modificaciones realizadas a campos *lob* pueden deshacerse sin problemas al anularse una transacción. La opción anterior solamente afecta al registro de rehacer.

Por último, el programador puede crear sus propios tipos de datos:

```
create distinct type Money as decimal(15,2) with comparisons;
```

Cuando creamos un tipo de datos en DB2, los operadores aplicables sobre el tipo base no están automáticamente a disposición del nuevo tipo. Para indicar los operadores que son válidos hay que hacer declaraciones en el siguiente estilo:

```
create function "+"(Money, Money) returns Money
source "+"(Decimal(), Decimal());
create function "-"(Money, Money) returns Money
source "-"(Decimal(), Decimal());
create function sum(Money) returns Money
source sum(Decimal());
```

Aunque hemos utilizado la notación prefijo en la definición de las sumas y restas, podemos sumar dinero en la forma habitual:

```
SalarioBase + PagaExtraordinaria
```

Pero, a no ser que definamos explícitamente la función de multiplicación, no podemos multiplicar dinero por dinero, operación que carece de sentido.

Se pueden añadir funciones adicionales a DB2 programadas externamente. Incluso es posible implementar estas funciones utilizando automatización OLE.

## Creación de tablas y restricciones

Todos los objetos que se crean en DB2 se agrupan en *esquemas*, que se crean implícita o explícitamente. Si al crear un objeto no indicamos su esquema, se verifica la exis-

tencia de un esquema con el nombre del usuario activo; si no existe, se crea automáticamente dicho esquema. El propietario de todos los esquemas implícitos es el usuario *SYSIBM*. Para crear un esquema explícitamente, y asignarle un propietario que no sea *SYSIBM*, se utiliza la siguiente instrucción:

```
create schema PERSONAL authorization Marteens;
```

A continuación muestro una serie de instrucciones para crear una tabla y colocar sus datos, índices y columnas de gran tamaño en particiones diferentes:

```
create tablespace Datos managed by system
using ('c:\db2data');
create tablespace Indices managed by system
using ('d:\db2idxs');
create long tablespace Blobs managed by database
using (file 'e:\db2blob\blobs.dat' 10000);
create table PERSONAL.Empleados (
  Codigo      int not null primary key,
  Apellidos   varchar(30) not null,
  Nombre      varchar(25) not null,
  Foto        blob(100K) not logged compact)
in Datos index in Indices long in Blobs;
```

En DB2 podemos utilizar las restricciones habituales en sistemas SQL: claves primarias, claves únicas, integridad referencial y cláusulas **check**. Estas últimas están limitadas a expresiones sobre las columnas de la fila activa, como en Oracle y SQL Server.

Las restricciones de integridad referencial en DB2 admiten las siguientes acciones referenciales:

<b>on delete</b>	<b>on update</b>
<i>cascade</i>	<i>no action</i>
<i>set null</i>	<i>restrict</i>
<i>no action</i>	
<i>restrict</i>	

Es interesante ver cómo DB2 distingue entre **no action** y **restrict**. Por ejemplo, si especificamos **restrict** en la cláusula **on update**, se prohíbe cualquier cambio en la clave primaria de una fila maestra que tenga detalles asociados. Pero si indicamos **no action**, lo único que se exige es que no queden filas de detalles huérfanas. Esta condición puede ser satisfecha si implementamos *triggers* de modificación convenientes sobre la tabla de detalles.

## Indices

La sintaxis del comando de creación de índices es:

```
create [unique] index NombreIndice
```

```
on NombreTabla (Columna [asc|desc], ...)
[include (Columna [asc|desc], ...)]
[cluster] [pctfree pct]
```

Como podemos ver, se pueden especificar sentidos de ordenación diferente, ascendente o descendente, para cada columna que forma parte del índice. También vemos que se pueden crear índices *agrupados* (*clustered indexes*), como los que ya estudiamos en MS SQL Server.

La cláusula **include** es interesante, pues se utiliza durante la optimización de columnas, y solamente se puede especificar junto a la opción **unique**. Las columnas que se indiquen en **include** se almacenan también en el índice. Por supuesto, la unicidad solamente se verifica con la clave verdadera, pero si el evaluador de consultas necesita también alguna de las columnas de **include** no tiene necesidad de ir a leer su valor al bloque donde está almacenado el registro. Analice la consulta siguiente:

```
select *
from customer, orders
where customer.custno = orders.custno and
customer.vip = 1
```

Si existe un índice único sobre el campo *CustNo* de la tabla *customer*, hemos incluido la columna *vip* en ese índice y el evaluador de consultas decide utilizarlo, nos ahorramos una segunda lectura en la cual, después de seleccionar la clave en el índice, tendríamos que leer el registro para saber si el cliente es una persona muy importante o no.

La longitud total de una clave de índice en DB2 no puede superar los 255 bytes. Aunque no lo mencioné antes, la longitud máxima de un registro es de 4005 bytes, sin contar el espacio ocupado por sus campos de tipo *lob*.

## Triggers

DB2 ofrece *triggers* muy potentes. Esta es la sintaxis de creación:

```
create trigger NombreTrigger
(no cascade before | after)
(insert | delete | update [of Columnas]) on NombreTabla
[referencing (old|new|old_table|new_table) as Ident ...]
[for each (statement | row)] mode db2sql
[when (Condición)]
(InstrucciónSimple |
begin atomic ListaInstrucciones end)
```

La sintaxis y semántica corresponde aproximadamente a la de Oracle, pero existen algunas peculiaridades:

- 1 Los triggers que se disparan *antes* deben ser siempre a nivel de fila, no de instrucción.
- 2 Un *trigger* de disparo previo nunca activa a otro trigger de este mismo tipo. Esta característica se indica en la cláusula obligatoria **no cascade before**.

- 3 En los *triggers* de disparo previo no se pueden ejecutar instrucciones **insert**, **update** o **delete**, aunque se permiten asignaciones a las variables de correlación. Esta es la explicación de la segunda regla.
- 4 DB2 no permite instrucciones condicionales generales, bucles y otros tipos de extensiones que podemos encontrar en InterBase, PL/SQL y Transact SQL.

Veamos un ejemplo sencillo:

```
create trigger ComprobarAumentoSalarial
no cascade before update of Salario on Empleados
referencing old as anterior new as nuevo
for each row mode db2sql
when (nuevo.Salario >= 2 * anterior.salario)
signal sqlstate 70001 ('Aumento desproporcionado');
```

La instrucción **signal** lanza una excepción con dos parámetros. El primero debe ser un código de cinco caracteres, el denominado **sqlstate**, cuyos valores están definidos en el estándar SQL. Para evitar conflictos con valores predefinidos, utilice para el primer carácter un dígito del 7 al 9, o una letra igual o posterior a la 'I'. El mensaje no está limitado a una constante, sino que puede ser cualquier expresión que devuelva una cadena de caracteres.

Al igual que sucede en InterBase y Oracle, una excepción dentro de un *trigger* anula cualquier cambio dentro de la operación activa. Este es el significado de la cláusula **begin atomic**, obligatoria cuando el cuerpo del *trigger* contiene más de una instrucción.

La actual carencia de instrucciones de control tradicionales en DB2 nos obliga a forzar la imaginación para realizar tareas bastante comunes. Es cierto que DB2 suministra extensiones al propio SQL que nos ayudan a superar estas dificultades. Por ejemplo, supongamos que cuando grabamos una cabecera de pedido queremos duplicar en la misma el nombre y la dirección del cliente. Este sería el *trigger* necesario:

```
create trigger DuplicarDatosClientes
no cascade before insert Pedidos
referencing new as nuevo
for each row mode db2sql
when (nuevo.Cliente is not null)
set (nuevo.NombreCliente, nuevo.DirCliente) =
select Nombre, Direccion
from Clientes
where Codigo = nuevo.Cliente;
```

Al no existir una instrucción **if**, estamos obligados a utilizar **when** para ejecutar condicionalmente el *trigger*. DB2 permite definir varios *triggers* para la misma operación sobre la misma tabla, así que tampoco se trata de una limitación radical. Por otra parte, DB2 no reconoce la variante **select/into** de la instrucción de selección, pero vea cómo se sustituye elegantemente con la equivalente instrucción **set**. Dicho sea de paso, tenemos también que recurrir a **set** para cualquier asignación en general:

```
set nuevo.Fecha = current timestamp
```

## Consultas recursivas

En el capítulo anterior mostré cómo Oracle permite expresar consultas recursivas. A modo de comparación, veamos como DB2 ofrece un recurso equivalente. Pero comencemos por algo aparentemente más sencillo. Tenemos una tabla con empleados, que almacena también el código del departamento en que estos trabajan. ¿Cuál departamento es el que tiene más empleados? Si existe una tabla de departamentos por separado, en la cual se almacene redundantemente el número de empleados, la respuesta es trivial:

```
select *
from Dept
where NumEmps = (select max(NumEmps) from Dept)
```

Para fastidiar un poco al lector, y obligarlo a acordarse de su SQL, mostraré una consulta equivalente a la anterior:

```
select *
from Dept
where NumEmps >= all (select NumEmps from Dept)
```

Pero, ¿qué pasaría si no existiera el campo redundante *NumEmp* en la tabla de departamentos? Para saber cuántos empleados hay por departamento tendríamos que agrupar las filas de empleados y utilizar funciones de conjunto. Si solamente nos interesa el código de departamento, la consulta sería la siguiente:

```
select DeptNo, count(*)
from Emp
group by DeptNo
```

Y la respuesta a nuestra pregunta sería:

```
select DeptNo, count(*)
from Emp
group by DeptNo
having count(*) >= all (select count(*) from Emp group by DeptNo)
```

Si no se le ha puesto la cara larga después de todo esto, es que usted es un caballero Jedi del SQL. Las cosas se han complicado al tener que evitar calcular el máximo de un total, pues SQL no permite anidar funciones de conjunto. ¿No habrá una forma más sencilla de expresar la pregunta? Sí, si utilizamos una *vista* definida como sigue:

```
create view Plantilla(DeptNo, Total) as
select DeptNo, count(*)
from Emp
group by DeptNo
```

Dada la anterior definición, podemos preguntar ahora:



```

select DeptNo
from Plantilla
where Total = (select max(Total) from Plantilla)

```

Sin embargo, no deja de ser un incordio tener que crear una vista temporal para eliminarla posteriormente. Y esta es precisamente la técnica que nos propone DB2:

```

with Plantilla(DeptNo, Total) as
  (select DeptNo, count(*) from Emp group by DeptNo)
select DeptNo
from Plantilla
where Total = (select max(Total) from Plantilla)

```

La instrucción de selección que va entre paréntesis al principio de la consulta, sirve para definir una vista temporal, que utilizamos dos veces dentro de la instrucción que le sigue.

Lo interesante es que **with** puede utilizarse en definiciones recursivas. Supongamos que la tabla de empleados *Emp* contiene el código de empleado (*EmpNo*), el nombre del empleado (*ENAME*) y el código del jefe (*Mgr*). Si queremos conocer a todos los empleados que dependen directa o indirectamente de un tal Mr. King, necesitamos la siguiente sentencia:

```

with Empleados(EmpNo, Mgr, EName) as
  ((select EmpNo, Mgr, EName
    from Emp
    where EName = 'KING')
  union all
  (select E1.EmpNo, E1.Mgr, E1.EName
    from Emp E1, Empleados E2
    where E2.EmpNo = E1.Mgr))
select *
from Empleados

```

El truco consiste ahora en utilizar el operador de conjuntos **union all**. El primer operando define la consulta inicial, en la cual obtenemos una sola fila que corresponde al jefe o raíz del árbol de mando. En el segundo operador de la unión realizamos un encuentro entre la tabla de empleados *Emp* y la vista temporal *Empleados*. DB2 interpreta la referencia a la vista temporal como una referencia a los registros generados inductivamente en el último paso. Es decir, en cada nuevo paso se añaden los empleados cuyos jefes se encontraban en el conjunto generado en el paso anterior. El algoritmo se detiene cuando ya no se pueden añadir más empleados al conjunto.

## Procedimientos almacenados

Una de las facetas que no me gustan de DB2 es que, aunque podemos crear procedimientos almacenados en el servidor, la implementación de los mismos debe realizarse en algún lenguaje externo: C/C++, Java, etc. En estos procedimientos pode-

mos realizar llamadas directas a la interfaz CLI de DB2, o utilizar sentencias SQL incrustadas que son traducidas por un preprocesador.

Evidentemente, se produce una ganancia de velocidad en la ejecución de estos procedimientos. ¿Pero a qué precio? En primer lugar, aumenta la complejidad de la programación, pues hay que tener en cuenta, por ejemplo, los convenios de traspaso de parámetros de cada lenguaje. Una modificación en un procedimiento nos obliga a pasar por todo el ciclo de editar/compilar con herramientas externas/instalar, algo que nos evitamos cuando el procedimiento se programa en alguna extensión procedimental de SQL. Además, lo típico es que el sistema operativo del servidor sea diferente que el de las estaciones de trabajo. Si el servidor reside en un UNIX o en OS/2, no podemos utilizar Delphi para esta tarea, lo cual nos obliga a utilizar otro lenguaje de programación adicional. En mi humilde opinión, se trata de una mala decisión de diseño.

# 3

## **Componentes de acceso a datos**

---

- **Interfaces de acceso a bases de datos**
- **MyBase: navegación**
- **Acceso a campos**
- **Controles de datos**
- **Rejillas y barras de navegación**
- **Indices, filtros y búsqueda**
- **Relaciones maestro/detalles**
- **Actualizaciones en MyBase**
- **Herencia visual y prototipos**

# Parte



## Interfaces de acceso a bases de datos

**D**EDIQUÉ CINCO AÑOS DE MI VIDA A estudiar Informática en una universidad de cuyo nombre no quiero acordarme. Tuve la suerte de tropezar con unos pocos profesores de los cuales aprendí la mayor parte de lo que puedo saber. Aprendí, o me enseñaron, análisis matemático, álgebra lineal y retorcida, matemática discreta e indiscreta, un sinnúmero de lenguajes de programación, casi todo lo que hay que saber sobre la teoría de las bases de datos relacionales; incluso me explicaron algo llamado Inteligencia Artificial, y me prometieron que diez años más tarde no tendría que volver a lavar los platos después de la cena, porque un simpático robot se haría cargo del asunto. ¡Ja!

Sin embargo, no me enseñaron cómo escribir una sencilla aplicación para bases de datos...

### ¿Nos basta con SQL?

Al lector le extrañará que, después de haber transcurrido buena parte de la historia contada en este libro, el autor se ponga solemne, enderece el nudo de su corbata y anuncie: “Estimados Señores, ¡bienvenidos al Mágico Mundo de Oz!” Pero soy sincero cuando confieso que las “reflexiones” que conforman este capítulo han venido al lugar donde se supone que tengo el cerebro hace muy poco tiempo.

Es fácil de explicar: Ya conocemos SQL lo suficiente como para pasar por expertos. Vamos incluso a imaginar que hemos implementado todo un sistema de base de datos con sus tablas, índices, procedimientos y transacciones. Lo llamaremos *imSQL*, por *Ian Marteens’ SQL* (juro que no es una parodia). Pero solamente hemos terminado el motor relacional. Ahora nos piden que programemos una DLL, o una clase COM, para respetar las modas, con el objetivo de que los programadores de Delphi, C++ o cualquier otro lenguaje decente puedan utilizar nuestro motor. Tenemos que diseñar primero las funciones que vamos a exportar en esa DLL.

La primera función nos vendrá automáticamente a la mente:

- Una función que permita ejecutar una sentencia SQL *cualquiera*.

¿Ha visto qué simple? Seamos un poco más precisos:

```
function imSqlEjecutar(  
    const Instruccion: string;  
    var RegistrosAfectados: Integer): Integer;
```

Se supone que vamos a pasar en *Instruccion* una cadena de caracteres con la sentencia SQL; por ahora ignoraremos la posibilidad de utilizar parámetros y otras ayudas similares. Para que nuestra interfaz sea compatible con cualquier lenguaje, incluyendo los más antiguos, tampoco vamos a utilizar excepciones para señalar errores, y haremos que el valor entero que retorna la función corresponda a un posible código de error. Finalmente, el parámetro *RegistrosAfectados* tendrá sentido solamente para ciertas instrucciones que modifiquen registros. En ese caso, retornará el número de registros afectados por la operación.

La función *imSqlEjecutar* es tan potente que con ella podemos hacer casi de todo en la base de datos. Podemos pasar sentencias DDL, del lenguaje de definición de datos, para crear, modificar o destruir tablas, índices y demás tipos de objetos. Si le pasamos instrucciones DML, del lenguaje de manipulación de datos, podremos crear, actualizar y eliminar registros de nuestras tablas. Pero, ¿qué pasa si queremos utilizar una sentencia **select**, para una consulta?

Es obvio que no debemos devolver todos los registros de golpe en un parámetro de salida. Eso serviría solamente cuando el tamaño de la relación generada por la consulta fuese pequeño. La solución es crear, al menos, dos funciones adicionales:

- Una función que ejecute, o inicie la ejecución, de la consulta en el servidor. Volvemos a ignorar la presencia de parámetros.
- Una función que podamos llamar varias veces, y en cada caso nos devuelva los datos del siguiente registro de la consulta, o “algo” que nos indique que no hay más registros disponibles.

Mostremos entonces los prototipos más plausibles para estas dos funciones:

```
function imSqlAbrir(  
    const Consulta: string;  
    var Cursor: Integer): Integer;  
function imSqlRecuperar(  
    Cursor: Integer;  
    Buffer: Pointer): Integer;
```

La primera función devuelve un valor entero: utilizaremos ese valor como un identificador de *cursor*, que pasaremos a la otra función, *imSqlRecuperar*, para copiar el próximo registro de la consulta dentro de un buffer reservado por el programador. El detalle sobre el buffer tiene relativamente poca importancia; el BDE, como nosotros,

le pide a la aplicación que reserve la memoria para el buffer de datos, mientras que DB Express se encarga internamente de estas estructuras.

El algoritmo típico de ejecución de una consulta sería algo así:

```
if imSqlAbrir('select * from productos', Cursor) = 0 then
  while imSqlRecuperar(Cursor, Buffer) = 0 do
    // ... hacer algo con los datos en el buffer ...
```

He asumido que la memoria para el buffer se reserva antes de iniciar este grupo de instrucciones, y he pasado deliberadamente por alto la necesidad de “cerrar” el cursor al terminar el recorrido.

### IMPORTANTE

He colado, de repente, el término *cursor* para referirme a la estructura que nos permite recuperar registros de una consulta desde una aplicación. Sin embargo, usted ya conoce la existencia de entidades en Transact SQL y PL/SQL que también se denominan *cursores*. Aunque la relación entre ambos conceptos es muy estrecha, se trata de objetos diferentes. De todos modos, es muy probable que la implementación del cursor que acabamos de explicar se base en algún tipo de cursor perteneciente al servidor SQL.

## Programación para tipos duros

¿Por qué nos hemos conformado con un cursor que solamente se puede mover en una dirección? Una de las razones es que no todos los sistemas de bases de datos permiten otros tipos de cursores. Pero también porque es el cursor que menos recursos consume. Para mantener la navegación en las dos direcciones sería necesario, como mínimo, que el servidor crease una lista de punteros a registros, o una copia física de los registros generados por la consulta. Aparte, tendría también que mantener la situación del registro activo de la conexión. Todo eso cuesta.

Y si el servidor no ayuda, podríamos implementar la navegación bidireccional en el lado cliente, que es lo que en realidad hacen el BDE y ADO, según veremos en breve. Pero tenga en cuenta que el cliente tendrá entonces que mantener una caché de registros, y que eso también es caro.

¿Es posible hacer algo entonces con una interfaz de acceso aparentemente tan limitada? La respuesta es afirmativa; es precisamente la forma en que los “tipos duros” han trabajado desde hace mucho tiempo. Enumeremos varios escenarios en los que los cursores unidireccionales pueden ser útiles:

- Impresión de informes  
Por lo general, para imprimir un informe solamente se necesita recorrer el resultado de una consulta en una sola dirección. No hace falta almacenar en ningún lugar los registros ya impresos. La única excepción ocurriría si quisiéramos que

en cada página apareciera el número total de páginas. Pero eso es un capricho tonto, y además existen varias maneras de implementarlo.

- Generación de gráficos  
En realidad podríamos considerarla como una forma especial de informe.
- Aplicaciones CGI/ISAPI para Internet  
Podríamos también considerar que una aplicación para Internet es una variante sofisticada de informes, en los que el resultado generado es texto HTML. Claro, también debemos tener en cuenta la posibilidad de que tengamos que actualizar datos. Pero no es arriesgado apostar que la mayoría de las consultas se recorrerán en un solo sentido.

De todos modos, los cursores unidireccionales asoman sus narices en los lugares menos esperados. Por ejemplo, la especificación 1.0 de JDBC, la interfaz de acceso a datos del nunca bien detestado Java, solamente incluía este tipo limitado de cursor; y hubo algunos que amenazaron con pasarse a Visual Basic cuando la siguiente revisión de JDBC hizo sitio a los cursores bidireccionales. Es a esta clase de “tipos duros” a los que me refería antes.

¿Cómo programan estos duros de matar cuando se ven obligados a trabajar en Delphi? Supongamos que tienen que editar los datos de un cliente seleccionado por el usuario:

- Primero lanzan una consulta, unidireccional claro está, preguntando por aquellos clientes que cumplen los requisitos exigidos por el usuario. Lógicamente, intentan que el conjunto resultado tenga un tamaño razonable.
- A continuación, utilizan un *TStringGrid* para introducir laboriosamente cada registro obtenido. Han oído decir que los controles *data aware* tienen vida propia, y eso es algo que les provoca pánico.
- Cuando el usuario finalmente selecciona un registro para edición, muestran un cuadro de diálogo plagado de componentes *TEdit*. Entonces se afanan en mover los datos del registro activo a su control de edición correspondiente.
- El usuario teclea, compara, vuelve a teclear y da retoques con el ratón como si fuera un artista. Mientras tanto, nuestro cowboy jadea comprobando la validez de cada una de las acciones del usuario: que las fechas que teclee sean fechas, que los datos numéricos tengan el número preciso de decimales... Todas las validaciones se realizan a mano.
- Finalmente, el artista, perdón, el usuario pulsa el botón de aceptar. El programador samurai debe entonces generar una instrucción **update** a partir de los valores iniciales y finales del registro, y enviarla a la base de datos. Y luego debe actualizar los valores del registro modificado en la rejilla de selección.

¿Sabe que, hasta cierto punto, siento algo de simpatía por este aguerrido colega? Después de todo, su forma de trabajo tiene mucho sentido. De hecho, como demostraré en este libro, *eso mismo es lo que Delphi hace* cuando nos dejamos de pamplinas y utilizamos los controles *data aware*. Lo único que gana un tipo duro programando a lo



bestia es estima personal. Pierde todas las ventajas de utilizar un sistema de programación RAD. Y tarda mucho más en terminar una aplicación que una persona inteligente que domine Delphi.

## Navegación es más que recuperación

Resumiendo, hay que reconocer que es muy engorroso limitarse a las técnicas básicas de acceso: ejecución directa y recuperación unidireccional. Sería conveniente disponer de estas dos facilidades adicionales:

- 1 Alguna técnica que permita almacenar en memoria del cliente los registros ya recuperados, y de esta forma simular la navegación en dos direcciones a lo largo de todo el resultado de la consulta.
- 2 Un sistema que permita efectuar modificaciones sobre registros en la caché, y que genere automáticamente las instrucciones SQL de actualización para sincronizar, en algún momento, el estado de la caché con el contenido real de la base de datos.

Y es aquí donde muchos fabricantes de software se han pegado de narices con el fracaso, comenzando por Ashton-Tate, el creador de dBase, y terminando incluso con la propia Borland y su abandono del BDE. ¿Causa del desastre? La inmensa complejidad de la interfaz resultante, casi siempre.

Hay dos técnicas importantes para realizar modificaciones sobre un cursor en el lado cliente, y la primera de ellas, siguiendo el orden histórico, es también la más sencilla. Usted puede modificar el *buffer* correspondiente al registro “activo”, siempre que avise antes al cursor acerca de sus intenciones. Antes de abandonar el registro activo, debe llamar a una función para enviar los cambios a la base de datos; si usted no la llama, el propio sistema se encargará de hacerlo. Desde el punto de vista de la implementación, es en ese momento cuando la interfaz de acceso genera las sentencias SQL de actualización necesarias. En resumen, cada actualización necesita un envío independiente de instrucciones. Es el mismo tipo de interfaz de actualización que utilizaba el lenguaje procedimental de dBase y sus secuelas.



El segundo modo de trabajo se conoce con el nombre de *cached updates*, o *actualizaciones en caché*, porque no obliga a aplicar los cambios realizados en el lado cliente antes de abandonar el registro activo. Para implementarlo, es necesario mantener una caché paralela para mantener el estado original de cada registro modificado o, como alternativa, el estado “actual” de esos mismos registros. Hace falta entonces enviar una señal explícita al software para que los cambios acumulados se apliquen sobre la base de datos. En el BDE, la operación se llama *ApplyUpdates*; en ADO, recibe el nombre de *UpdateBatch*.

La principal ventaja de este sistema de caché es que todas las instrucciones de actualización se envían en un mismo paquete, y eso ahorra tráfico en la red. En algunos sistemas como el BDE, como consecuencia secundaria, las actualizaciones en caché ofrecen al programador la oportunidad de tomar el control e indicar, de forma exacta, cómo desea que sus datos sean modificados. Quizás usted quiera que la eliminación de un cliente se haga por medio de un procedimiento almacenado, en vez de utilizar la simple instrucción **delete** que lanzaría el BDE. Pues bien, la única forma de hacerlo en el BDE es activando la caché de actualizaciones. En ADO, ni siquiera es posible.

## Con estado, o sin él

Hay otra consideración importante cuando se trata de diseñar una interfaz de acceso a datos, y que se olvida con mucha frecuencia. Supongamos que accedemos a una base de datos SQL y queremos los datos sobre nuestros cien mejores clientes. Abrimos un cursor a través de la conexión a la base de datos, y vamos recuperando registros paulatinamente. Nos da lo mismo ahora qué hacemos con esos registros: puede que los estemos mostrando en una rejilla, o que los estemos examinando de uno en uno. Incluso no nos interesa saber si se ha implementado un cursor bidireccional en el lado cliente o no.

Lo que nos interesa en este momento es saber cuánto tiempo debe estar abierto dicho cursor. ¿Por qué? Pues porque cada cursor abierto en un servidor SQL consume recursos: con toda seguridad, memoria RAM y espacio temporal en disco. Es también posible, de acuerdo al tipo de cursor y al nivel de aislamiento de transacciones (¿los recuerda?), que mientras esté activo el cursor, el sistema se vea obligado a bloquear de un modo u otro los registros del conjunto resultado.

Llamemos también al Abogado del Diablo<sup>17</sup> y escuchemos sus argumentos: sí, es cierto que un cursor consume recursos. También es cierto que las propias conexiones a bases de datos son muy caras. Pero también es costoso establecerlas. Si lo que este

---

<sup>17</sup> Ya sé que la mayoría de los abogados son personas decentes. Pero cada vez que veo a alguno de los que he tenido la desdicha de conocer, me llevo las manos a los bolsillos para comprobar que la billetera sigue en su sitio.

mentecato de Ian propone es cerrar las conexiones entre peticiones, se trata de una salvajada evidente y ...

Vale, ya está bien. Lo que estoy proponiendo es reciclar esas conexiones: cuando usted no las necesite, dígalos claramente, y podrán ser utilizadas por otros clientes. Sí, es muy probable que vuelva a necesitar una conexión dentro de 30 segundos, pero en ese momento usted recibirá una conexión de segunda mano. La principal condición para que esto funcione es que no dejemos dato alguno asociado a una conexión, como la posición dentro de un cursor. Cuando un libro pasa de mano en mano, no es conveniente dejar marcada la página con un trozo de cartulina; el siguiente lector tendría que hacer lo mismo, y al final el libro terminaría con más marcas que páginas. Es más sencillo si usted mismo apunta en su agenda en cuál párrafo de qué página abandonó la lectura por última vez.

La idea de utilizar interfaces sin almacenamiento del estado en el lado servidor es relativamente nueva, y ha sido inspirada por la forma en que funciona el protocolo HTTP, entre otras cosas. Cuando estudiemos Midas, también conocido como Data-Snap, veremos lo fácil que es implementar un servidor sin mantenimiento del estado, y cómo esta técnica multiplica el número máximo de usuarios simultáneos a los que puede dar respuesta un servidor de datos.

## La solución ¿perfecta?

Andar sobre el mar es posible ... si antes lo congelamos. La solución que le voy a mostrar en esta sección es la solución perfecta, con la única condición de que el esquema relacional nunca cambie, y que haya surgido de cuerpo entero y danzando de la mente de un superprogramador, como dicen que nació Palas Atenea a partir de un pensamiento de Zeus. La técnica en sí no tiene nombre propio: algunos fundamentalistas la conocen como “La Única Verdad Revelada”, aunque es más común encontrarla bajo algunos nombres más modestos, como *persistent frameworks* o simplemente *object frameworks* (no sé cómo traducir *framework*).

A uno de estos hijos del dolor le proponen desarrollar una simple aplicación de facturación. El genio entra en trance, e identifica una serie de objetos que interactúan en la futura aplicación. Hay clientes, productos, facturas, direcciones... aunque en su mente él las pronuncia con la mayúscula inicial. Y todos esos objetos se convierten en Objetos (de ahí lo de *object frameworks*).

A continuación, plasma su inspiración sobre papel. Crea una clase *CLIENTES*, y define propiedades para el nombre, los apellidos, el número de identidad y las otras tonterías habituales. Y lo principal: crea un método *Load* y otro método *Save*, para leer o guardar el objeto en una base de datos (de ahí lo de *persistent frameworks*). Por supuesto, no todo es un jardín de rosas, porque algún usuario malvado ha ideado que el cliente tenga asociada una lista de direcciones. Nada arredra a nuestro héroe, que programa una clase *TPersistentCollection* para leer desde el disco cualquier colección de

registros... siempre que derivemos una nueva clase a partir de ella y redefinamos 17 de los 19 métodos que posee. Por cierto, el programador que se siente a tres escritorios de distancia ha programado una clase parecida, pero el hijo del dolor no se fía de una persona que prefiere South Park antes que los Simpson, y que para colmo tiene una novia sexualmente satisfecha. ¿Qué clase de programador es ésa?

Tras largas semanas de mezclar sales y ácidos en su atanor, nuestro programador no ha diseñado aún ni una sola ventana, pero ya tiene listas todas las clases que van a participar en la aplicación. De repente, lo llama el jefe de proyecto a su despacho, y tienen una larga charla: el cliente está algo impaciente, y ha decidido cambiar alguna de las especificaciones. Por ejemplo, ahora hay que almacenar los datos del cónyuge de cada cliente, y el cónyuge a su vez puede ser un cliente, cada producto debe admitir distinto precio según la cantidad de unidades facturadas, cada factura... Entonces, para sorpresa de todos, el programador abre la ventana, lanza un alarido y se precipita desde la vigésima planta hacia el asfalto.

¿Imposible, verdad? Pues es el tipo de interfaz más utilizado por los programadores de Java. Eso, dicho sea de paso, es típico en Java: hay una solución ideal para un mundo ideal, que suele adaptarse al imperfecto mundo cotidiano y funcionar de pena, como podrá imaginar.

Menciono esta forma de trabajo no sólo para verter una dosis de veneno sobre la vida privada de los programadores de Java. Sobre todo quiero aclarar que no se trata de una forma disparatada de trabajar, aunque hay que reconocer que es muy poco productiva. Es una pena que Delphi y sus bibliotecas de clases no ofrezcan ni el más mínimo soporte en esta dirección. Y no podemos perder de vista que, aunque Delphi es un lenguaje orientado a objetos, estamos utilizándolo con bases de datos relaciones, que utilizan convenciones que se hallan en el extremo opuesto del espectro. Quizás algún día...

## Los contrincantes

Lo más sorprendente no es la variedad de sistemas de acceso a bases de datos que vamos a encontrar, sino la forma en que la “industria” ha logrado imponer hasta el momento a unos pocos de ellos.

- Sistemas de acceso nativo
- ODBC
- BDE
- OLE DB
- ADO
- DB Express

Dedicaré el resto de este capítulo a presentarlos.

## Sistemas de acceso nativos

Tenemos que comenzar por las interfaces de programación definidas por los propios fabricantes de los sistemas de base de datos. Por ejemplo, InterBase ofrece el módulo *gds32.dll* como interfaz de más bajo nivel. Oracle, a partir de la versión 8, nos propone el uso de un módulo similar: la *oci.dll*. Antes de la versión 7, la interfaz de acceso más eficiente para SQL Server era conocida con el nombre de *DB Library*, y era implementada por la *ntwdblib.dll*. Pero a partir de la versión 7, Microsoft la descartó en favor de OLE DB. Dentro de esta categoría entran también los motores de Paradox, dBase y Access.

Sobre estos motores nativos, se pueden crear componentes de acceso a datos especializados para Delphi. Los más conocidos:

- InterBase Express: comenzó siendo una colección de componentes *freeware*, pero fue adoptada por Borland en la versión 5 de Delphi.
- IB Objects, también para InterBase. Esta es una colección de pago, desarrollada en solitario por un programador llamado Jason Wharton ([www.ibobjects.com](http://www.ibobjects.com)). Lleva mucho tiempo en el mercado, y goza de buena fama.
- Direct Oracle Access, o DOA, para Oracle. Componentes desarrollados por una compañía holandesa, Allround Automations ([www.allroundautomations.nl](http://www.allroundautomations.nl)). Muy buenas críticas.

Suele decirse que los componentes basados en interfaces nativas son los que ofrecen mejor rendimiento, y casi siempre es cierto. Los componentes que llevan más tiempo en el mercado, como DOA e IB Objects, son de fiar. Ahora bien, no recomendaría a nadie la primera versión de una colección de componentes de acceso directo. El principal motivo es que, a pesar de los pesares, es bastante complicado crear descendientes de *TDataSet*; hay muchos subsistemas que implementar, y no todos están bien documentados. En particular, recele de la implementación del soporte para DataSnap (¡ha visto, no he dicho Midas!).

Otro aspecto a tener en cuenta, que puede ser un problema o no, es que al utilizar componentes de acceso nativo hacemos más difícil la migración de la aplicación a otra base de datos. Existe, además, el peligro de que aparezca una nueva versión de Delphi, y que el autor tarde en actualizar los componentes. En cualquier caso, si decide utilizar una colección de este tipo, asegúrese de disponer del código fuente... por lo que pueda pasar.

## ODBC: conectividad abierta

La más conocida de las interfaces de programación para bases de datos en Windows es ODBC, cuyas siglas significan *Open Database Connectivity*, y fue desarrollada por Microsoft, como era de esperar. Se ganó una mala reputación entre los programado-

res de Delphi por causas ajenas a ella misma. ODBC es una interfaz SQL y, evidentemente, el rendimiento de este tipo de interfaces cuando se accede a bases de datos de escritorio, como Paradox, dBase y FoxPro, deja mucho que desear. En la época en que muchas aplicaciones seguían utilizando el formato DBF, el programador de Delphi que comparaba el rendimiento “nativo” del BDE sobre estas bases de datos con el de ODBC, tenía motivos para quejarse de Microsoft.

Pero había otro malentendido común, más inmerecido si es posible. El BDE puede extraer sus datos indirectamente desde un controlador ODBC. Por supuesto, al configurar un sistema de este modo estamos interponiendo demasiadas capas entre la base de datos y la aplicación. Lo sorprendente es que este sistema funcionase, aunque dejase mucho que desear su velocidad de acceso.

A pesar de que ODBC ha sido sustituido por OLE DB y ADO como interfaces de acceso, sigue siendo importante porque algunas bases de datos no disponen todavía de controladores OLE DB específico. Un programador de Delphi, en tal caso, podría utilizar ADO Express, mediante un controlador especial de OLE DB que utiliza ODBC como fuente de datos. Claro, esto supondría nuevamente capas y más capas de software y bastante lentitud. En tal caso, lo recomendable es utilizar otra colección de componentes llamada ODBC Express ([www.odbcexpress.com](http://www.odbcexpress.com)), que utiliza directamente ODBC como interfaz nativa.

## OLE DB: el esperado sucesor

ODBC, o al menos el núcleo básico de esta interfaz, se diseñó antes de que las técnicas de programación COM alcanzaran su madurez. La interfaz de programación es un conjunto de funciones, tipos de datos planos y constantes. Por una parte, un sistema así es complicado de mantener y extender por parte de su propio fabricante. Por otro lado, es una interfaz demasiado complicada para ser utilizada por lenguajes de *script*, como VBScript y JScript. Entonces alguien en Microsoft musitó un par de palabras mágicas y... ¡ODBC ha muerto, viva OLE DB!

OLE DB consiste principalmente en un conjunto de tipos de interfaz, y alguna que otra rutina auxiliar, para acceder a *casi* cualquier tipo de base de datos que podamos imaginar. No sólo a lo que se entiende por tal tradicionalmente; hasta la lista de directorios virtuales manejados por Internet Information Server puede ser manejada desde OLE DB. En particular, existe el previsible controlador sobre ODBC que, como comprenderá, no se debe utilizar si hay alguna otra solución a mano.

Ya sabemos que una interfaz existe para ser implementada por una clase, o para ser utilizada. Los controladores OLE DB, *proveedores* en la jerga de Microsoft, son los encargados de implementar las interfaces de acceso. Además de ofrecer controladores para sus propios productos, Microsoft también ha programado uno para Oracle, aunque con algunas limitaciones para el trabajo con tipos blob. Al principio Oracle

fue reacio a entrar en el juego de Bill, pero al final triunfó la cordura y desarrollaron su propio controlador.

En teoría, podríamos programar directamente con OLE DB, pero hay muy pocos locos que se atrevan, y no me cuento entre ellos. Menciono esta posibilidad porque he visto anunciar varias veces el desarrollo de componentes de acceso directo a OLE DB para Delphi, aunque hasta el momento no conozca ninguno que esté a la venta.

## ADO: sencillez y potencia

Precisamente es la potencia de OLE DB lo que más dolores de cabeza produjo a Microsoft. Un programador experimentado puede dominar OLE DB con cierta facilidad si utiliza el lenguaje de programación adecuado. Estoy pensando en Delphi, por supuesto, pero incluso un Visual C++ bastaría<sup>18</sup>. Sin embargo, manejar las decenas de interfaces necesarias desde el lenguaje estrella de Microsoft, mi nunca bien detestado Visual Basic, es tarea para masoquistas o alopécicos en ciernes. También encontraríamos dificultades si quisiéramos utilizar OLE DB desde lenguajes útiles pero sencillos, como JScript. Y con esta última justificación sí estoy de acuerdo.

Consecuentemente, Microsoft sacó de su sombrero de copa un conejo llamado ADO (porque si lo hubiesen llamado ADA habría sido una coneja, claro está). ADO está basado en unas pocas interfaces de automatización, derivadas de *IDispatch*: *Connection*, *RecordSet* y cosas así, que en el capítulo correspondiente veremos lo fáciles que son de usar. ADO no se limita a enmascarar las interfaces nativas de OLE DB, sino que implementa algunas golosinas en el lado cliente, como un estupendo “motor de cursores”, que consiste realmente en una caché de registros.

En Delphi 5, Borland introdujo componentes para trabajar directamente con ADO, y llamó ADO Express a la colección. Al parecer, existen problemas legales con este nombre, y en la versión 6 han “intentado” renombrarlos como dbGo. Veo complicado que un nombre tan ridículo tenga éxito.

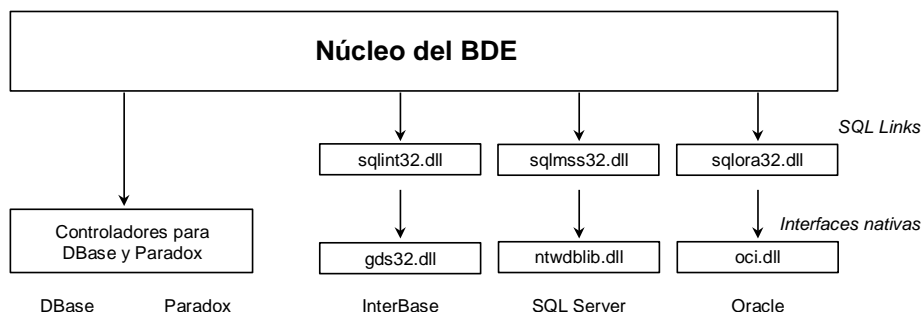
## BDE: el Motor de Datos de Borland

La primera incursión de Borland en el mundo de las bases de datos fue la compra de Paradox a una compañía cuyo nombre ya he olvidado; parte importante de la compra fue una biblioteca de funciones, llamada por aquel entonces el Motor de Paradox. Sabemos que comprar puede convertirse en una adicción: a la compra de Paradox siguió la de InterBase, y la de dBase. Al final, Borland se vio con varios productos de acceso a datos muy diferentes en sus manos, y decidió implementar una interfaz de acceso común para todos ellos, utilizando como base el motor de Paradox. Tuvieron éxito donde otras empresas, como la propia Ashton-Tate, creadora del dBase, habían fracasado. Y así nació el Borland Database Engine, o BDE.

---

<sup>18</sup> Lo de “visual” era una broma.

A grandes rasgos, el siguiente diagrama representa la arquitectura interna del BDE:



Tendremos tiempo para hablar de las características técnicas del BDE. Aquí sólo quiero adelantarte una mala noticia: hay *bugs* muy importantes en el BDE, que llevan mucho tiempo sin resolverse, y que a estas alturas es muy probable que acompañen al producto hasta su muerte. Porque Borland ha decidido que mantener vivo el monstruo que ellos mismos han creado es demasiado caro; además, sería demasiado complicado migrarlo a Linux (migrar cualquier cosa a Linux *ya* lo es), y visto el cariño que profesan a su Kylix...

#### PROBLEMA SIN RESOLVER

El bug más importante está relacionado con las actualizaciones en caché, un recurso que se introdujo en la primera versión del BDE para 32 bits. En concreto, con las relaciones maestro/detalles en caché. La única forma sencilla de evitarlo consiste en renunciar a las actualizaciones en caché, y utilizar conjuntos de datos clientes, soportados por DataSnap, en sustitución. Quiero aclarar, sin embargo, que el sistema de caché de ADO presenta el mismo problema en su versión actual.

## DB Express: de vuelta a los orígenes

Borland padeció su correspondiente calvario con el BDE. Tras el fracaso técnico y comercial de Visual dBase 7, se deshicieron de gran parte de la plantilla relacionada con dBase, Paradox y un fallido intento de sistema RAD para el desarrollo en Internet llamado IntraBuilder. Con el agua de la bañera tiraron también al niño, y de repente se encontraron con problemas para mantener funcionando al BDE. Como he dicho, sin embargo, el pretexto para rematar al moribundo lo dio la aparición de Kylix para Linux.

Así que, puestos a definir una nueva interfaz de acceso que funcionase tanto en Windows como en Linux, los cerebros pensantes de Borland dieron un paso inteligente: ¿cuál es el principal problema del BDE? Intentar serlo todo: una interfaz de acceso a SQL, un motor de bases de datos locales, una caché en el lado cliente... Por lo tanto, la solución sería crear una interfaz de acceso minimalista.



¿Qué podemos sacrificar? En primer lugar, el motor de bases de datos locales: una sentencia de muerte para Paradox y dBase, pero muy en el espíritu de la época. También Microsoft está intentando abandonar Access, y FoxPro ya es un espectro que vaga por nuestros ordenadores rechinando sus cadenas. ¿Algo más para tirar por la borda? ¡Uf!, ese complicado sistema de caché que, en definitiva, nunca funcionó bien. ¿Con qué nos quedamos, entonces? ¡Con DB Express!

Estos son los principales rasgos de DB Express, junto con una breve justificación:

- 1 El número de controladores necesarios para una conexión es mínimo, y puede usarse como una DLL o enlazarse estáticamente al ejecutable.
- 2 Todos los conjuntos de datos son unidireccionales. Solamente se mantiene en memoria el registro activo del cursor, y no se utiliza una caché en el cliente.
- 3 Todos los conjuntos de datos son de sólo lectura. Para modificar registros hay que lanzar sentencias de modificación directamente sobre la base de datos.

Esta interfaz mínima es estupenda para desarrollar aplicaciones para Internet, o si desea crear su propio sistema de persistencia orientado a objetos. Pero para la programación de aplicaciones con interfaz gráfica “clásica” sería un desastre... si no existiera DataSnap. Este otro sistema puede proporcionar la caché de navegación bidireccional, y es capaz de ocuparse con éxito de la modificación a través del conjunto de datos.

En la presente edición de Delphi, hay controladores de DB Express para InterBase, Oracle, DB2 y MySQL. El de InterBase es bastante decente y da pocos problemas. Hay algunas limitaciones en el controlador de Oracle, pues no puede trabajar todavía con tablas anidadas, aunque sí admite campos ADT; supongo que lo corregirán en la siguiente versión. No he probado personalmente el acceso a DB2, pero las referencias son positivas. Y ha sido el controlador de MySQL el que más problemas ha dado, pero apostaría que la culpa es del propio MySQL, que tiene unas cuantas manías de mucho cuidado. Borland, además, ha prometido un controlador para SQL Server.



## MyBase: navegación

**L**A PEOR PESADILLA DE UN ESCRITOR técnico es la necesidad de organizar los capítulos de un libro sin que existan referencias circulares. A veces tienes dos temas que tratar: llamémosles *A* y *B*. Para comprender *B*, debes conocer bien *A*. Pero para dominar *A* hay que tener, aunque sea, nociones de *B*. Ahora mismo, estoy lidiando con una de las referencias circulares de Delphi: la que engloba los conjuntos de datos con los objetos de acceso a campos. No podemos explicar los campos sin haber mencionado antes los conjuntos de datos. Sin embargo, no podremos presentar ejemplos decentes sobre los conjuntos de datos si se nos prohíbe utilizar campos.

Romperemos el nudo gordiano presentando primero una clase especial de conjuntos de datos, *TClientDataSet*, que permite implementar sencillas bases de datos para un solo usuario, en un formato bautizado por Borland como *MyBase*. *TClientDataSet* es, en realidad, una pieza muy importante del acceso de datos en Delphi, porque puede servir como caché de datos extraídos de otros sistemas, como veremos en los capítulos dedicados a *DataSnap*.

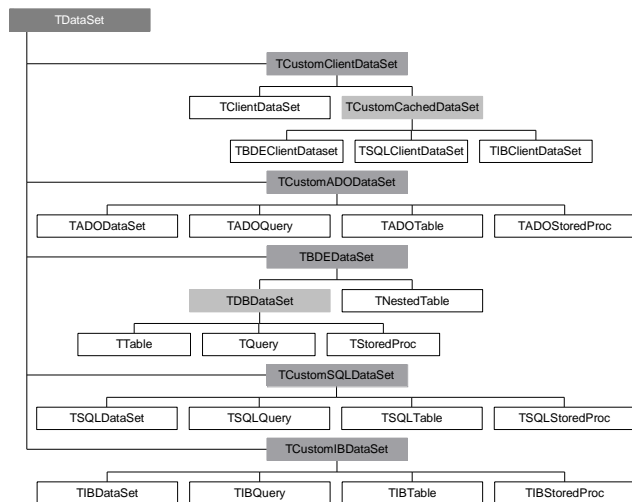
Por último, aprovecharemos para explicar aquellos métodos, propiedades y eventos que son comunes a todos los conjuntos de datos de Delphi.

### Jerarquía de los conjuntos de datos

Delphi utiliza la palabra *dataset*, o conjunto de datos, para referirse a varias clases que representan *relaciones*, o lo que es igual, colecciones de registros. Para Delphi, un conjunto de datos es una noción abstracta, que deja sin especificar dos características importantes: de dónde provienen esos registros, y qué método utilizamos para su obtención.

En las primeras versiones de Delphi, cuando Borland decía “conjunto de datos”, se refería realmente a colecciones de registros obtenidos mediante el Motor de Datos de Borland, o BDE; la única interfaz de acceso a datos que Delphi podía utilizar por aquel entonces. Las cosas cambiaron con la aparición de Delphi 3, pues la biblioteca de clases para acceso a datos fue remodelada para que pudiésemos manejar datos provenientes de otras fuentes.

El siguiente diagrama muestra la jerarquía de las clases que implementan conjuntos de datos en Delphi 6:



La raíz de todo el árbol es la clase abstracta *TDataSet*. He coloreado las clases abstractas con distintos tonos de grises, reservando el color blanco para las clases “concretas”. Usted trabajará de forma directa sólo con estas últimas, pero muchos métodos y eventos utilizan parámetros cuyos tipos pertenecen a alguna de las clases intermedias.

La primera ramificación de la jerarquía crea cuatro grupos de componentes:

### 1 *TCustomClientDataSet*

Los descendientes de esta clase manejan datos almacenados en la memoria RAM del ordenador. Esos datos se pueden crear endogámicamente, definiendo el esquema relacional desde cero, y añadiendo entonces registros al conjunto de datos. Pero lo más habitual es que estos componentes sirvan de caché a otras interfaces de acceso a datos. En ese caso, el componente imita la estructura y los datos asociados de otro conjunto de datos, que puede estar situado dentro de la misma aplicación, en otra aplicación e incluso en otro ordenador.

Este capítulo está dedicado al estudio de la clase *TClientDataSet*.

### 2 *TBDEDataSet*

Bajo esta rama se agrupan los componentes de acceso a datos más antiguos de Delphi: aquellos que obtienen sus datos a partir del BDE. Observe que esta rama, lo mismo que la anterior, tiene una estructura más compleja que las restantes.

### 3 *TCustomSQLDataSet*

Esta clase y sus derivadas se han introducido en Delphi 6, y recuperan sus datos mediante DB Express. En su versión más reciente, permiten trabajar con datos en InterBase y MySQL; si utiliza la versión Enterprise, también acepta DB2 y Oracle.

### 4 *TCustomADODDataSet*

Estas clases utilizan ADO como su fuente de datos. Es el mejor sistema para trabajar con SQL Server y Access.

### 5 *TCustomIBDataSet*

Los datos se piden directamente a InterBase, a través de su interfaz de programación de más bajo nivel. Se incluye por compatibilidad con versiones anteriores de Delphi, en las que no existía DB Express. No me gusta, en absoluto.

#### NOTA

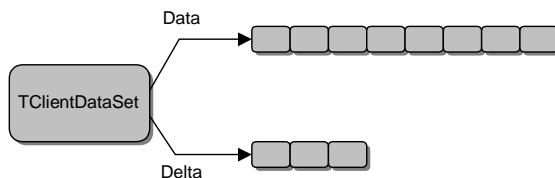
Delphi incluye otras clases derivadas directa o indirectamente a partir de *TDataSet*, puede comprobarlo examinando el gráfico de herencia impreso que acompaña a Delphi. En particular, destacan varias clases de la rama del BDE, cuyos nombres contienen las siglas *DR*: *Data Repository*, o depósito de datos. Son utilizadas por el Entorno de Desarrollo para el manejo del Diccionario de Datos, una herramienta de diseño que, lamentablemente, sólo puede utilizarse con los conjuntos de datos del BDE.

## Conjuntos de datos clientes

Para nuestra presentación y para los experimentos iniciales utilizaremos, como he advertido antes, componentes de la clase *TClientDataSet*, a los que llamaremos *conjuntos de datos cliente*. La siguiente imagen corresponde a la página *Data Access* de la Paleta de Componentes de Delphi. El segundo componente, contando desde la izquierda, es el que ahora nos interesa:



*TClientDataSet* almacena sus datos en dos vectores situados en memoria RAM. Al primero de estos vectores se accede mediante la propiedad *Data*, de tipo *OleVariant*, y contiene los datos leídos inicialmente por el componente; después veremos desde dónde se leen estos datos. La segunda propiedad, del mismo tipo que la anterior, se denomina *Delta*, y contiene los cambios realizados sobre los datos iniciales. Los datos reales de un *TClientDataSet* son el resultado de mezclar el contenido de las propiedades *Data* y *Delta*.



Gracias a esta estructura de datos se implementan dos importantes operaciones:

- 1 Se pueden deshacer los cambios siguiendo un orden cronológico. Supongamos que actualizamos el registro de Ian Marteens para subirle el salario, y luego vamos al registro de Gill Bates para echarlo a la calle. Ambas modificaciones se almacenan en *Delta*: primero la que afecta a mi salario, y luego la carta de despido. Si Mr. Bates amenaza con quitarle el bozal a sus abogados y el operador de datos pulsa el botón de deshacer, primero recupera su puesto de trabajo. Y si sigue presionando al cobarde operador, mi salario recupera su lastimoso estado inicial.
- 2 Para cada campo de cada registro, siempre sabemos cuál era el valor “inicial” y cuál es su valor actual. Esta posibilidad es crucial para poder generar automáticamente instrucciones SQL de modificación, cuando el conjunto de datos cliente se utiliza como sistema de caché de una base de datos SQL.

Otra interesante característica de *TClientDataSet* es que permite utilizar *tablas anidadas* (*nested tables*), de manera parecida a la utilizada por Oracle a partir de su versión 8. Por ejemplo, si vamos a trabajar con una tabla de clientes, podemos definir un campo que se llame *Direcciones*, y hacer que ese campo almacene en su interior una lista de cero o más direcciones. Cada dirección individual, por supuesto, sería a su vez un registro con campos para la ciudad, el país, el código postal, etc. Cuando expliquemos el funcionamiento de los conjuntos de datos clientes en DataSnap, verá con claridad que esta posibilidad tiene importancia extrema.

La implementación de la clase *TClientDataSet* se encuentra en la unidad *DBClient* de Delphi. Pero los métodos más importantes de la clase efectúan llamadas a un núcleo de código residente en el módulo dinámico *midas.dll*. La versión de este fichero que viene con Delphi 6 ocupa unos 287KB. Para distribuir una aplicación que utilice conjuntos de datos clientes tenemos dos posibilidades. La obvia consiste en copiar el fichero ejecutable de la aplicación además de *midas.dll*. Pero Delphi 6 nos permite también integrar el código de ese módulo en nuestro ejecutable, para simplificar su instalación. Para lograrlo basta con añadir la unidad *MidasLib* en alguna de las cláusulas **uses** del proyecto.

## Datos provenientes de ficheros

Para no tener que explicar cómo se define el esquema relacional de un conjunto de datos cliente y cómo se le añaden filas desde un programa, veremos ahora cómo

podemos leer el esquema y el contenido de uno de estos componentes desde un fichero que ya contiene datos. Comenzaremos con el siguiente par de métodos:

```
procedure TClientDataSet.LoadFromFile(const Fichero: string = '');  
procedure TClientDataSet.SaveToFile(const Fichero: string = '';  
    Format: TDataPacketFormat = dfBinary);
```

Si queremos poder leer un fichero, primero tendremos que crearlo, por lo que comenzaremos examinando *SaveToFile*. En el primer parámetro pasamos un nombre de fichero, y en el segundo especificamos en qué formato queremos que se graben el esquema relacional y los datos del *TClientDataSet*. Delphi 6 soporta tres diferentes formatos:

Formato	Significado
<i>dfBinary</i>	Formato binario, exclusivo de Borland
<i>dfXML</i>	Formato XML, también definido por Borland
<i>dfXMLUTF8</i>	Similar al anterior, con alfabeto codificado en UTF8

¡Mucho cuidado con los dos últimos formatos! Los programadores que todavía no se han familiarizado con XML tienden a creer que “todos los gatos son pardos” ... al menos en la oscuridad. Alguien puede oír que ADO permite también guardar el resultado de una consulta en formato XML ... y “deducir” que un *TClientDataSet* puede leer directamente esos ficheros.

En realidad, y hasta donde tengo conocimiento, el formato XML utilizado por Borland no corresponde a ninguna definición estándar y es, como era de esperar, diferente del utilizado por Microsoft en ADO. Pero eso no debe preocuparle, porque más adelante veremos cómo un *TClientDataSet* puede leer virtualmente casi cualquier fichero XML que represente un conjunto de datos; solamente que la técnica no va a ser tan directa.

UTF-8 es un formato de representación de caracteres Unicode. En la representación más sencilla, cada carácter ocupa sus dos bytes preceptivos. Esto es un desperdicio de espacio, especialmente para textos en inglés, porque los caracteres más frecuentes pertenecerán también al ASCII restringido de 7 bits. Por ese motivo existe UTF-8, en el que un carácter puede ocupar uno, dos o tres bytes. Esos caracteres más frecuentes ocupan un solo byte.

Es muy importante que comprenda que la grabación de datos en un fichero es un proceso cerrado que requiere el acceso en exclusiva al fichero. Con esto quiero decir que:

- No es posible grabar “sólo un trocito” del contenido del conjunto de datos. Siempre se guarda el contenido completo.
- Además, no se mezcla el contenido de *Data* con el de *Delta*, a no ser que lo hagamos explícitamente, llamando a un método que estudiaremos en su debido momento: *MergeLogFile*.

- El contenido del fichero anterior a la grabación se destruye.
- Para iniciar la grabación, el método exige el acceso en modo exclusivo al fichero. No puede estar en uso en ese momento.

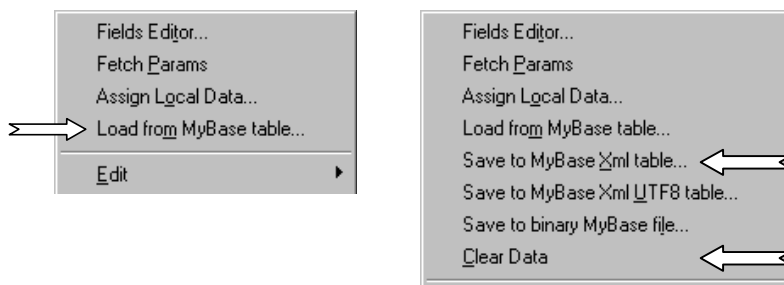
El método que más nos interesa ahora es *LoadFromFile*, que sirve para cargar dotar de contenido a las propiedades *Data* y *Delta* de un conjunto de datos. Como podemos ver, solamente se necesita el nombre del fichero con los datos, porque el propio componente puede determinar el formato utilizado.

Incluso podemos omitir el nombre del fichero, ya que el parámetro admite como valor por omisión una cadena vacía. Y si regresa a la declaración de *SaveToFile* verá que también permite que le pasemos una cadena vacía. En ese caso, se utiliza el valor de la propiedad *FileName* para determinar qué fichero queremos leer o sobre qué fichero queremos escribir:

```
property TClientDataSet.FileName: string;
```

En realidad, *FileName* tiene un propósito más amplio: si el conjunto de datos cliente tiene asignado un valor razonable en esta propiedad, el componente llama a *LoadFromFile* cuando lo activamos, llamando a *Open* o asignando *True* a su propiedad *Active*. Cuando cerramos el fichero, asignando *False* a *Active* o ejecutando su método *Close*, se llama automáticamente a *SaveToFile* para guardar las posibles modificaciones.

Es interesante saber que las operaciones de lectura y escritura de ficheros pueden también efectuarse en tiempo de diseño, desde el menú de contexto que aparece al pulsar el botón derecho del ratón sobre un *TClientDataSet*.

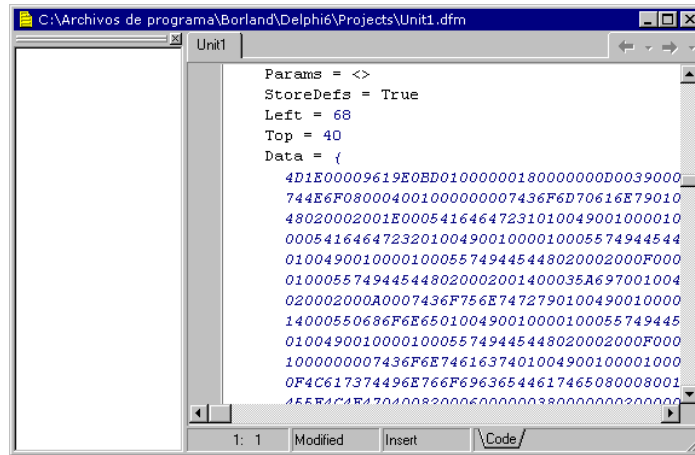


El menú de la izquierda es el que se muestra inicialmente, cuando aún no hemos cargado datos en el objeto. La opción que nos interesa es *Load from MyBase table*. El menú de la derecha aparece una vez que se ha efectuado la carga. Como se puede apreciar, hay tres comandos de escritura, uno para cada formato de almacenamiento. Si quisiéramos eliminar los datos del conjunto de datos, podríamos utilizar el comando *Clear Data*, que he señalado con una flecha en la imagen de la derecha.

Cuando se llama exitosamente a *LoadFromFile*, la propiedad *Active* del conjunto de datos cambia su valor a *True*. Si cargamos datos en tiempo de diseño, dejamos el



valor *True* en *Active* y guardamos el formulario, los datos leídos se almacenan en formato binario dentro del fichero *dfm* asociado al formulario. Puede comprobarlo repitiendo los pasos enumerados y ejecutando el comando *View as text*, del menú de contexto del formulario:



Eso quiere decir que los datos correspondientes van también a parar al interior del fichero ejecutable. No es una técnica que sea siempre recomendable, porque el tamaño del ejecutable crece mucho y es complicado reemplazar esos datos una vez compilada la aplicación. Pero puede servir para crear aplicaciones sencillas que deban utilizar bases de datos sólo lectura. Por ejemplo, para distribuir un catálogo de productos en CDROM.

## Conexión con componentes visuales

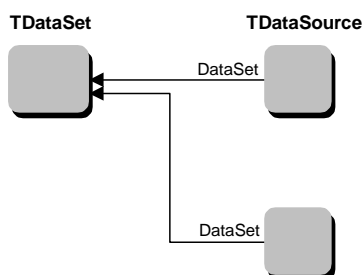
Un conjunto de datos, sea del tipo que sea, no puede mostrar directamente los datos con los que trabaja. Para comunicarse con los controles visuales, el conjunto de datos debe tener asociado un componente auxiliar, perteneciente a la clase *TDataSource*. Traduciré esta palabra como *fuentes de datos*, pero trataré de utilizarla lo menos posible, pues el parecido con “conjunto de datos” puede dar lugar a confusiones.

Un objeto *TDataSource* es, en esencia, un emisor de notificaciones. Los objetos conectados a este componente reciben avisos sobre los cambios de estado y de contenido del conjunto de datos controlado por *TDataSource*. Las dos propiedades principales de *TDataSource* son:

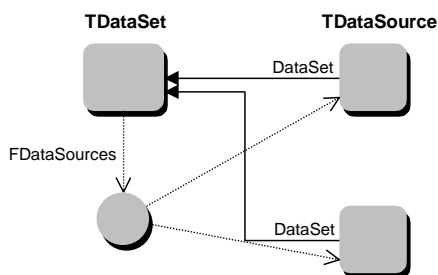
- *DataSet*: Es un puntero, de tipo *TDataSet*, al conjunto de datos que se controla.
- *AutoEdit*: Cuando es *True*, el valor por omisión, permite editar directamente sobre los controles de datos asociados, sin tener que activar explícitamente el modo de edición en la tabla. Los modos o estados de los conjuntos de datos se estudian en el capítulo 22.

A la fuente de datos se conectan entonces todos los *controles de datos* que deseemos. Estos controles de datos se encuentran en la página *Data Controls* de la Paleta de Componentes, y todos tienen una propiedad *DataSource* para indicar a qué fuente de datos, y por lo tanto, a qué conjunto de datos indirectamente se conectan. En el presente libro dedicaremos un par de capítulos al estudio de los controles de datos.

Es posible acoplar más de una fuente de datos a un conjunto de datos, incluso si los componentes se encuentran en formularios o módulos de datos diferentes. El propósito de esta técnica es establecer canales de notificación independientes, algo particularmente útil para ciertas técnicas que tratan con relaciones *maestro/detalles*. Un mecanismo oculto empleado por Delphi hace posible la conexión de varias fuentes de datos a un conjunto de datos. Cuando usted engancha un *data source* a un *TDataSet*, esto es lo que ve:



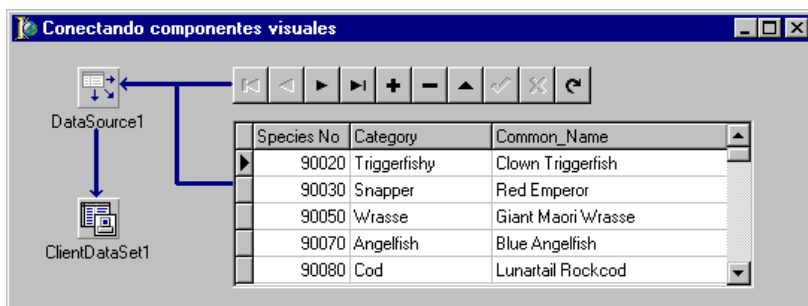
Sin embargo, existe un objeto interno, propiedad del conjunto de datos, que contiene una lista de fuentes de datos y que completa el cuadro real:



Un ejemplo común de conexión de componentes es utilizar una *rejilla de datos*, de la clase *TDBGrid*, para mostrar el contenido de una tabla o consulta, con la ayuda de una *barra de navegación* (*TDBNavigator*) para desplazarnos por el conjunto de datos y manipular su estado. Esta configuración la utilizaremos bastante en este libro, por lo cual nos adelantamos un poco mostrando cómo implementarla:

Objeto	Propiedad	Valor
<i>ClientDataSet1: TClientDataSet</i>		
<i>DataSource1: TDataSource</i>	<i>DataSet</i>	<i>ClientDataSet1</i>
<i>DBGrid1: TDBGrid</i>	<i>DataSource</i>	<i>DataSource1</i>

Objeto	Propiedad	Valor
<i>DBNavigator1: TDBNavigator</i>	<i>DataSource</i>	<i>DataSource1</i>

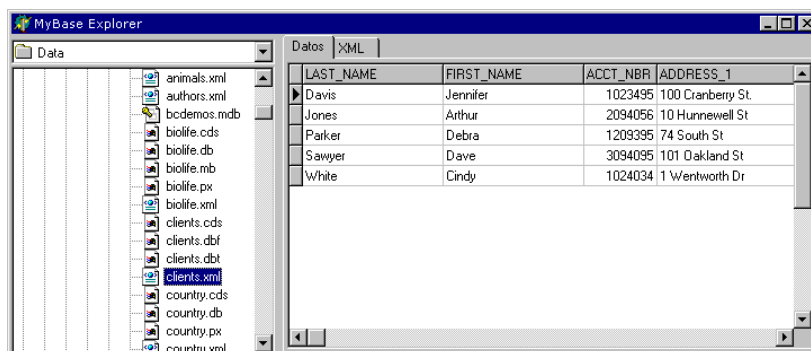


Es tan frecuente ver juntos a estos componentes, que le recomiendo que guarde la combinación como una plantilla de componentes.

## Una sencilla demostración

Comenzaremos por un ejemplo muy sencillo, para demostrar la carga de datos a partir de ficheros XML:

- Entre en Delphi y ejecute el comando de menú *File | New | Application*.
- Dividiremos la superficie de la ventana principal en dos partes: en la izquierda exploraremos el contenido del disco, y en la derecha examinaremos el contenido de los ficheros XML que se instalan con Delphi 6.
- Traiga un *TPanel (Standard)* y cambie su alineación (*Align*) a *alLeft*. Luego traiga un componente *TSplitter (Additional)* y cambie su propiedad *Width* a tres píxeles de ancho. Finalmente, añada un *TPageControl (Win32)*, asigne *alClient* en su propiedad *Align* y cree dos páginas en su interior, seleccionando el comando *New page* desde el menú de contexto del componente.



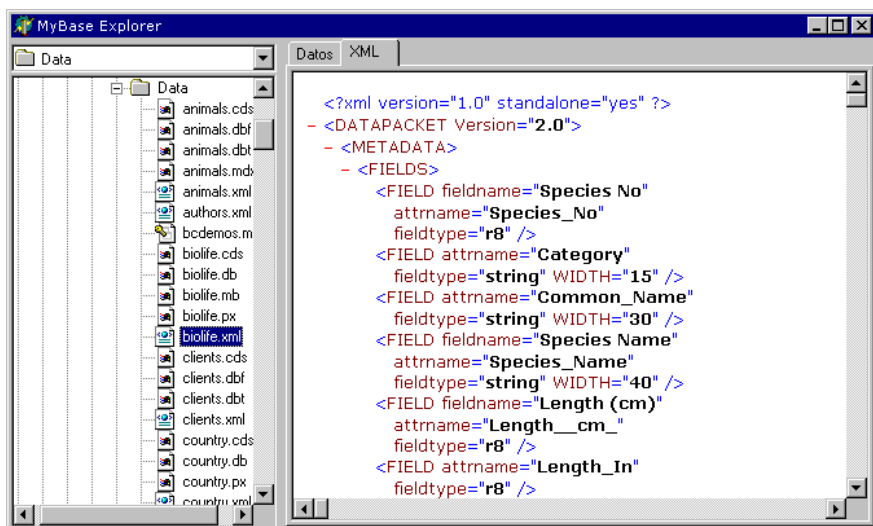
- En este paso prepararemos el panel izquierdo para la exploración. Vamos a la página *Samples* de la Paleta de Componentes, para traer un *TShellComboBox* y un

*TShellTreeView*. El primero de ellos, un combo, no tiene propiedad *Align*, que nos sería muy útil para alinear el control en la parte superior del panel. Pero lo solucionaremos añadiendo la opción *akRight* a su propiedad *Anchors*. Lo mismo haremos con el árbol, añadiendo esta vez *akRight* y *akBottom*; podríamos asignar *alClient* en su propiedad *Align*, pero eso escondería el combo.

- Seleccionamos entonces la propiedad *ObjectTypes* del árbol y añadimos la opción *otNonFolders*, para que además de carpetas aparezcan también ficheros. Por último, hacemos que la propiedad *ShellTreeView* del combo apunte al árbol compañero. Y ya está lista la navegación.

Es el momento de ocuparnos del control de páginas. Comenzamos por la primera página:

- Vamos a la página *Data Access* de la Paleta de Componentes y traemos un *TClientDataSet*. Como no se trata de un componente visual, da lo mismo donde lo coloquemos. Pero he descrito su configuración en este punto porque haremos que su contenido se muestre en la primera página. No modificaremos ahora ninguna de sus propiedades.
- Para poder *ver* el contenido de un conjunto de datos, traemos un *TDataSource* de la misma página *Data Access*. Cambiamos entonces su propiedad *DataSet* para que apunte a *ClientDataSet1*, el componente que añadimos en el paso anterior.
- Finalmente, vamos a la página *Data Controls*, seleccionamos un componente *TDBGrid* y lo dejamos caer en la primera página del control de páginas. Entonces cambiamos a *alClient* su propiedad *Align*.



Próxima parada: la segunda página del control de páginas. La utilizaremos para mostrar el contenido del fichero XML que seleccionemos. Podríamos utilizar un control

de edición de texto, pero solamente por fastidiar un poco, vamos a traer un componente *TWebBrowser*, que encontrará en la página *Internet* de la Paleta de Componentes.

El componente *TWebBrowser* encapsula el control ActiveX de visualización HTML utilizado por Internet Explorer. Para poder utilizarlo, debe tener instalado en su ordenador una versión de Internet Explorer igual o superior a la 4.0. Para ser más exactos, el control reside en el fichero *shdocvw.dll*. Debe cambiar la alineación del control a *alClient*, para que ocupe todo el interior de la página.

Ya casi hemos terminado, pues sólo nos queda seleccionar el árbol de ficheros e interceptar su evento *OnDbClick*:

```
procedure TwndPrincipal.ShellTreeView1DbClick(Sender: TObject);
var
  FName: string;
begin
  FName := ShellTreeView1.Path;
  if SameText(ExtractFileExt(FName), '.XML') then
    begin
      ClientDataSet1.LoadFromFile(FName);
      WebBrowser1.Navigate(FName);
    end;
end;
```

Ya conocemos qué es lo que hace *LoadFromFile*. He utilizado además el método *Navigate* de *WebBrowser1*, que hace eso mismo que está usted imaginando. Dejo a su cargo añadir un botón o comando de menú para grabar los posibles cambios del conjunto de datos en disco, con la ayuda del método *SaveToFile*.

## MyBase

Al parecer, alguien en Borland contrajo una misteriosa enfermedad endémica en el mundo Pokemon: la de poner nombres solemnes a cualquier tontería. Supongamos que Pikachu, el animalucho amarillo que lanza descargas eléctricas, pilla un catarro, estornuda y salpica a uno de los malos, que entonces resbala y cae por un barranco, pegándose un tortazo que lo aleja de la tele hasta el siguiente episodio. Pues bien, no se trataba de un simple estornudo, señor mío, sino del famoso Ataque Moco Resplandeciente. Las estupideces que acabo de escribir, por poner otro ejemplo, no serían una chorrada sin más, sino la temible Chorrada Trueno de Acero de las 15:40.

Borland padece los mismos síntomas. La técnica que acabamos de mostrar, que permite leer un conjunto de datos desde un fichero, editarlo en memoria y finalmente guardarlo de vuelta al disco, ha recibido nombre propio: MyBase. A esta manía debemos agregar los serios problemas de identidad que últimamente hemos presenciado: toda una temporada negando llamarse Borland y adoptando la personalidad de una tal Inprise. Para rematar, ya Midas no es Midas, sino DataSnap, y al empleado que sorprenden utilizando el antiguo apelativo le administran descargas eléctricas, como las de Pikachu, en las plantas de los pies...

Llámesese como se llame, el trabajo con MyBase es diferente al utilizado por las bases de datos de escritorio tradicionales. Imagine una aplicación escrita, en los tiempos de MSDOS en Clipper, dBase o algún sistema similar. En concreto, suponga que está trabajando con el “fichero” (terminología de la época) de clientes. Ante usted tiene un registro con datos de un cliente, o si el programador ha utilizado un *browser*, una lista de esos registros. Cuando usted modifica los datos de un cliente, la correspondiente grabación se efectúa lo antes posible. Eso quiere decir que, si la aplicación ha sido programada con cierto cuidado, el registro ha sido bloqueado antes de que a usted se le permita efectuar esos cambios. La idea general es: usted está trabajado directamente sobre zonas físicas de un fichero de datos.

En Delphi todavía es posible trabajar de esa manera, sobre todo cuando se utiliza dBase o Paradox. Pero ya no es una forma de trabajo universalmente aconsejable, porque es muy difícil lograr que un número considerable de usuarios utilicen de modo concurrente la base de datos. Voy a explicar la “metáfora” de interacción de MyBase, porque es también utilizada por las aplicaciones basadas en DataSnap, e incluso por muchas aplicaciones que se ejecutan a través de Internet.

En MyBase la metáfora es otra: usted carga un fichero, o parte de él, en su instancia de la aplicación, del mismo modo que Word o Excel permiten leer documentos y hojas de cálculos. La única diferencia es que MyBase, cuando se utiliza en asociación con DataSnap, permite leer partes aisladas del documento, que en este caso sería una tabla física o virtual. Los cambios que usted pueda realizar son similares a las modificaciones que se realizan al editar un documento Word; son modificaciones que se almacenan en la memoria RAM. Para que esas modificaciones se hagan permanentes, es necesario guardar el fichero a la vuelta. Insisto: con DataSnap, que estudiaremos más adelante, es posible también guardar modificaciones en fragmentos del origen de los datos, e incluso establecer sofisticados sistemas de *conciliación*, para coordinar el trabajo de varios usuarios concurrentes. Con MyBase puro y duro, que es lo que estamos analizando ahora, solamente podemos leer y escribir el fichero completo.

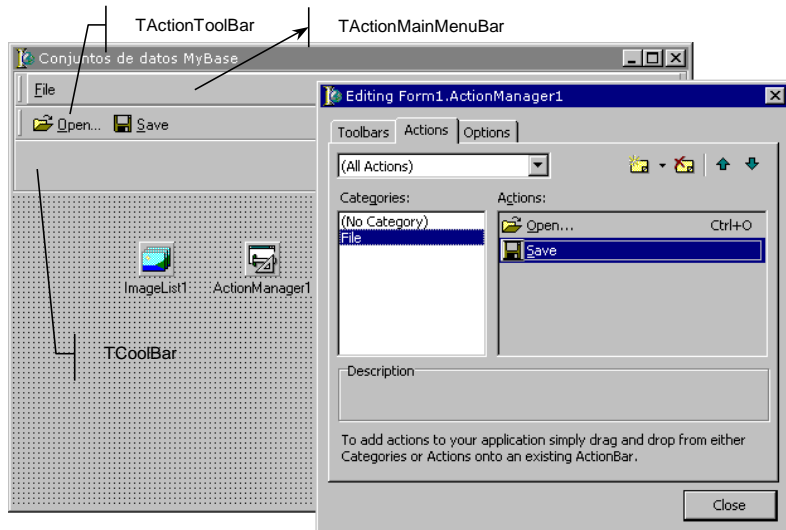
## Un ejemplo con acciones

Actuaremos consecuentemente, y desarrollaremos el núcleo inicial de una aplicación que trabaje con ficheros de MyBase utilizando la metáfora del procesador de textos. De paso, aprovecharemos el ejemplo para explicar el trabajo con *acciones* y con el nuevo componente *TActionManager*.

Inicie una nueva aplicación, y sobre la ventana principal añada los siguientes componentes:

- 1 *TImageList*, que se encuentra en la página *Win32*.
- 2 *TActionManager*, que está escondido casi al final de la página *Additional*. Conecte la propiedad *Images* con la lista de imágenes del paso anterior.

- 3 *TCoolBar*, también de la página *Win32*. Sobre este componente colocaremos el menú y la barra de herramientas.
- 4 *TActionMainMenuBar* y *TActionToolBar*, que se encuentran en *Additional*. Deje caer ambos componentes sobre la barra del paso anterior. Luego modifique la propiedad *ActionManager* de ambos para que apunte al *ActionManager1*.
- 5 Por último, haga doble clic sobre el *ActionManager1*.



El componente *TActionManager* es una novedad de Delphi 6; en parte, actúa de contenedor de componentes *TAction*, al igual que el más antiguo *TActionList*, pero se supone además que facilita la asociación de esas acciones a los controles visuales. En la práctica, al tratarse de un nuevo componente, tiene algunos *bugs*, carencias y excentricidades.

Una vez que estamos dentro del editor del componente *TActionManager*, vamos a la segunda página del diálogo y seleccionamos el comando *New Standard Action* del menú de contexto. Veremos una lista de acciones definidas por la VCL, y seleccionaremos *FileOpen*. Vaya al Inspector de Objetos y seleccione la propiedad *Dialog* de la acción. Se trata de una referencia a un componente interno de *FileOpen*; una característica nueva en Delphi 6. Expanda la referencia y configure la propiedad *Filter* del cuadro de diálogo, para que acepte ficheros con las extensiones *xml* y *cds*. Esta última corresponde a las siglas de *Client DataSet*, y es la que se utiliza por omisión para los ficheros en formato binario de MyBase.

Luego pulsaremos el comando alternativo *New Action* y añadiremos una acción en blanco, a la que llamaremos *FileSave*, asegurándonos de que su propiedad *Category* contenga el valor *File*, al igual que sucede con *FileOpen*. Si observa el contenido de la lista de imágenes *ImageList1* comprobará que al añadir *FileOpen* hicimos que Delphi añadiese a la lista el mapa de bits asociado a la acción. Sin embargo, para *FileSave*

tendremos que añadir manualmente una imagen apropiada, y luego asociarla a la acción, por medio de su propiedad *ImageIndex*.

Necesitamos traer un *TClientDataSet*, que pondremos donde menos estorbe. Luego, añadiremos un *TDataSource*, de la página *Data Access*, al igual que hicimos en el ejemplo anterior. Recuerde que hay que modificar su propiedad *DataSet* para que apunte al conjunto de datos que trajimos inicialmente. Finalmente, traiga de la página *Data Controls*, un componente *TDBGrid*. Asigne *alClient* en su propiedad *Align*, y *DataSource1* en *DataSource*.

Ahora nos ocuparemos de darle vida a las acciones. Para *FileSave*, que estamos definiendo a partir de cero, tenemos que interceptar sus eventos *OnUpdate*, para activar o desactivar la acción según sea o no aplicable, y *OnExecute*, para ejecutar la acción en sí. *FileOpen*, al ser una acción predefinida, ya se encarga de mostrar el cuadro de diálogo correspondiente, pero tenemos todavía que interceptar su evento *OnAccept*, que se dispara cuando el usuario acepta un nombre de fichero en el diálogo. El código necesario es el siguiente:

```
procedure TForm1.FileOpen1Accept(Sender: TObject);
begin
    ClientDataSet1.LoadFromFile(FileOpen1.Dialog.FileName);
end;

procedure TForm1.FileSave1Update(Sender: TObject);
begin
    TAction(Sender).Enabled :=
        ClientDataSet1.Modified or
        (ClientDataSet1.ChangeCount > 0);
end;

procedure TForm1.FileSave1Execute(Sender: TObject);
begin
    ClientDataSet1.SaveToFile(FileOpen1.Dialog.FileName);
end;
```

Deténgase un momento en la respuesta anterior al evento *OnUpdate*. Me he adelantado un poco para utilizar las propiedades *ChangeCount* y *Modified*. Por ahora, sólo necesita saber que *ChangeCount* se refiere a cambios confirmados, mientras que las modificaciones que cambian el valor de *Modified* pueden ser canceladas pulsando la tecla ESC sobre la rejilla unas cuantas veces.

Para terminar, solamente necesitamos crear el menú y los botones de la barra de herramientas:

- 1 Haga doble clic sobre *TActionManager* y seleccione la segunda página.
- 2 Arrastre las acciones desde la lista de la derecha y déjelas caer sobre la barra de herramientas.
- 3 Luego arrastre la categoría *File*, de la lista a la izquierda, y déjela caer sobre la barra de menú.



Y eso es todo; ya puede probar la aplicación. Podemos incluso llevar la analogía con los procesadores de textos hasta el final, y añadir una comprobación asociada a la salida de la aplicación, para permitir que el usuario guarde los cambios. Sólo hay que interceptar el evento *OnCloseQuery* del formulario:

```

procedure TForm1.FormCloseQuery(Sender: TObject;
    var CanClose: Boolean);
begin
    if ClientDataSet1.Modified or
    (ClientDataSet1.ChangeCount > 0) then
        case MessageDlg('¿Guardar cambios?', mtConfirmation,
            mbYesNoCancel, 0) of
            mrYes: FileSavel.Execute;
            mrCancel: CanClose := False;
        end;
end;

```

## Apertura y cierre

¿Qué tal si nos alejamos por un momento del *TClientDataSet* y, con más perspectiva, analizamos algunas características comunes de todos los conjuntos de datos? No es una tarea fácil, pues la clase base *TDataSet* tiene una interfaz muy “ancha”, abarrotada de propiedades, métodos y eventos. La mejor forma de abordar su estudio es dividir esos recursos y características en áreas más o menos delimitadas de funcionalidad.

Comenzaremos por los métodos y propiedades que permiten abrir o activar (son sinónimos) un conjunto de datos y luego cerrarlo, o desactivarlo. Parecerá una tonteería que nos detengamos en algo tan sencillo, pero estoy seguro de poder contarle algún detalle interesante. Tenemos, en primer lugar, la propiedad *Active*:

```

property TDataSet.Active: Boolean;

```

Fácil, ¿verdad? Asignamos *True* para abrir, y *False* para cerrar... ¿Qué pasa si intentamos activar un conjunto de datos que ya estaba activo? No pasa absolutamente nada. *Active*, del mismo modo que todas las propiedades decentes de Delphi, implementa su método de escritura siguiendo aproximadamente el siguiente patrón:

```

procedure TDataSet.SetActive(Value: Boolean);
begin
    if Value <> FActive then
        begin
            // ... hay que activar realmente el componente
        end;
end;

```

Existen dos métodos alternativos para realizar el trabajo de *Active*:

```

procedure TDataSet.Open;
procedure TDataSet.Close;

```

A pesar de lo que sugeriría el sentido común, al menos *mi* sentido común, *Open* y *Close* se implementan por medio de *Active*, y no al contrario. Eso significa que llamar a *Open* es varios microsegundos más lento que asignar *True* en *Active*. En compensación, la llamada a estos métodos ocupa menos espacio que la asignación a la propiedad.

Hay otra propiedad relacionada con *Active*, aunque en realidad contiene mucha más información. Se trata de la propiedad de tipo enumerativo *State*, disponible sólo en tiempo de ejecución y únicamente en modo de lectura. La relación consiste en que *State* vale *dsInactive* cuando el conjunto de datos está cerrado:

```
Active = False ⇔ State = dsInactive
```

Cuando presentemos los métodos de actualización, veremos todos los restantes valores que puede adoptar *State*. Pero ahora nos interesa otra cosa: normalmente, cuando *State* cambia su valor, el conjunto de datos dispara eventos antes y después del cambio. En particular, durante la apertura y cierre de un conjunto de datos, se disparan los siguientes eventos:

```
type
  TDataSetNotifyEvent = procedure (DataSet: TDataSet) of object;

property TDataSet.BeforeOpen: TDataSetNotifyEvent;
property TDataSet.AfterOpen: TDataSetNotifyEvent;
property TDataSet.BeforeClose: TDataSetNotifyEvent;
property TDataSet.AfterClose: TDataSetNotifyEvent;
```

### IMPORTANTE

Recuerde que estas propiedades, métodos y eventos son comunes a todas las clases de conjuntos de datos de Delphi.

## Lectura de propiedades durante la carga

El “truco” que voy a presentar tiene que ver más con la escritura de componentes que con problemas específicos de bases de datos. Pero se ocupa de un detalle muy importante de la inicialización de componentes, y es bueno que lo conozca.

En concreto, sabemos que la asignación de un valor lógico en la propiedad *Active* de un conjunto de datos de la clase que sea provoca la apertura o cierre del mismo, esto es, un efecto colateral muy notable. También sabemos que eso es posible gracias a que la propiedad *Active* utiliza un método privado para la asignación de valores:

```
type
  TDataSet = class(TComponent)
  // ...
private
  function GetActive: Boolean; virtual;
  procedure SetActive(Value: Boolean); virtual;
public
  property Active: Boolean read GetActive write SetActive;
```

end;

Si Delphi fuese Java, que gracias al cielo no lo es, la inicialización de componentes se haría siempre mediante instrucciones explícitas incluidas en el constructor o en algún método igual de especial. Pero Delphi utiliza datos binarios, obtenidos a través de ficheros *dfm* para crear los componentes de un formulario y asignar en sus propiedades valores iniciales. Sigue siendo posible la inicialización dinámica, a lo Java, pero la mayoría de las veces la inicialización sigue la partitura escrita en el *dfm*.

Por ejemplo, Delphi está iniciando la aplicación de nuestro último ejemplo y va a crear la ventana principal. Inicialmente, se crea un formulario vacío, llamando al constructor *CreateNew* desde el constructor *Create*. E inmediatamente se busca un recurso binario dentro del ejecutable, que contiene la información que configuramos en tiempo de diseño dentro del fichero *dfm*. Se abre el recurso como si se tratase de un flujo de datos secuencial, y se van leyendo sus partes integrantes. Por ejemplo, primero aparece información relacionada con un componente de la clase *TActionManager*. La aplicación crea un componente de esa clase y pasa a leer las propiedades que debe asignarle al nuevo objeto. Así va avanzando sobre el flujo de datos del recurso, y en algún momento tropezará con *ClientDataSet1*, el componente que añadimos en tiempo de diseño.

La aplicación crea entonces un conjunto de datos y lee y asigna las propiedades almacenadas. En nuestro ejemplo, *Active* valía *False* en tiempo de diseño. Ya sabemos que asignar *False* en una propiedad que contiene *False* no produce efecto alguno. ¿Y si el valor a asignar fuese *True*? No hay nada que temer: en ese caso habremos proporcionado algún sistema al conjunto de datos para que extraiga su contenido. Lo más probable en este ejemplo sería colocar un nombre de fichero en *FileName*. Así que cuando leamos que *ClientDataSet1.Active* debe valer *True* entonces...

... un momento, ¿qué valor se lee primero, *Active* o *FileName*? Como las propiedades se guardan en el *dfm* siguiendo el orden alfabético, *Active* es la primera propiedad con la que tropezaremos. Y está muy claro que en esas condiciones es imposible activar el conjunto de datos. Vayamos más lejos: el problema se nos presentará con mayor virulencia con otras clases de conjuntos de datos. Por ejemplo, para configurar un conjunto de datos en DB Express, es habitual utilizar un componente de la clase *TSQLConnection*, para centralizar el acceso a la base de datos. Es completamente aceptable que, en tiempo de diseño, dejemos caer primero sobre el formulario un conjunto de datos como *TSQLQuery*, y que después recordemos que también necesitamos el componente de conexión. En el fichero *dfm* se almacenarán los componentes en ese mismo orden. Cuando leamos el valor de *Active*, todavía no habremos creado el componente de conexión.

Los problemas de disponibilidad de componentes y propiedades son más comunes, y por lo tanto más graves, de lo que se piensa. Un desarrollador de componentes suele probar sus objetos creándolos dinámicamente, a la *javanesa*. Porque al estar desarrollando el componente, no puede por lo común traerlo desde la Paleta de Componen-

tes. Como consecuencia, los problemas de orden en la carga de propiedades no son debidamente señalados y corregidos.

La solución es sencilla. Delphi utiliza un algoritmo en varias fases para completar la carga de un formulario o módulo de datos. La principal razón para esto son las posibles referencias a objetos que aún no han sido cargados; no voy a explicar aquí cómo tiene lugar el proceso de resolución de referencias, principalmente porque podemos aceptar el algoritmo implementado por Delphi para casi todas las situaciones. Vamos a concentrarnos, por el contrario, en la forma en que realmente tiene lugar la apertura de conjuntos de datos durante la creación de formularios. He aquí, en primer lugar, lo que sucede cuando se le asigna un valor a la propiedad *Active* de un conjunto de datos general (note que la clase base es *TDataSet*):

```
procedure TDataSet.SetActive(Value: Boolean);
begin
    if csReading in ComponentState then
        FStreamedActive := Value
    else if Active <> Value then
        begin
            // ... el verdadero algoritmo de apertura ...
        end;
end;
```

El truco está en la propiedad *ComponentState* y en el estado *csReading*. El sistema de lectura de módulos de Delphi incluye ese valor dentro del estado del objeto mientras se efectúa la primera fase de carga. Si no estamos en esa fase, la asignación a *Active* dispara el verdadero algoritmo de apertura o cierre del conjunto de datos, sea éste el que sea. Pero si se está restaurando un conjunto de datos desde el contenido compilado de un fichero *dflm*, la asignación se realiza sobre una “falsa” variable auxiliar del objeto, *FStreamedActive* ... y no sucede nada más en esa fase.

Claro, queda pendiente la segunda fase de la carga. Cuando Delphi ha terminado de leer todos los componentes y los valores de sus propiedades, elimina *csReading* del estado del componente, y llama al método protegido virtual *Loaded*, que es común a todos los componentes. Este es el motivo por el cual se redefine *Loaded* para los conjuntos de datos, para hacer efectivo el valor almacenado en *FStreamedActive* durante esa segunda fase.

```
procedure TDataSet.Loaded;
begin
    inherited Loaded;
    try
        if FStreamedActive then Active := True;
    except
        if not (csDesigning in ComponentState) then raise;
        InternalHandleException
    end;
end;
```

¿Qué provecho podemos sacar de este conocimiento? Con la configuración de conjuntos de datos clientes que estamos viendo en este capítulo, poca cosa. Pero con

otros componentes y otros tipos y configuraciones de conjuntos de datos, podemos interceptar el método *Loaded* ... pero no del componente, sino del director de orquesta: el formulario.

```
procedure TForm1.Loaded;
begin
    inherited Loaded;
    // Todavía no se han activado realmente los conjuntos de datos,
    // ni las posibles conexiones a bases de datos SQL.
    // Aquí podemos todavía cambiar valores de propiedades,
    // según las condiciones encontradas en tiempo de ejecución
end;
```

A lo largo del libro, veremos varios ejemplos de aplicación de esta técnica.

## El buffer de datos y la posición del cursor

Me va a tener que perdonar, porque voy a ponerme “filosófico” para explicarle algunos detalles internos de diseño e implementación comunes a todos los tipos de conjuntos de datos. En primer lugar, aunque ya se habrá dado cuenta, la clase *TDataSet* sacrifica a veces la elegancia con tal de que podamos programar con rapidez. Este compromiso se nota en el siguiente hecho:

*“Un TDataSet representa, simultáneamente, a todo un conjunto de registros y a un registro particular de este conjunto”*

Quiero decir con esto que no existe una clase especial en la VCL para representar registros “sueltos”. Si en algún momento necesitamos guardar los datos de un registro para más adelante, tendremos dos alternativas:

- 1 Pasar laboriosamente el valor de cada campo a una estructura de datos definida por nosotros mismos.
- 2 Guardar, de alguna manera, la posición del registro, para retornar a él cuando sea necesario. En este mismo capítulo veremos que hay varias formas de lograrlo.

Por supuesto, el registro particular al que acabo de hacer referencia es el *registro activo* del conjunto de datos. Si usted lleva algún tiempo programando en Delphi esto le parecerá evidente: es el componente *TDataSet* el que determina la posición del cursor, es decir, cuál es el registro activo de todos los asociados al conjunto de datos. Lo menciono porque cuando comencé a investigar mi primera versión de Delphi tuve que hacer un par de experimentos por cuenta propia para saber si esa posición la mantenía el *TDataSet*, el *TDataSource* o algún otro componente.

Casi todas las clases derivadas de *TDataSet*, además, mantienen una copia del registro activo en un *buffer* gestionado por el propio componente, y al cual se accede mediante la función *ActiveBuffer*. En realidad, es posible que se mantengan varios *buffers* a la vez. No tenemos acceso a ellos desde fuera de la clase, pero *TDataSet* ofrece las siguientes propiedades dentro de su sección **protected**:

```
property TDataSet.BufferCount: Integer;  
property TDataSet.Buffers[Index: Integer]: PChar;
```

El propósito de estos *buffers* es acelerar el trabajo con los controles de datos que pueden mostrar más de un registro simultáneamente, como *TDBGrid* y *TDBCtrlGrid*. Lo curioso es que el número real de *buffers* dentro de un *TDataSet*, en un momento dado, depende de los controles de datos que estén conectados a él. Por ejemplo, si solamente utilizamos *TDBEdit*, hará falta un solo *buffer*. Si tenemos una rejilla que muestra veinte registros, y una rejilla de control con capacidad para siete, el conjunto de datos reservará veinte *buffers*. Así, cuando sea necesario redibujar uno de estos controles, no habrá necesidad alguna de volver a pedir datos a la interfaz nativa de acceso a datos.

La excepción a esta forma de trabajo son los conjuntos de datos de DB Express, que estudiaremos en próximos capítulos; en este caso, el registro activo es mantenido internamente por el código de DB Express. Por una parte, se hace imposible conectar una rejilla de datos a uno de estos componentes. La buena noticia, sin embargo, es que así se ahorra algo de tiempo, porque evitamos copiar el registro desde la interfaz nativa de acceso al *buffer*, para luego extraer desde ahí los valores de sus campos.

## Navegación

Estudiaremos ahora los métodos de navegación comunes a todos los conjuntos de datos. Parte importante de lo que debemos saber es cuán eficiente o no puede ser cada uno de ellos; naturalmente, por el momento nos limitaremos a realizar este análisis para los *TClientDataSet* que leen sus valores desde un fichero MyBase. Los métodos básicos de navegación son los siguientes:

```
procedure TDataSet.First;  
procedure TDataSet.Prior;  
procedure TDataSet.Next;  
procedure TDataSet.Last;  
procedure TDataSet.MoveBy(Filas: Integer);
```

No hay mucho que explicar: hay un método para ir al primer registro, al anterior, al siguiente y al último. Si aplicamos estos métodos a un conjunto de datos cliente que ha leído sus datos desde un fichero MyBase, encontraremos que la implementación de todos ellos es lógicamente muy eficiente. Pero no debemos confiarnos mucho cuando utilicemos *TClientDataSet* como caché de otro componente de acceso a datos. En algunos casos, una llamada inoportuna a *Last* podría causar la evaluación innecesaria de un gran número de registros.

*MoveBy* nos permite desplazarnos en un número relativo de registros respecto a la posición actual. ¿La posición actual? Se obtiene mediante la propiedad *RecNo*:

```
property TDataSet.RecNo: Integer;
```

Y aquí comienzan nuestras dificultades, pues no todos los tipos de conjuntos de datos implementan correctamente *RecNo*. Por ejemplo, los conjuntos de datos del BDE, cuando acceden a bases de datos SQL, siempre devuelven -1 como posición del cursor. En los conjuntos de datos cliente, para nuestra tranquilidad, siempre funciona correctamente *RecNo*. A propósito, en un alarde de excentricidad, *RecNo* considera que el primer registro es el número uno, no el cero. Casi todas las propiedades en Delphi toman siempre el cero como punto de partida.

No obstante, casi todos los derivados de *TDataSet* implementan correctamente el cálculo del número total de registros:

```
property TDataSet.RecordCount: Integer;
```

Antes hablábamos de *MoveBy*, ¿existe un *MoveTo*? No, pero podemos asignar valores en *RecNo* para mover la posición del cursor. Por ejemplo, para un conjunto de datos MyBase sería correcta y eficiente la siguiente instrucción:

```
ClientDataSet1.RecNo := ClientDataSet1.RecordCount;
```

No todos los tipos de conjuntos de datos, por supuesto, permiten asignar impunemente valores en *RecNo*.

## El Alfa y la Omega

Hay dos propiedades lógicas que nos permiten averiguar si estamos en el principio o en el final del conjunto de datos:

```
property TDataSet.Bof: Boolean;  
property TDataSet.Eof: Boolean;
```

### MAYUSCULAS, MINUSCULAS Y OTRAS TONTERIAS

Para mí sería más natural escribir *EOF* y *BOF* en mayúsculas, porque se trata de siglas, en definitiva. Sucede, sin embargo, que hay un entorno de desarrollo llamado C++ Builder, basado en un tonto lenguaje, inventado cuando la gente masticaba cactus y la gasolina era barata, y ese tonto lenguaje distingue entre mayúsculas y minúsculas como si le fuera la vida eterna en el empeño. Para más desgracia, en ese lenguaje se abusa de las macros, y hay dos macros llamadas precisamente *EOF* y *BOF*. Como este humilde autor utiliza C++ Builder de vez en cuando, porque a pesar del tonto lenguaje no es un mal entorno de programación, se ha acostumbrado a escribir los métodos y propiedades con las mayúsculas y minúsculas establecidas en su definición.

¡Tenga mucho cuidado al interpretar el significado de *Eof* y *Bof*! Pruebe la siguiente secuencia de instrucciones, asumiendo que *ClientDataSet1* es un conjunto de datos cliente activo, debidamente configurado y con más de una fila:

```
ClientDataSet1.First;  
Assert(ClientDataSet1.RecNo = 1);  
Assert(ClientDataSet1.Bof);
```

```
ClientDataSet1.Next;
Assert(ClientDataSet1.RecNo = 2);
ClientDataSet1.Prior;
Assert(ClientDataSet1.RecNo = 1);
Assert(ClientDataSet1.Bof);           // Esta aserción fallará
```

*Assert* es un procedimiento que verifica el valor de la expresión lógica que le pasamos como primer parámetro, y que espera que sea verdadera. En caso contrario, se dispara una excepción. Existe la posibilidad de usar un segundo parámetro para indicar el mensaje de error si se produce la excepción, pero no nos hará falta para este experimento.

Durante la prueba, la primera línea envía el cursor a la primera fila del conjunto de datos. Naturalmente, el registro activo será el *1*, de acuerdo al convenio délfico... y comprobará que *Bof* devuelve *True* para este caso. Luego nos movemos al segundo registro, y verificamos nuevamente el número del registro activo. Con la llamada a *Prior* regresamos a la primera fila. Nuevamente, *RecNo* vale *1*, ¡pero la última aserción disparará un error, porque *Bof* devolverá *False*!

La explicación es que *Bof* no nos dice simplemente que estamos en el primer registro, sino que hemos intentado retroceder más allá del inicio del conjunto de datos. Si hubiésemos añadido otra llamada a *Prior* en el experimento anterior, sí que se habría activado *Bof*; naturalmente, *RecNo* seguiría valiendo *1*. Por convenio, además, el método *First* también activa *Bof*. Y lo mismo se aplica a *Eof*, que devuelve *True* cuando la carabela llega al finisterre, mientras los marineros lanzan gemidos de espanto ante la visión del abismo de espuma.

El bucle típico de recorrido por las filas de un conjunto de datos se escribe entonces en una de las siguientes dos formas:

#### Hacia delante

```
ClientDataSet1.First;
while not ClientDataSet1.Eof do
begin
    // ... acción ...
    ClientDataSet1.Next;
end;
```

#### Hacia atrás

```
ClientDataSet1.Last;
while not ClientDataSet1.Bof do
begin
    // ... acción ...
    ClientDataSet1.Prior;
end;
```

Es interesante saber que podemos combinar los valores de *Bof* y *Eof* para saber si el conjunto de datos está vacío o no:

```
if ClientDataSet1.Bof and ClientDataSet1.Eof then
    // ... conjunto de datos vacío ...
```

En realidad, Delphi nos sugiere que usemos la siguiente función:

```
function TDataSet.IsEmpty: Boolean;
```



## Un viaje de ida y vuelta

Cuando cambiamos la fila activa de una tabla, es importante saber cómo regresar al lugar de origen. Delphi nos permite recordar una posición para volver más adelante a la misma mediante la técnica de *marcas de posición*. La VCL nos ofrece un tipo de datos, *TBookmark*, representado como un puntero, y tres métodos para usarlo:

```
function TDataSet.GetBookmark:TBookmark;
procedure TDataSet.GotoBookmark (B: TBookmark);
procedure TDataSet.FreeBookmark (B: TBookmark);
```

¿Para qué usar marcas de posición? ¿No sería más sencillo almacenar la posición del registro para luego movernos a ella? Para *TClientDataSet* es cierto lo anterior, pero no para el caso general. Internamente, la mayoría de las implementaciones de marcas de posición para conjuntos de datos SQL se quedan con los valores de la clave primaria del registro al que queremos regresar.

Mostraré a continuación el algoritmo típico de marcas de posición; tome nota de la instrucción **try/finally**, para garantizar el regreso y la destrucción de la marca:

```
var
    BM: TBookmark;
begin
    // Recordar la posición actual
    BM := ClientDataSet1.GetBookmark;
    try
        // Mover la fila activa ...
    finally
        // Regresar a la posición inicial
        ClientDataSet1.GotoBookmark(BM);
        // Liberar la memoria ocupada por la marca
        ClientDataSet1.FreeBookmark(BM);
    end;
end;
```

En realidad, *TBookmark* es un fósil de la primera versión de la VCL para 16 bits. Para simplificar el trabajo con las marcas de posición puede utilizarse el tipo de datos *TBookmarkStr*, y una nueva propiedad en los conjuntos de datos, *Bookmark*, del tipo anterior. El algoritmo anterior queda de la siguiente forma:

```
var
    BM: TBookmarkStr;
begin
    // Recordar la posición actual
    BM := ClientDataSet1.Bookmark;
    try
        // Mover la fila activa
    finally
        // Regresar a la posición inicial
        ClientDataSet1.Bookmark := BM;
    end;
end;
```

*TBookmarkStr* se implementa mediante una cadena de caracteres, a la cual se le aplica una conversión de tipos estática. Así se aprovecha la destrucción automática de las cadenas para liberar la memoria asociada a la marca, y evitamos el uso de *FreeBookmark*.

## Encapsulamiento de la iteración

Nos queda por resolver un pequeño asunto con la iteración: si el conjunto de datos que recorremos está asociado a controles visuales, cada vez que movemos la fila activa se redibujan todos los controles asociados. Observe la siguiente imagen:

N° pedido	N° cliente	Vendido	Enviado	Importe total	Impuestos	Gastos envío	Pagado
1.003	1.351	12/abr/1988	03/may/1988	1.250,00 €	4,00%	0,00 €	0,00 €
1.004	2.156	17/abr/1988	18/abr/1988	7.885,00 €	0,00%	0,00 €	7.885,00 €
1.005	1.356	20/abr/1988	21/ene/1988	4.807,00 €	0,00%	0,00 €	4.807,00 €
1.006	1.380	06/nov/1994	07/nov/1988	31.987,00 €	0,00%	0,00 €	0,00 €
1.007	1.384	01/may/1988	02/may/1988	6.500,00 €	0,00%	0,00 €	6.500,00 €
1.008	1.510	03/may/1988	04/may/1988	1.449,00 €	0,00%	0,00 €	0,00 €
1.009	1.513	11/may/1988	12/may/1988	5.587,00 €	0,00%	0,00 €	0,00 €
1.010	1.551	11/may/1988	12/may/1988	4.996,00 €	0,00%	0,00 €	4.996,00 €
1.011	1.560	18/may/1988	19/may/1988	2.679,00 €	0,00%	0,00 €	2.679,00 €
1.012	1.563	19/may/1988	20/may/1988	5.201,00 €	0,00%	0,00 €	5.201,00 €

Total: 2.922.607,00 €      Pagado: 2.633.409,00 €

El formulario pertenece a la aplicación *Iteracion*, incluida en el CD. Sobre la rejilla se muestra el contenido de *orders.xml*, una tabla con pedidos. He puesto un botón *Sumar*, para calcular el importe total de todos los pedidos. En el capítulo 20 explicaré una técnica más eficiente de calcular totales y mantenerlos actualizados, pero con lo que sabemos hasta ahora, tendremos que recorrer el conjunto de datos y sumar el importe de cada fila. Hágalo usted mismo, y comprobará que la rejilla parpadea frenéticamente mientras se ejecuta la iteración.

La solución, no obstante, es muy simple. Podemos llamar al método *DisableControls* del conjunto de datos. Al hacerlo, desactivamos el envío de notificaciones a los controles visuales asociados. Naturalmente, al terminar el algoritmo tenemos que volver a activarlos, para lo que contamos con el método *EnableControls*. Podemos combinar en un mismo algoritmo todo lo que hemos aprendido sobre marcas de posición, anulación de notificaciones... y por qué no, mostrar durante el recorrido ese cursor tan bonito del reloj de arena, ese que adoran los usuarios de las aplicaciones escritas en Java:

```
var
  BM: TBookmarkStr;
begin
  BM := ClientDataSet1.Bookmark;
  ClientDataSet1.DisableControls;
```

```

Screen.Cursor := crHourglass;
try
    // ... aquí vendría la "carne" ...
    // ... el resto son los huesos del "esqueleto" ...
finally
    ClientDataSet1.Bookmark;
    ClientDataSet1.EnableControls;
    Screen.Cursor := crDefault;
end;
end;

```

No sé lo que pensará, pero para mí es demasiado código. Si tengo que repetirlo varias veces, prefiero definir una clase auxiliar como la siguiente:

```

type
    TIterador = class(TInterfacedObject, IInterface)
    protected
        FDataSet: TDataSet;
        FBookmark: TBookmarkStr;
    public
        constructor Create(ADataset: TDataSet);
        destructor Destroy; override;
    end;

```

Observe que *TIterador* implementa una interfaz, aunque los métodos básicos que ésta contiene se van a heredar de la clase base *TInterfacedObject*. Nos interesa ahora el constructor y el destructor:

```

constructor TIterador.Create(ADataset: TDataSet);
begin
    inherited Create;
    FDataSet := ADataset;
    FBookmark := ADataset.Bookmark;
    FDataSet.DisableControls;
    Screen.Cursor := crHourglass;
end;

destructor TIterador.Destroy;
begin
    FDataSet.Bookmark := FBookmark;
    FDataSet.EnableControls;
    Screen.Cursor := crDefault;
    inherited Destroy;
end;

```

Para programar el recorrido que suma los importes en la aplicación mencionada al inicio de la sección, he utilizado la clase *TIterador* de esta forma:

```

procedure TwndMain.bnSumarClick(Sender: TObject);
var
    I: IInterface;
    Total: Currency;
begin
    I := TIterador.Create(Pedidos); // Aquí está el "truco"
    Total := 0;
    Pedidos.First;
    while not Pedidos.Eof do
        begin

```

```

        Total := Total + PedidosItemsTotal.Value;
        Pedidos.Next;
    end;
    StatusBar.SimpleText := 'Total: ' + FormatCurr('0,.00 €', Total);
end;

```

Observe que el “truco” no reside en los métodos que define la interfaz, porque usamos el plebeyo tipo *IInterface*, sino en que aprovechamos el contador de referencias que Delphi mantiene automáticamente para las variables de ese tipo. La inicialización del algoritmo es ejecutada dentro del constructor de *TIterador*. Cuando termina el procedimiento, la variable local *I* desaparece, y se llama al destructor del objeto asociado, porque *I* era quien único se preocupaba por este pobre objeto abandonado. Entonces se ejecutan las instrucciones finales de la iteración. Tome nota de que la finalización tiene lugar incluso si se producen excepciones durante el recorrido.

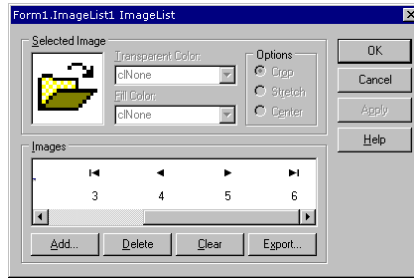
### ADVERTENCIA

Cuando estudiemos las relaciones maestro/detalles haré algunas precisiones importantes sobre el uso de *DisableControls* y *EnableControls*.

## Acciones de navegación

Para practicar el uso de los métodos y propiedades que acabamos de presentar vamos a integrarlos dentro de varios componentes de acción. Es una técnica que vamos a utilizar a lo largo del resto del libro, cada vez que presentemos un grupo de recursos de programación relacionados, porque nos permitirá no sólo saber *qué* podemos hacer (evento *OnExecute*) sino además aclarar *cuándo* podemos hacerlo (evento *OnUpdate*).

Retomaremos el “ejemplo con acciones” de este capítulo, en el que utilizamos un *TActionManager*, para aprovechar los componentes ya configurados. Antes quiero advertirle que *ya* existen acciones predefinidas de navegación, aunque si las utilizamos ahora nos perderemos la mitad de la diversión. Eso no quiere decir que seamos masoquistas: vamos al menos a aprovechar las imágenes que están asociadas a esas acciones predefinidas. En el editor de *TActionManager*, ejecute el comando *New Standard Action*, y traiga las cuatro acciones *DataSetFirst*, *DataSetPrior*, *DataSetNext* y *DataSetLast*. Luego bórrelas. Sí, ha leído bien. Para comprender el propósito de este aparente sin sentido, haga doble clic sobre la lista de imágenes, y compruebe que los cuatro iconos se han añadido al componente y están ahora a nuestra disposición.



Ahora sí: añade cuatro acciones vacías (*New Action*, a secas), llámelas igual que las anteriores y asóciesles sus correspondientes imágenes, en sus propiedades *ImageIndex*. La programación de los eventos *OnExecute* es trivial:

```

procedure TForm1.DatasetFirstExecute(Sender: TObject);
begin
    ClientDataSet1.First;
end;

procedure TForm1.DatasetPriorExecute(Sender: TObject);
begin
    ClientDataSet1.Prior;
end;

procedure TForm1.DatasetNextExecute(Sender: TObject);
begin
    ClientDataSet1.Next;
end;

procedure TForm1.DatasetLastExecute(Sender: TObject);
begin
    ClientDataSet1.Last;
end;

```

Más interesantes, en este caso, son los eventos *OnUpdate*. Resulta que las condiciones necesarias para que *First* y *Prior* tengan sentido son las mismas, y lo mismo sucede con *Next* y *Last*. Por lo tanto, crearemos manejadores de eventos compartidos de dos en dos. Seleccionamos a la vez, por ejemplo, los componentes *DataSetFirst* y *DataSetPrior*, con la ayuda del ratón y la tecla de mayúsculas. Vamos al Inspector de Objetos, seleccionamos la página de eventos y, sobre la línea del evento *OnUpdate*, tecleamos el identificador *FirstPriorUpdate*; podríamos limitarnos a un doble clic, pero el nombre que generaría Delphi no nos indicaría que se trata de un método compartido:

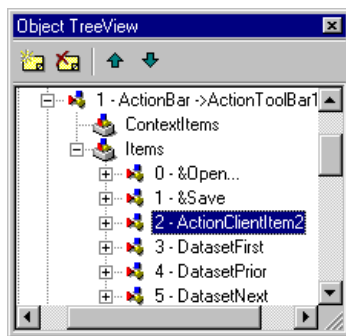
```

procedure TForm1.FirstPriorUpdate(Sender: TObject);
begin
    TAction(Sender).Enabled := ClientDataSet1.Active and
        not ClientDataSet1.Bof;
end;

procedure TForm1.NextLastUpdate(Sender: TObject);
begin
    TAction(Sender).Enabled := ClientDataSet1.Active and
        not ClientDataSet1.Eof;
end;

```

Nos queda añadir las cuatro acciones a la barra de herramientas... y entonces tropezamos con un problema inesperado. ¿Cómo demonios se crea una barra de separación vertical con estos nuevos componentes? La respuesta está muy escondida en la ayuda en línea, pero es bastante coherente con el funcionamiento de otros componentes similares. Tiene que seleccionar la barra de herramientas, y fijarse en el árbol de objetos que está sobre el Inspector de Objetos:

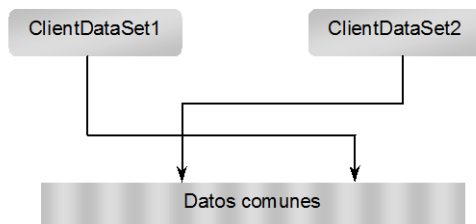


Tendrá que añadir un nuevo ítem en ese árbol, como muestra la imagen anterior. Y asignar un guión '-' en su propiedad *Caption*.

Recuerde que nuestro propósito al crear estas acciones es docente. Las acciones predefinidas de navegación tienen una ventaja: pueden configurarse para que actúen sobre la rejilla activa del formulario activo, en vez de limitarse, como las nuestras, a navegar sobre un conjunto de datos fijo. Esta posibilidad nos permitiría, por ejemplo, colocar esas acciones en la barra de herramientas de una ventana MDI principal, de modo que pueda utilizarse con cualquier ventana hija que contenga un conjunto de datos navegable.

## El ataque de los clones

Los conjuntos de datos cliente de MyBase, y los de ADO Express, tienen una potente característica: es posible clonarlos. Se puede crear una copia de uno de ellos que comparta los mismos datos, pero que mantenga su propia posición sobre ellos. Ahí es donde está la gracia, en poder mantener dos cursores independientes sobre el mismo conjunto de registros.



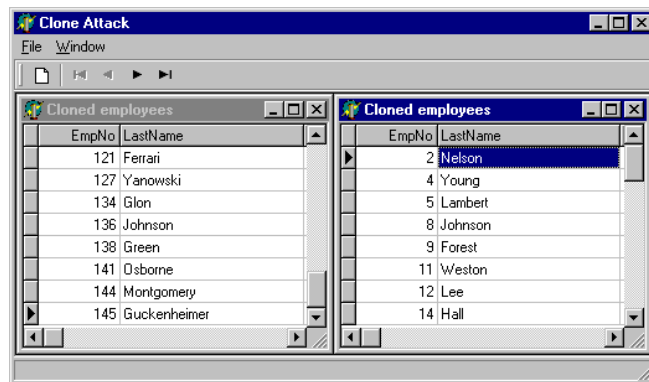
El método necesario es el siguiente:

```
procedure TCustomClientDataSet.CloneCursor(
    Source: TCustomClientDataSet; Reset, KeepSettings: Boolean);
```

*CloneCursor* se aplica sobre el conjunto de datos que se va a convertir en un clon, y que debe existir con anterioridad. Si desea que el clon se cree dinámicamente, puede ayudarse con una función auxiliar como la siguiente:

```
function NuevoClon(
    Origen: TCustomClientDataSet): TCustomClientDataSet;
type
    TCDSClass = class of TCustomClientDataSet;
begin
    Result := TCDSClass(Origen.ClassType).Create(Origen.Owner);
    try
        Result.CloneCursor(Origen, False, False);
    except
        Result.Free;
        raise;
    end;
end;
```

Los parámetros *Reset* y *KeepSettings* determina con lo que pasará con las propiedades *Filter*, *IndexName*, *MasterSource*, etcétera... del clon. Si *Reset* es verdadero, reciben sus valores por omisión; si *KeepSettings* es verdadero, es porque *Reset* es *False*, y queremos mantener los valores originales de la copia. Si usamos la función anterior para crear el clon desde cero, pasamos *False* en ambos parámetros para copiar también el valor de las propiedades mencionadas.



Para demostrar la clonación de conjuntos de datos, he incluido en el CD una aplicación MDI llamada *Clones*. La ventana principal contiene un *TClientDataSet*, de nombre *Empleados*, en el que he cargado el fichero *employee.xml*; este componente se encuentra activo desde el tiempo de diseño. Hay, además, una barra de herramientas, con botones genéricos de navegación, y un botón adicional asociado a una acción que he llamado *FileNewClone*. Esta es su respuesta al evento *OnExecute*:

```

procedure TwndMain.FileNewCloneExecute(Sender: TObject);
begin
    TwndClone.Create(Self);
end;

```

Por su parte, la clase *TwndClone* corresponde a un formulario hijo MDI, con otro *TClientDataSet*, esta vez sin configurar. Además, he puesto un *TDataSource* y una rejilla de datos, para mostrar el contenido del conjunto de datos cliente. El componente obtiene sus datos durante la creación del formulario hijo:

```

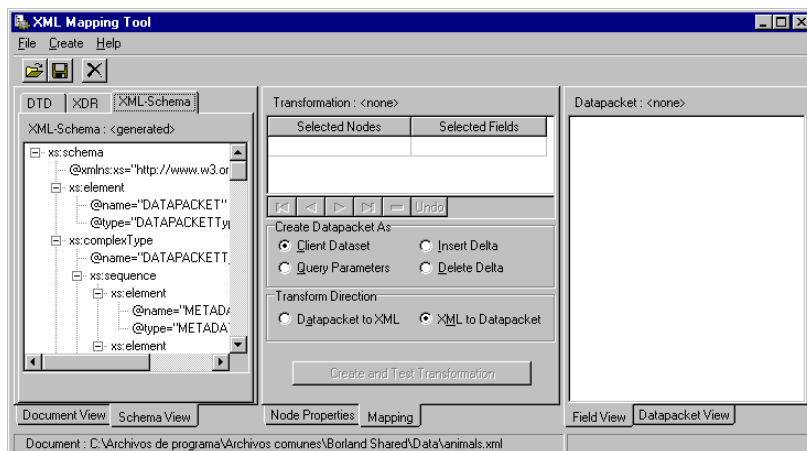
procedure TwndClone.FormCreate(Sender: TObject);
begin
    ClientDataSet1.CloneCursor(wndMain.Empleados, False, False);
end;

```

Ejecute la aplicación, y cree al menos un par de ventanas hijas. Verifique que el cursor de cada una de ellas es independiente de los cursores de las otras. Pero compruebe lo que sucederá cuando realice una actualización en cualquiera de las instancias: el cambio se propagará automáticamente por todas las ventanas.

## Transformaciones XML

Antes de terminar el capítulo, quiero mencionar un conjunto de técnicas que, al menos teóricamente, son muy interesantes. Se trata de técnicas basadas en la transformación de textos XML. La siguiente imagen corresponde a la aplicación *XML Mapper*, que se puede ejecutar desde el menú *Tools* de Delphi:



Con la ayuda de esta herramienta, se pueden crear ficheros que definan una transformación entre dos formatos diferentes de XML. El fichero resultante, de extensión *xtr*, puede ser utilizado directamente por el componente *TXMLTransform*, para convertir ficheros o flujos de datos (*streams*) XML en una sola dirección. Para obtener la transformación inversa necesitaríamos definirla explícitamente.



Más interesante es el componente *TXMLTransformProvider*. Con su ayuda podemos lograr que *TClientDataSet* pueda leer y escribir información en un fichero XML de formato arbitrario... siempre que existan las transformaciones correspondientes entre el formato de paquetes de datos de MyBase y el del fichero de marcos. El conjunto de datos clientes haría referencia al *TXMLTransformProvider* a través de su propiedad *ProviderName*. En el capítulo 31 estudiaremos cómo asociar un *TClientDataSet* a un proveedor, en general, para recuperar información con su ayuda.

Finalmente, tenemos también un *TXMLTransformClient*, que actúa como cliente de un proveedor arbitrario. Necesita también varios ficheros de transformación y se puede utilizar, por ejemplo, para convertir a formato XML registros provenientes de una fuente de datos relacionales arbitraria.



## Acceso a campos

**L**OS COMPONENTES DE ACCESO A CAMPOS son parte fundamental de la estructura de la VCL. Estos objetos permiten manipular los valores de los campos, definir formatos de visualización y edición, y realizar ciertas validaciones básicas. Sin ellos, nuestros programas tendrían que trabajar directamente con la imagen física del *buffer* del registro activo en un conjunto de datos. Afortunadamente, Delphi crea campos aún cuando se nos olvida hacerlo. En este capítulo estudiaremos las clases de campos y sus propiedades. Incluiremos también el tratamiento de los campos *blob*, que permiten el trabajo con imágenes o textos grandes, pero dejaremos para más adelante algunos tipos de campos específicos de ciertos sistemas, como los relacionados con las extensiones orientadas a objeto de Oracle.

### Creación de componentes de campos

Por mucho que busquemos, nunca encontraremos los componentes de acceso a campos en la Paleta de Componentes. El quid está en que estos componentes se vinculan al conjunto de datos (tabla o consulta) al cual pertenecen, del mismo modo en que los ítems de menú se vinculan al objeto de tipo *TMainMenu* ó *TPopupMenu* que los contiene. Siguiendo la analogía con los menús, para crear componentes de campos necesitamos realizar una doble pulsación sobre una tabla para invocar al *Editor de Campos* de Delphi. Este Editor se encuentra también disponible en el menú local de las tablas como el comando *Fields editor*.

Antes de explicar el proceso de creación de campos necesitamos aclarar una situación: podemos colocar una conjunto de datos en un formulario, asociarle una fuente de datos y una rejilla, y echar a andar la aplicación resultante. ¿Para qué queremos campos entonces? Aún cuando no se han definido componentes de acceso a campos explícitamente para una tabla, estos objetos están ahí, pues han sido creados automáticamente por Delphi. Si durante la apertura de una tabla se detecta que el usuario no ha definido campos en tiempo de diseño, la VCL crea objetos de acceso de forma implícita. Por supuesto, estos objetos reciben valores por omisión para sus propiedades, que quizás no sean los que deseamos.

Precisamente por eso creamos componentes de campos en tiempo de diseño: para poder controlar las propiedades y eventos relacionados con los mismos. La creación

en tiempo de diseño no nos hace malgastar memoria adicional en tiempo de ejecución, pues los componentes se van a crear de una forma u otra. Pero sí tenemos que contar con el aumento de tamaño del fichero *dflm*, que es donde se va a grabar la configuración persistente de los valores iniciales de las propiedades de los campos. Este es un factor a tener en cuenta, como veremos más adelante.



El Editor de Campos es una ventana de ejecución no modal; esto quiere decir que podemos tener a la vez en pantalla distintos conjuntos de campos, correspondiendo a distintas tablas, y que podemos pasar sin dificultad de un Editor a cualquier otra ventana, en particular, al Inspector de Objetos. Para realizar casi cualquier acción en el Editor de Campos hay que pulsar el botón derecho del ratón y seleccionar el comando de menú adecuado. Tenemos además una pequeña barra de navegación en la parte superior del Editor. Esta barra no está relacionada en absoluto con la edición de campos, sino que es un medio conveniente de mover, en tiempo de diseño, el cursor o fila activa de la tabla asociada.

<u>A</u> dd fields...	Ctrl+A
<u>N</u> ew field...	Ctrl+N
A <u>dd</u> all fields	Ctrl+F
<u>C</u> ut	Ctrl+X
<u>C</u> opy	Ctrl+C
<u>P</u> aste	Ctrl+V
<u>D</u> elete	Del
S <u>e</u> lect all	Ctrl+L

Añadir componentes de campos es muy fácil, pues basta con ejecutar el comando *Add fields* del menú local. Se presenta entonces un cuadro de diálogo con una lista de los campos físicos existentes en la tabla y que todavía no tienen componentes asociados. Esta lista es de selección múltiple, y selecciona por omisión todos los campos. Es aconsejable crear componentes para todos los campos, aún cuando no tengamos en mente utilizar algunos campos por el momento. La explicación tiene que ver también con el

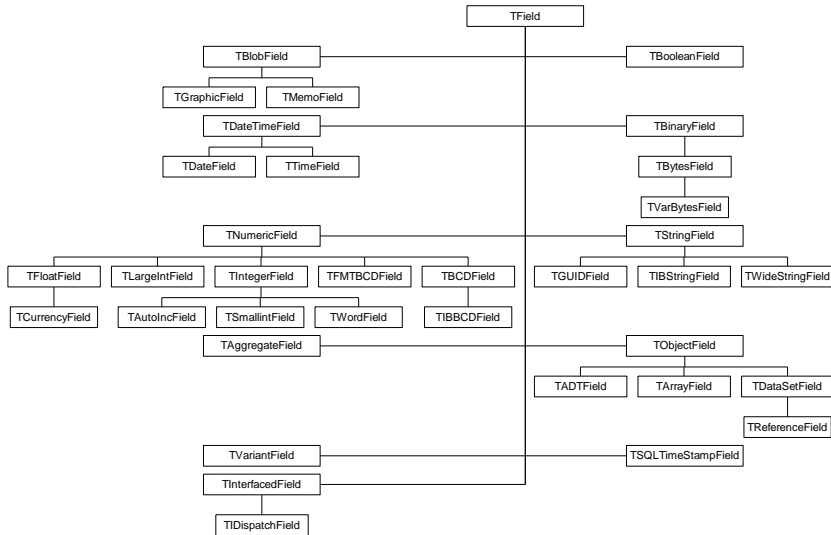
proceso mediante el cual la VCL crea los campos. Si al abrir la tabla se detecta la presencia de al menos un componente de campo definido en tiempo de diseño, Delphi no intenta crear objetos de campo automáticamente. El resultado es que los campos que no creemos durante el diseño *no existirán*, en lo que a Delphi concierne.

Esta operación puede repetirse más adelante, si añadimos nuevos campos durante una reestructuración de la tabla, o si modificamos la definición de un campo. En este último caso, es necesario destruir primeramente el viejo componente antes de añadir el nuevo. Para destruir un componente de campo, sólo es necesario seleccionarlo en el Editor de Campos y pulsar la tecla SUPR.

## Clases de campos

Una vez creados los componentes de campo, podemos seleccionarlos en el Inspector de Objetos a través de la lista de objetos, o mediante el propio Editor de Campos. Lo primero que llama la atención es que, a diferencia de los menús donde todos los comandos pertenecen a la misma clase, *TMenuItem*, aquí cada componente de acceso a campo puede pertenecer a una clase distinta. En realidad, todos los componentes de

acceso a campos pertenecen a una jerarquía de clases derivada por herencia de una clase común, la clase *TField*. El siguiente diagrama muestra la mencionada jerarquía:



De todos estos tipos, *TField*, *TNumericField* y *TBinaryField* nunca se utilizan directamente para crear instancias de objetos de campos; su papel es servir de ancestro a campos de tipo especializado.

La clase *TAggregateField* permite definir campos agregados con cálculo automático en conjuntos de datos clientes: sumas, medias, máximos, mínimos, etc. Tendremos que esperar un poco para examinarlos. En cuanto a la jerarquía que parte de la clase abstracta *TObjectField*, sirve para representar los nuevos tipos de datos orientados a objetos de Oracle 8: objetos incrustados (*TADTField*), referencias a objetos (*TReferenceField*), vectores (*TArrayField*) y campos de tablas anidadas (*TDataSetField*).

Aunque InterBase permite definir campos que contienen matrices de valores, las versiones actuales de la VCL y del BDE no permiten tratarlos como tales directamente.

## Nombre del campo y etiqueta de visualización

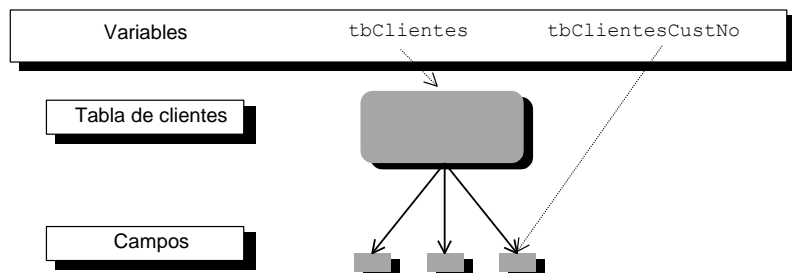
Existen tres propiedades de los campos que muchas veces son confundidas entre sí por el programador. Son las propiedades *Name*, *FieldName* y *DisplayLabel*. La primera es, como sucede con casi todos los componentes, el nombre de la variable de campo, o sea, del puntero al objeto. *FieldName* es el nombre de la columna de la tabla a la que se refiere el objeto de campo. Y *DisplayLabel* es un texto descriptivo del campo, que se utiliza, entre otras cosas, como encabezamiento de columna cuando el campo se muestra en una rejilla de datos.

De estas propiedades, *FieldName* es la que menos posibilidades nos deja: contiene el nombre de la columna, y punto. Por el contrario, *Name* se deduce inicialmente a partir del nombre de la tabla y del nombre de la columna. Si el nombre de tabla es *tbClientes* y el nombre del campo (*FieldName*) es *CustNo*, el nombre que Delphi le asigna a *Name*, y por consiguiente a la variable que apunta al campo, es *tbClientesCustNo*, la concatenación de ambos nombres.

Esto propicia un error bastante común entre los programadores, pues muchas veces escribimos por inercia, pensando en un esquema *tabla.campo*:

```
tbClientes.CustNo      // ¡¡¡INCORRECTO!!!
```

El siguiente gráfico puede ayudar a comprender mejor la relación entre los nombres de variables y los componentes de tablas y de campos:



La asignación automática de nombres de componentes de campos nos plantea un problema práctico: el tamaño del fichero *dflm* crece desmesuradamente. Tomemos por ejemplo una aplicación pequeña que trabaje con diez tablas, y supongamos que cada tabla tiene diez campos; éstas son estimaciones a la baja. Entonces, tendremos cien componentes de campos, y cada componente tendrá un nombre kilométrico que estará ocupando espacio en el fichero *dflm* y luego en la memoria, en tiempo de ejecución. Es por eso que buscando un menor tiempo de carga de la aplicación, pues la memoria no es una consideración primordial en estos días, tengo la costumbre de renombrar los componentes de campos con el propósito de disminuir la longitud de los nombres en lo posible, sin caer en ambigüedades. Por ejemplo, el nombre de componente *tbClientesCustNo* puede abreviarse a algo así como *tbClCustNo*; ya sé que son sólo seis letras menos, pero multiplíquelas por cien y verá.

## Acceso a los campos por medio de la tabla

Aunque la forma más directa, segura y eficiente de acceder a un campo es crear el componente en tiempo de diseño y hacer uso de la variable asociada, es también posible llegar indirectamente al campo a través de la tabla a la cual pertenece. Estas son las funciones y propiedades necesarias:

```
function TDataSet.FieldByName(const Nombre: string): TField;
property TDataSet.Fields: TFields;
```

La clase *TFields*, por su parte, es una colección que implementa, entre otras, las siguientes propiedades:

```
property TFields.Count;
property TFields.Fields[Index: Integer]: TField; default;
```

Con *FieldByName* podemos obtener el componente de campo dado su nombre, mientras que con *Fields* lo obtenemos si conocemos su posición. Está claro que esta última propiedad debe utilizarse con cautela, pues si la tabla se reestructura cambian las posiciones de las columnas. Mediante *FieldByName* y *Fields* obtenemos un objeto de tipo *TField*, la clase base de la jerarquía de campos. Por lo tanto, no se pueden utilizar directamente las propiedades específicas de los tipos de campos más concretos sin realizar una conversión de tipo.

Si pasamos un nombre inexistente de campo a la función *FieldByName*, se producirá una excepción, por lo cual no debemos utilizar esta función si lo que queremos es saber si el campo existe o no. Para esto último contamos con la función *FindField*, que devuelve el puntero al objeto si éste existe, o el puntero vacío si no:

```
function TDataSet.FindField(const Nombre: string): TField;
```

Recuerde que el componente de campo puede haber sido creado explícitamente por usted en tiempo de diseño, pero que si no ha realizado esta acción, la VCL construirá automáticamente estos objetos al abrir el conjunto de datos.

## Extrayendo información de los campos

Un componente de campo contiene los datos correspondientes al valor almacenado en la columna asociada de la fila activa de la tabla, y la operación más frecuente con un campo es extraer o modificar este valor. La forma más segura y eficiente es, una vez creados los campos persistentes con la ayuda del Editor de Campos, utilizar las variables generadas y la propiedad *Value* de las mismas. Esta propiedad se define del tipo apropiado para cada clase concreta de campo. Si el campo es de tipo *TStringField*, su propiedad *Value* es de tipo **string**; si el campo es de tipo *TBooleanField*, el tipo de *Value* es *Boolean*.

```
ShowMessage(Format('%f-%s',
    [ClientesCodigo.Value, // Un valor flotante
    ClientesNombre.Value])); // Una cadena de caracteres
```

Si la referencia pertenece al tipo genérico *TField*, como las que se obtienen con la propiedad *Fields* y la función *FieldByName*, es necesario utilizar propiedades con nombres como *AsString*, *AsInteger*, *AsFloat*, etc., que aclaran el tipo de datos que queremos recuperar.

```
ShowMessage(
  IntToStr(Clientes.FieldByName('Codigo').AsInteger)
  + '-' + Clientes.FieldByName('Nombre').AsString);
```

Las propiedades mencionadas intentan siempre hacer la conversión del valor almacenado realmente al tipo especificado; cuando no es posible, se produce una excepción. En el ejemplo anterior he usado *AsInteger* para convertir el valor del código de cliente en un entero, pero hubiéramos podido utilizar la propiedad *AsString* del campo.

Ahora bien, existe un camino alternativo para manipular los datos de un campo: la propiedad *FieldValues* de *TDataSet*. La declaración de esta propiedad es la siguiente:

```
property TDataSet.FieldValues[FieldName: string]: Variant; default;
```

Al estar marcada como la propiedad vectorial por omisión de *TDataSet*, no hace falta mencionarla explícitamente para usarla, sino que basta colocar los corchetes a continuación de la variable que apunta al conjunto de datos. Además, como la propiedad devuelve valores variantes, no es necesario preocuparse demasiado por el tipo del campo, pues la conversión transcurre automáticamente:

```
ShowMessage(Clientes['Codigo'] + '-' + Clientes['Nombre']);
```

También puede utilizarse *FieldValues* con una lista de nombres de campos separados por puntos y comas. En este caso se devuelve una matriz variante formada por los valores de los campos individuales:

```
var
  V: Variant;
begin
  V := Clientes['Codigo;Nombre'];
  ShowMessage(V[0] + '-' + V[1]);
end;
```

## ADVERTENCIA

A pesar de lo atractivo que pueda parecer el uso de *FieldValues*, debemos utilizar esta propiedad lo menos posible, pues tiene dos puntos débiles: puede que nos equivoquemos al teclear el nombre del campo (no se detecta el error sino en ejecución), y puede que nos equivoquemos en el tipo de retorno esperado.

Intencionalmente, todos los ejemplos que he mostrado leen valores desde las componentes de campos, pero no modifican este valor. El problema es que las asignaciones a campos sólo pueden efectuarse estando la tabla en alguno de los estados especiales de edición; en caso contrario, provocaremos una excepción. En el capítulo 22 trataremos estos temas con mayor detalle; un poco más adelante, en este mismo capítulo, veremos cómo se pueden asignar valores a campos calculados.

Es útil saber también cuándo es nulo o no el valor almacenado en un campo. Para esto se utiliza la función *IsNull*, que retorna un valor de tipo *Boolean*.



Por último, si el campo es de tipo memo, gráfico o BLOB, no existe una propiedad simple que nos proporcione acceso al contenido del mismo. Más adelante explicaremos cómo extraer información de los campos de estas clases.

## Las máscaras de formato y edición

El formato en el cual se visualiza el contenido de un campo puede cambiarse, para ciertos tipos de campos, por medio de una propiedad llamada *DisplayFormat*. Esta propiedad es aplicable a campos de tipo numérico, flotante y de fecha y hora; los campos de cadenas de caracteres no tienen una propiedad tal, aunque veremos en la próxima sección una forma de superar este “inconveniente”.

Si el campo es numérico o flotante, los caracteres de formato son los mismos que los utilizados por la función predefinida *FormatFloat*:

Carácter	Significado
0	Dígito obligatorio
#	Dígitos opcionales
.	Separador decimal
,	Separador de millares
;	Separador de secciones

La peculiaridad principal de estas cadenas de formato es que pueden estar divididas hasta en tres secciones: una para los valores positivos, la siguiente para los negativos y la tercera sección para el cero. Por ejemplo, si *DisplayFormat* contiene la cadena "\$#,;(#.00);Cero", la siguiente tabla muestra la forma en que se visualizará el contenido del campo:

Valor del campo	Cadena visualizada
12345	\$12.345
-12345	(12345.00)
0	Cero

Observe que la coma, el separador de millares americano, se traduce en el separador de millares nacional, y que lo mismo sucede con el punto. Otra propiedad relacionada con el formato, y que puede utilizarse cuando no se ha configurado *DisplayFormat*, es *Precision*, que establece el número de decimales que se visualizan por omisión. Tenga bien en cuenta que esta propiedad no limita el número de decimales que podemos teclear para el campo, ni afecta al valor almacenado finalmente en el mismo.

Cuando el campo es de tipo fecha, hora o fecha y hora, el significado de las cadenas de *DisplayFormat* coincide con el del parámetro de formato de la función *FormatDateTime*. He aquí unos pocos aunque no exhaustivos ejemplos de posibles valores de esta propiedad:

**Valor de *DisplayFormat* Ejemplo de resultado**

<i>dd-mm-yy</i>	04-07-64
<i>dddd, d "de" mmmm "de" yyyy</i>	sábado, 26 de enero de 1974
<i>hh:mm</i>	14:05
<i>h:mm am/pm</i>	2:05 pm

---

La preposición “de” se ha tenido que encerrar entre dobles comillas, pues en caso contrario la rutina de conversión interpreta la primera letra como una indicación para poner el día de la fecha.

Si el campo es de tipo lógico, de clase *TBooleanField*, la propiedad *DisplayValues* controla su formato de visualización. Esta propiedad, de tipo **string**, debe contener un par de palabras o frases separadas por un punto y coma; la primera frase corresponde al valor verdadero y la segunda al valor falso:

```
tbDiarioBuenTiempo.DisplayValues :=
  'Un tiempo maravilloso;Un día horrible';
```

Por último, la edición de los campos de tipo cadena, fecha y hora puede controlarse mediante la propiedad *EditMask*. Los campos de tipo numérico y flotante no permiten esta posibilidad. Además, la propiedad *EditFormat* que introducen estos campos no sirve para este propósito, pues indica el formato inicial que se le da al valor numérico cuando comienza su edición. Por ejemplo, usted desea que el campo *Precio* se muestre en dólares, y utiliza la siguiente máscara de visualización en *DisplayFormat*:

```
$#0,.00
```

Entonces, para eliminar los caracteres superfluos de la edición, como el signo de dólar y los separadores de millares, debe asignar *#0.00* a la propiedad *EditFormat* del campo.

## Los eventos de formato de campos

Cuando no bastan las máscaras de formato y edición, podemos echar mano de dos eventos pertenecientes a la clase *TField*: *OnGetText* y *OnSetText*. El evento *OnGetText*, por ejemplo, se intenta ejecutar cada vez que Delphi necesita una representación visual del contenido de un campo. Esto sucede en dos circunstancias diferentes: cuando se está visualizando el campo de forma normal, y cuando hace falta un valor inicial para la edición del campo. El prototipo del evento *OnGetText* es el siguiente:

```
type
  TFieldGetTextEvent = procedure (Sender: TField;
    var Text: string; DisplayText: Boolean) of object;
```

Un manejador de eventos para este evento debe asignar una cadena de caracteres en el parámetro *Text*, teniendo en cuenta si ésta se necesita para su visualización normal (*DisplayText* igual a *True*) o como valor inicial del editor (en caso contrario).

Inspirado en la película de romanos que pasaron ayer por la tele, he desarrollado un pequeño ejemplo que muestra el código del cliente en números romanos:

```

procedure TForm1.tbClientesCustNoGetText(Sender: TField;
  var Text: string; DisplayText: Boolean);
const
  Unidades: array [0..9] of string =
    ('', 'I', 'II', 'III', 'IV', 'V', 'VI', 'VII', 'VIII', 'IX');
  Decenas: array [0..9] of string =
    ('', 'X', 'XX', 'XXX', 'XL', 'L', 'LX', 'LXX', 'LXXX', 'XC');
  Centenas: array [0..9] of string =
    ('', 'C', 'CC', 'CCC', 'CD', 'D', 'DC', 'DCC', 'DCCC', 'CM');
  Miles: array [0..3] of string =
    ('', 'M', 'MM', 'MMM');
begin
  if Sender.AsInteger > 3999 then
    Text := 'Infinitum'
    // Hay que ser consecuentes con el lenguaje
  else
    begin
      I := Sender.AsInteger;
      Text := Miles[I div 1000] + Centenas[I div 100 mod 10] +
        Decenas[I div 10 mod 10] + Unidades[I mod 10];
    end;
end;

```

### NOTA IMPORTANTE

Más adelante presentaré los *campos calculados* de la VCL. Muchos programadores los utilizan para resolver problemas de visualización o incluso entrada de datos, aunque los campos calculados no permiten modificaciones. En la mayoría de los casos, sin embargo, el problema puede resolverse más fácilmente utilizando *OnGetText* y *OnSetText*. En particular, si alguien le pregunta cómo teclear un valor para un “campo calculado”, sospeche inmediatamente que es un problema soluble con estos eventos.

Si a algún usuario se le ocurriese la peregrina idea de teclear sus datos numéricos como números romanos, el evento adecuado para programar esto sería *OnSetText*. El prototipo del evento es el siguiente:

```

type
  TFieldSetTextEvent = procedure (Sender: TField;
    const Text: string) of object;

```

Este evento es utilizado con frecuencia para realizar cambios sobre el texto tecleado por el usuario para un campo, antes de ser asignado al mismo. Por ejemplo, un campo de tipo cadena puede convertir la primera letra de cada palabra a mayúsculas, como sucede en el caso de los nombres propios. Un campo de tipo numérico puede eliminar los separadores de millares que un usuario puede colocar para ayudarse en la edición. Como este evento se define para el campo, es independiente de la forma en que se visualice dicho campo y, como veremos al estudiar los módulos de datos, formará parte de las reglas de empresa de nuestro diseño.

Para ilustrar el uso del evento *OnSetText*, aquí está el manejador que lleva a mayúsculas la primera letra de cada palabra de un nombre:

```

procedure TForm1.tbClientesCompanySetText(Sender: TField;
  const Text: string);
var
  I: Integer;
  S: string;
begin
  S := Text;
  for I := 1 to Length(S) do
    if (I = 1) or (S[I - 1] = ' ') then
      CharUpperBuff(@S[I], 1);
  Sender.AsString := S;
end;

```

Otra situación práctica en la que *OnSetText* puede ayudar es cuando se necesite completar una entrada incompleta, en el caso de que el conjunto de valores a teclear esté limitado a ciertas cadenas. Por ejemplo, usted tiene un campo alfanumérico, en el que pueden existir tres valores: rojo, verde y azul. Si desea que el usuario pueda teclear R y que la aplicación entienda que significa *rojo*, debe utilizar *OnSetText*.

## Las propiedades *Text* y *DisplayText*

Hay dos propiedades comunes a todos los campos, que muchos programadores de Delphi suelen pasar por alto:

```

property TField.Text: string;
property TField.DisplayText: string;

```

Aunque *DisplayText* es de sólo lectura, *Text* admite lecturas y escrituras.

¿Me pregunta usted que para qué sirven? ¡Pues para casi todo! Supongamos que hay que generar un listado por impresora. Recorremos las filas del conjunto de datos, y para fila, recorremos todos sus campos. Y para cada campo, ¿qué es lo que debemos imprimir? Si contestó *DisplayText*, anótese un tanto, pero sepa que hay programadores con mucha experiencia que hubiesen utilizado *AsString*, o cosas aún peores. La ventaja de *DisplayText* es que se formatea el valor interno del campo de acuerdo con lo estipulado en su propiedad *DisplayFormat*, si es que la hay, o según dicte la respuesta a *OnGetText*, si se ha programado.

¿Y qué hay con *Text*? Suponga esta vez que colocamos un componente *TEdit* en un formulario, y pretendemos que el usuario teclee en él un valor para un campo. Para precisar, digamos que el campo es de tipo entero. Cuando el usuario teclee INTRO queremos asignar la cadena tecleada al campo. Esto es lo que haría la mayoría de la gente:

```

ClientDataSet1Campol.AsInteger := IntToStr(Edit1.Text);

```

Pero esto es lo que usted y yo debemos hacer:

```

ClientDataSet1Campol.Text := Edit1.Text;

```

Así, si hemos interceptado el evento *OnSetText*, le damos la posibilidad de ejecutarse; además de ahorrarnos las molestias de la validación y conversión.

## Caracteres correctos

Cuando se utiliza el evento *OnSetText* de un campo, es casi seguro que tendremos que modificar otra de sus propiedades:

```
type
    TFieldChars = set of Char;

property TField.ValidChars: TFieldChars;
```

*ValidChars* es utilizada por la función *IsValidChar*, también definida en *TField*, y esta última es consultada por todos los controles de datos para verificar si debe aceptar o no cada carácter que teclea el usuario. Si el programador obsesionado con los números romanos quiere torturar a sus usuarios obligándolos a teclear en este sistema, tendrá que modificar el valor de *ValidChars* para todos los campos numéricos, porque de lo contrario ningún control de datos existente aceptará letras.

A mí me ha ocurrido algo parecido: una vez tuve que escribir una aplicación para una base de datos en la que no habían utilizado campos de fechas. Por el contrario, las fechas se almacenaban en formato numérico; mi último cumpleaños, por ejemplo, se representaría como 20010704. No sólo tuve que interceptar *OnGetText* y *OnSetText* para todos ellos, sino suplicarles además que aceptasen la barra inclinada como carácter correcto dentro de un “número”.

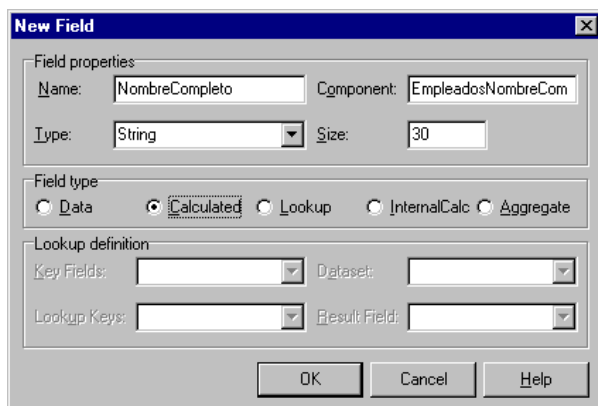
Normalmente, *ValidChars* se inicializa en el constructor del campo, pero por suerte para nosotros, podemos modificar su contenido en cualquier momento. Sin llegar a los extremos anteriores, puede ser corriente que ejecutemos instrucciones como la siguiente:

```
procedure TmodPrincipal.EmpleadosAfterOpen(DataSet: TDataSet);
begin
    // Podría haberse hecho también durante OnCreate
    EmpleadosEmpNo.ValidChars :=
        EmpleadosEmpNo.ValidChars - ['-', '+', DecimalSeparator];
end;
```

En este caso, *Empleados* extrae sus datos del fichero *customer.xml*, en el que el campo con el código de empleado se ha definido, innecesariamente, como *TFloatField*. Con la modificación anterior, evitamos que el usuario teclee el separador decimal, sea éste el que sea, o los caracteres de signo. Esto también le sucederá con algunas interfaces de acceso a Oracle, que representarán los valores enteros mediante campos *TBCDField*, una clase que admite decimales.

## Campos calculados

Una potente característica de Delphi nos permite crear *campos calculados*, que no corresponden a campos almacenados “físicamente” en su origen, sino que se calculan a partir de los campos “reales”. Por ejemplo, la antigüedad de un trabajador puede deducirse a partir de su fecha de contrato y de la fecha actual. No tiene sentido, en cambio, almacenar físicamente este dato, pues tendría que actualizarse día a día. Otro candidato a campo calculado es el nombre completo de una persona, que puede deducirse a partir del nombre y los apellidos; es conveniente, en muchos casos, almacenar nombre y apellidos en campos independientes para permitir operaciones de búsqueda sobre los mismos por separado.



Para definir campos calculados, utilizamos el comando *New field*, del menú local del Editor de Campos. El cuadro de diálogo que aparece nos sirve para suministrar el nombre que le vamos a dar al campo, el nombre de la variable asociada al objeto, el tipo de campo y su longitud, si el campo contendrá cadenas de caracteres. También hay que aclarar que el campo es *Calculated*.

### NOTA

En el siguiente capítulo presentaré los *campos calculados internos*, que solamente son soportados en este momento por *TClientDataSet*. En cambio, los campos calculados de esta sección son admitidos por cualquier descendiente de *TDataSet*.

Observe que en ninguna parte de este cuadro de diálogo se ha escrito algo parecido a una fórmula. El algoritmo de cálculo del campo se especifica realmente durante la respuesta al evento *OnCalcFields* de la tabla a la cual pertenece el campo. Durante el intervalo de tiempo que dura la activación de este evento, la tabla se sitúa automáticamente en el estado *dsCalcFields*. En este estado, no se permiten las asignaciones a campos que no sean calculados: no se puede “aprovechar” el evento para realizar actualizaciones sobre campos físicos de la tabla, pues se produce una excepción. Tampoco debemos mover la fila activa, pues podemos provocar una serie infinita de llamadas recursivas. El evento *OnCalcFields* se lanza precisamente cuando se cambia la

fila activa de la tabla. También puede lanzarse cuando se realiza alguna modificación en los campos de una fila, si la propiedad lógica *AutoCalcFields* del conjunto de datos vale *True*.

Para la tabla *employee.xml*, perteneciente a los ejemplos de Delphi 6, podríamos definir un par de campos calculados con esta estructura:

Campo	Tipo	Tamaño
<i>NombreCompleto</i>	<i>String</i>	30
<i>Antigüedad</i>	<i>Integer</i>	

Esto se realiza en el Editor de Campos. Después seleccionamos la tabla de empleados, y mediante el Inspector de Objetos creamos un manejador para su evento *OnCalcFields*; suponemos que el nombre del componente de tabla es *Empleados*:

```

procedure TmodDatos.EmpleadosCalcFields(DataSet: TDataSet);
var
    Y1, M1, D1, Y2, M2, D2: Word;
begin
    EmpleadosNombreCompleto.Value :=
        EmpleadosLastName.Value + ', ' + EmpleadosFirstName.Value;
    if not EmpleadosHireDate.IsNull then
    begin
        DecodeDate(Date, Y1, M1, D1);
        DecodeDate(EmpleadosHireDate.Value, Y2, M2, D2);
        Dec(Y1, Y2);
        if (M2 > M1) or (M1 = M2) and (D2 > D1) then Dec(Y1);
        EmpleadosAntigüedad.Value := Y1;
    end;
end;

```

Hay que tener cuidado con los campos que contienen valores nulos. En principio basta con que uno de los campos que forman parte de la expresión sea nulo para que la expresión completa también lo sea, en el caso de una expresión que no utilice operadores lógicos binarios (**and/or**). Esto es lo que hacemos antes de calcular el valor de la antigüedad, utilizando la propiedad *IsNull*. Por el contrario, he asumido que el nombre y los apellidos no pueden ser nulos, y no me he tomado la molestia de comprobar este detalle.

## Campos de referencia

Hay un caso muy frecuente de campo calculado, en el que el cálculo consiste en una búsqueda. Si analizamos la tabla *orders.xml*, que contiene pedidos, encontraremos una columna *CustNo*, para almacenar un código de cliente. Los códigos de cliente hacen referencia a los valores almacenados en la columna *CustNo*, pero de *customer.xml*, otra tabla diferente. El lector familiarizado con bases de datos SQL reconocerá inmediatamente que se trata de la típica relación de integridad referencial.

Teniendo el código de cliente de un pedido, sería deseable disponer del nombre de la empresa representada por ese código. Si estamos trabajando con una base de datos SQL, lo mejor es que sea el propio servidor el responsable de la traducción. En vez de pedir registros de pedidos “a secas”, mediante una consulta sobre esa tabla, podríamos calcular el encuentro natural con la tabla de clientes, y traernos de ésta solamente el nombre de la empresa. Cuando estudiemos DataSnap, veremos cómo ignorar el campo adicional durante las actualizaciones.

Pero hay otro método, que funciona en ocasiones: realizar la traducción en la aplicación cliente. Podríamos crear un campo calculado que buscase un código de cliente para transformarlo en algo más descriptivo. No sería eficiente realizar directamente la búsqueda en el servidor, por lo que tendríamos que inventarnos algún sistema de caché. En cualquier caso, la tabla de referencia tendría que ser relativamente pequeña para que esto tuviese sentido. Para el código de cliente, se trata de una mala idea, a no ser que tengamos cuatro gatos... perdón, cuatro clientes en la base de datos. Pero sí sería factible la traducción de los códigos de formas de pago, de provincias, de tipos de impuestos. Y una vez lanzados por este camino, ¿qué tal si recibiéramos alguna ayuda para modificar estos campos?

Hace ya mucho tiempo, en la época de Paradox, Delphi introdujo un tipo especial de campo calculado: los campos de referencia, o *lookup fields*. Gracias a información adicional que se suministra durante su creación, estos campos:

- 1 Traducen códigos a descripciones, buscando en un segundo conjunto de datos.
- 2 Los controles de datos pueden aprovechar la información semántica adicional para ayudarnos a modificar el valor de la columna base.

¡Mucho cuidado, porque el segundo punto no significa que los campos de referencia sean actualizables! Sólo sucede que algunos controles de datos pueden aprovechar la información adicional asociada a ellos. Esto lo veremos enseguida.

Los campos de referencia se crean por medio del comando de menú *New field* del menú local del Editor de Campos; es el mismo cuadro de diálogo con el que creamos campos calculados. Pongamos por caso que queremos un campo que nos dé el nombre del cliente asociado a un pedido. Utilizaremos ahora las tablas de pedidos (*Pedidos*, asociada a *orders.xml*) y de clientes (*Clientes*, asociada a *customer.xml*). Nos vamos a la tabla de pedidos, activamos el Editor de Campos y el comando *New field*. La información de la parte superior del cuadro de diálogo es la misma que para un campo calculado:

Campo	Tipo	Tamaño
<i>Cliente</i>	<i>String</i>	30

Después, tenemos que indicar el tipo de campo como *Lookup*; de este modo, se activan los controles de la parte inferior del diálogo. Estos son los valores que ha que suministrar, y su significado:



- Key fields* El campo o conjunto de campos que sirven de base a la referencia. Están definidos en la tabla base, y por lo general, aunque no necesariamente, tienen definida una clave externa. En este ejemplo, teclee *CustNo*.
- Dataset* Conjunto de datos en el cual se busca la referencia: use *Cientes*.
- Lookup keys* Los campos de la tabla de referencia sobre los cuales se realiza la búsqueda. Para nuestro ejemplo, utilice *CustNo*; aunque es el mismo nombre que hemos tecleado en *Key fields*, esta vez nos estamos refiriendo a la tabla de clientes, en vez de la de pedidos.
- Result field* El campo de la tabla de referencia que se visualiza. Seleccione *Company*, el nombre de la empresa del cliente.

Un error frecuente es dejar a medias el diálogo de definición de un campo de búsqueda. Esto sucede cuando el programador inadvertidamente pulsa la tecla INTRO, pensando que de esta forma selecciona el próximo control de dicha ventana. Cuando esto sucede, no hay forma de volver al cuadro de diálogo para terminar la definición. Una posibilidad es eliminar la definición parcial y comenzar desde cero. La segunda consiste en editar directamente las propiedades del campo recién creado. La siguiente tabla enumera estas propiedades y su correspondencia con los controles del diálogo de definición:

Propiedad	Correspondencia
<i>Lookup</i>	Siempre igual a <i>True</i> para estos campos
<i>KeyFields</i>	Campos de la tabla en los que se basa la búsqueda ( <i>Key fields</i> )
<i>LookupDataset</i>	Tabla de búsqueda ( <i>Dataset</i> ).
<i>LookupKeyFields</i>	Campos de la tabla de búsqueda que deben corresponder al valor de los <i>KeyFields</i> ( <i>Lookup keys</i> )
<i>LookupResultField</i>	Campo de la tabla de búsqueda cuyo valor se toma ( <i>Result field</i> )

### RECOMENDACION

El campo que se selecciona en *LookupResultField* debe ser, casi siempre, un campo casi único. Por ejemplo, en *Cientes* seleccionamos *Company*, porque es muy difícil que dos empresas tengan exactamente el mismo nombre. Nunca elegiríamos el nombre de la ciudad, por ejemplo, a no ser que sólo tengamos un cliente por ciudad. Sucede en ocasiones que cierta tabla de referencia no tiene un solo campo que sea

único, como pasa con la tabla de empleados, en la que hay apellidos y nombres repetidos. En este caso, podríamos crear un campo calculado en *Empleados* para obtener el nombre completo concatenando nombres y apellidos, y utilizar este campo como “traducción” del código de empleado almacenado en la tabla de pedidos.

Debe saber que los conjuntos de datos que se asocian a propiedades *LookupDataset* de campos de referencia se abren de forma automática cada vez que la tabla base se activa. Sin embargo, al cerrarse ésta última, no se restaura el estado inicial de la tabla de referencia. Si un campo de *Pedidos* hace referencia a *Clientes* y abrimos la tabla de pedidos, entonces también se activa la de clientes, si no se encontrase inicialmente abierta.



¿Y qué hay de la ayuda prometida para la edición? La imagen anterior corresponde a la aplicación *CamposReferencias*, del CD. La columna titulada *Cliente* está asociada precisamente a un campo de referencia. Observe que la rejilla, sin ayuda ninguna por parte nuestra, es capaz de desplegar un combo para cambiar el cliente asociado a un pedido. Naturalmente, cuando seleccionemos un nombre de empresa en la lista desplegable, estaremos modificando el valor del campo *CustNo*, no el de *Cliente*, que como era de esperar sigue siendo un campo de sólo lectura.

### ADVERTENCIA

Como ya he mencionado, el ejemplo del código de cliente no es muy realista. En circunstancias típicas, una base de datos tendrá muchos registros de clientes. Por una parte, no es recomendable traer todos esos registros a memoria. Además, ¿qué sentido tiene permitir que un usuario escoja uno de entre diez mil clientes con un cobarde combo de siete filas? Ninguno, por supuesto.

Cuando estudiemos los controles de datos, veremos que el control *TDBLookupComboBox* nos permite realizar una búsqueda en otra tabla sin necesidad de definir campos de referencia.

## El orden de evaluación de los campos

Cuando se definen campos calculados y campos de búsqueda sobre una misma tabla, los campos de búsqueda se evalúan antes que los campos calculados. Así, durante el algoritmo de evaluación de los campos calculados se pueden utilizar los valores de los campos de búsqueda.

Tomemos como ejemplo la tabla *items.xml*, que cargaremos en *Detalles*. Este fichero contiene líneas de detalles de pedidos, y en cada registro hay una referencia al código del artículo vendido (*PartNo*), además de la cantidad (*Qty*) y el descuento aplicado (*Discount*). A partir del campo *PartNo*, y utilizando el fichero *parts.xml* como tabla de referencia para los artículos, puede crearse un campo de búsqueda *Precio* que devuelva el precio de venta del artículo en cuestión, extrayéndolo de la columna *List-Price* de *parts.xml*. En este escenario, podemos crear un campo calculado, de nombre *SubTotal* y de tipo *Currency*, que calcule el importe total de esa línea del pedido:

```
procedure TmodDatos.DetallesCalcFields(DataSet: TDataSet);
begin
    DetallesSubTotal.Value :=
        DetallesQty.Value *                // Campo base
        DetallesPrecio.Value *            // Campo de búsqueda
        (1 - DetallesDiscount.Value / 100); // Campo base
end;
```

Lo importante es que, cuando se ejecuta este método, ya se han evaluado los campos de búsqueda, y tenemos un valor utilizable en el campo *Precio*.

## Creación de tablas para MyBase

Hasta el momento, hemos trabajado con ficheros XML creados por otras personas. ¿Cómo los han creado? Ampliando la pregunta, ¿cómo se crea una tabla dentro de una base de datos? Depende. Si es una base de datos SQL, la mejor forma es tener a mano una instrucción **create table** a mano, o generarla dinámicamente en una cadena de caracteres, y ejecutarla entonces sobre la base de datos. Discusión zanjada.

Cuando se trata de tablas de Paradox y dBase, sin embargo, es preferible utilizar funciones del API del BDE. Y sí, también es posible crear este tipo de tablas con instrucciones SQL, pero se perderían muchas posibilidades. Algunas de las funciones de creación de tablas del BDE son encapsuladas por el componente *TTable*. La existencia de esos métodos llevó a muchos programadores a intentar retorcer el BDE al máximo, para usarlos en la creación de tablas SQL. Repito: sería un disparate.

Pero si se trata de crear tablas para MyBase, al menos partiendo desde cero, la única forma es utilizar un conjunto de métodos de *TClientDataSet* muy parecidos a los métodos de *TTable* para crear tablas con el BDE. Hagamos la lista de la compra. Para crear una tabla de MyBase:

- 1 Necesitamos definir los campos que va a tener. Esto implica definir nombres, tipos, opcionalmente tamaño o precisión y escala, y especificar si permiten o no valores nulos.
- 2 Opcionalmente, podemos definir uno o más índices físicos. Como todavía no hemos estudiado los índices en MyBase, me saltaré este paso por el momento.
- 3 Finalmente, llamamos a un método muy sencillo, *CreateDataSet*, para crear la tabla.

¿Cómo podemos definir los campos? Para un *TClientDataSet*, lo más sencillo es crear campos “de datos” dentro del propio componente. Un campo de dato se crea entrando en el Editor de Campos, ejecutando el comando *New field*, y añadiendo un campo de tipo *Data*.



Para cada campos que añadimos, hay que configurar las siguientes propiedades:

- 1 Lo más importante es la clase a la que pertenece el objeto del campo. Pero esto lo determina el diálogo de creación de campos, lo mismo que el valor de la propiedad *FieldKind*, que siempre será *fkData*.
- 2 La propiedad *FieldName* es asignada también por el diálogo, y contiene el nombre de la columna.
- 3 Si la columna será una cadena de caracteres, fija o variable, el diálogo asignará *Size*. También se usa *Size* para algunos tipos de campos blob.
- 4 La primera propiedad que no se asigna automáticamente es *Precision*. Si el campo pertenece a uno de los tipos *BCD* o *FMTBCD*, *Precision* es el número total de dígitos admitidos, y curiosamente *Size* es el número de dígitos decimales. Preste atención a este detalle, porque contradice la intuición; la mía, al menos.
- 5 Es muy conveniente tener en cuenta el valor de *Required*. Si vale *True*, el campo no admitirá valores nulos.

La llamada a *CreateDataSet*, que también puede ejecutarse en tiempo de diseño, utilizando el menú de contexto del *TClientDataSet*, solamente inicializa un conjunto de datos en memoria. Si queremos guardar el fichero sin registros en disco, podemos ejecutar el método *SaveToFile*, que también está disponible en el menú de contexto.

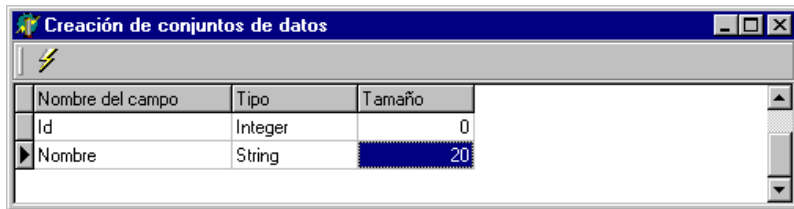
## Información sobre campos

Delphi distingue entre la información sobre los campos físicos de un conjunto de datos y los componentes de acceso a los mismos. La información sobre campos se encuentra en la propiedad *FieldDefs*, y es el sistema que debe utilizarse para la creación dinámica de conjuntos de datos que no pertenecen a MyBase.

La clase *TFieldDefs* es una colección de elementos *TFieldDef*. Las propiedades publicadas de esta última clase son:

Propiedad	Significado
<i>Attributes</i>	¿Admite nulos el campo? ¿Es de sólo lectura, tiene tamaño fijo?
<i>ChildDefs</i>	Util para tablas anidadas
<i>DataType</i>	El tipo del campo, ¿qué otra cosa podría ser?
<i>Name</i>	Nombre del campo
<i>Precision</i>	Para campos de tipo BCD, la precisión
<i>Size</i>	Para cadenas de caracteres, el tamaño; para campos de tipo BCD, el número total de decimales

¡Cuidado con *Name*! La clase *TFieldDef* no descende de *TComponent*, sino de *TPersistent*, a través de *TCollectionItem* y *TNamedItem*. Por lo tanto, *Name* no es el nombre del componente; es el nombre de un campo.



He incluido en el CD un proyecto muy sencillo, llamado *Creacion*. En él utilizo un conjunto de datos clientes preparado para que el usuario pueda introducir una lista de campos. La acción *Crear*, al ejecutarse, pide al usuario un nombre de fichero, crea un segundo conjunto de datos cliente de acuerdo a la información suministrada por el usuario y lo guarda en el disco, en formato XML. Este es el método que se ejecuta en respuesta a la acción:

```

procedure TwndPrincipal.CrearExecute(Sender: TObject);
var
    FType: TFieldType;
begin
    Nuevo.Close;
    Nuevo.FieldDefs.Clear;
    Esquema.First;
    while not Esquema.Eof do
        begin
            if SameText(EsquemaTipo.Value, 'STRING') then
                FType := ftString
            else if SameText(EsquemaTipo.Value, 'INTEGER') then
                FType := ftInteger
            else if SameText(EsquemaTipo.Value, 'DATETIME') then
                FType := ftDateTime
            else if SameText(EsquemaTipo.Value, 'BOOLEAN') then
                FType := ftBoolean
            else if SameText(EsquemaTipo.Value, 'CURRENCY') then
                FType := ftCurrency
            else
                raise Exception.Create('Tipo no soportado');
        end
    end

```

```

        Nuevo.FieldDefs.Add(EschemaNombre.Value, FType,
            EsquemaTamano.Value);
        Esquema.Next;
    end;
    Nuevo.CreateDataSet;
    if SaveDialog.Execute then
        Nuevo.SaveToFile(SaveDialog.FileName, dfXML);
    end;

```

Como puede ver, solamente he implementado un conjunto reducido de tipos de campos. Para añadir definiciones a *FieldDefs* hay dos métodos:

```

function TFieldDef.AddFieldDef: TFieldDef;
procedure TFieldDef.Add(
    const Name: string; DataType: TFieldType; Size: Integer = 0;
    Required: Boolean = False);

```

Aunque he utilizado el segundo, es recomendable utilizar el primero, *AddFieldDef*, porque nos permite configurar directamente aquellas propiedades de la columna en las que estamos interesados.

## Controles de datos

ESTE CAPÍTULO TRATA ACERCA DE LOS CONTROLES que ofrece Delphi para visualizar y editar información procedente de bases de datos, la filosofía general en que se apoyan y las particularidades de cada uno de ellos. Es el momento de ampliar, además, nuestros conocimientos acerca de un componente esencial para la sincronización de estos controles, el componente *TDataSource*. Veremos cómo un control “normal” puede convertirse, gracias a una sencilla técnica basada en este componente, en un flamante control de acceso a datos. Por último, estudiaremos cómo manejar campos blob, que pueden contener grandes volúmenes de información, dejando su interpretación a nuestro cargo.

### Controles *data-aware*

Los controles de bases de datos son conocidos en la jerga delfica como controles *data-aware*. Estos controles son, generalmente, versiones especializadas de controles “normales”. Por ejemplo, *TDBMemo* es una versión orientada a bases de datos del conocido *TMemo*. Al tener por separado los controles de bases de datos y los controles tradicionales, Delphi evita que una aplicación que no haga uso de bases de datos tenga que cargar con todo el código necesario para estas operaciones. No sucede así con Visual Basic, lenguaje en que todos los controles pueden potencialmente conectarse a una base de datos.



Los controles de acceso a datos de Delphi se pueden dividir en dos grandes grupos:

- Controles asociados a campos
- Controles asociados a conjuntos de datos

Los controles asociados a campos visualizan y editan una columna particular de una tabla. Los componentes *TDBEdit* (cuadros de edición) y *TDBImage* (imágenes almacenadas en campos gráficos) pertenecen a este tipo de controles. Los controles aso-

ciados a conjuntos de datos, en cambio, trabajan con la tabla o consulta como un todo. Las rejillas de datos y las barras de navegación son los ejemplos más conocidos de este segundo grupo. Todos los controles de acceso a datos orientados a campos tienen un par de propiedades para indicar con qué datos trabajan: *DataSource* y *DataField*; estas propiedades pueden utilizarse en tiempo de diseño. Por el contrario, los controles orientados al conjunto de datos solamente disponen de la propiedad *DataSource*. La conexión con la fuente de datos es fundamental, pues es este componente quien notifica al control de datos de cualquier alteración en la información que debe visualizar.

Casi todos los controles de datos permiten, de una forma u otra, la edición de su contenido. En el capítulo 22 explicaremos que hace falta que la tabla esté en uno de los modos *dsEdit* ó *dsInsert* para poder modificar el contenido de un campo. Pues bien, todos los controles de datos son capaces de colocar a la tabla asociada en este modo, de forma automática, cuando se realizan modificaciones en su interior. Este comportamiento se puede modificar con la propiedad *AutoEdit* del *data source* al que se conectan. Cada control, además, dispone de una propiedad *ReadOnly*, que permite utilizar el control únicamente para visualización.

Actualmente, los controles de datos de Delphi son los siguientes:

Nombre de la clase	Explicación
<b>Controles orientados a campos</b>	
<i>TDBText</i>	Textos no modificables (no consumen recursos)
<i>TDBEdit</i>	Cuadros de edición
<i>TDBMemo</i>	Textos sin formato con múltiples líneas
<i>TDBImage</i>	Imágenes BMP y WMF
<i>TDBListBox</i>	Cuadros de lista (contenido fijo)
<i>TDBComboBox</i>	Cuadros de combinación (contenido fijo)
<i>TDBCheckBox</i>	Casillas de verificación (dos estados)
<i>TDBRadioGroup</i>	Grupos de botones (varios valores fijos)
<i>TDBLookupListBox</i>	Valores extraídos desde otra tabla asociada
<i>TDBLookupComboBox</i>	Valores extraídos desde otra tabla asociada
<i>TDBRichEdit</i>	Textos en formato RTF
<b>Controles orientados a conjuntos de datos</b>	
<i>TDBGrid</i>	Rejillas de datos para la exploración
<i>TDBNavigator</i>	Control de navegación y estado
<i>TDBCtrlGrid</i>	Rejilla que permite incluir controles

Ahora nos ocuparemos principalmente de los controles orientados a campos. El resto de los controles serán estudiados en el siguiente capítulo.



**NOTA**

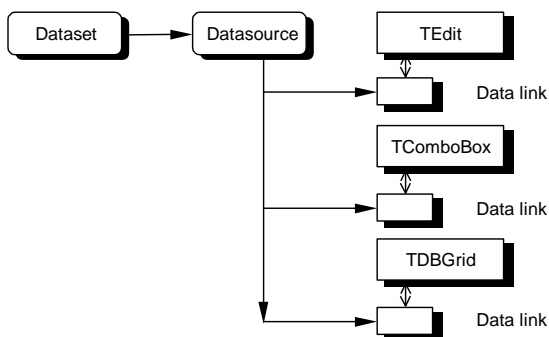
He omitido intencionalmente el control *TDBChart* de la lista anterior, por trabajar con un sistema de notificaciones distinto a los demás. Este componente se estudiará más adelante.

## Enlaces de datos y notificaciones

Según lo explicado, todos los controles de datos deben ser capaces de reconocer y participar en el juego de las notificaciones, y esto supone la existencia de un montón de código común a todos ellos. Pero, si observamos el diagrama de herencia de la VCL, notaremos que no existe un ancestro compartido propio para los controles *data-aware*. ¿Qué solución se ha utilizado en la VCL para evitar la duplicación de código?

La respuesta la tiene un objeto generalmente ignorado: *TDataLink*, y su descendiente *TFieldDataLink*. Este desconocimiento es comprensible, pues no es un componente visual, y sólo es imprescindible para el desarrollador de componentes. Cada control de datos crea durante su inicialización un componente interno perteneciente a una de estas clases. Es este componente interno el que se conecta a la fuente de datos, y es también a éste a quien la fuente de datos envía las notificaciones acerca del movimiento y modificaciones que ocurren en el conjunto de datos subyacente. Todo el tratamiento de las notificaciones se produce entonces, al menos de forma inicial, en el *data link*. Esta técnica es conocida como *delegación*, y nos evita el uso de la herencia múltiple, recurso no soportado por la VCL hasta el momento.

El siguiente esquema muestra la relación entre los componentes de acceso y edición de datos:



## Creación de controles de datos

Podemos crear controles de datos en un formulario trayendo uno a uno los componentes deseados desde la Paleta e inicializando sus propiedades *DataSource* y *DataField*. Esta es una tarea tediosa. Muchos programadores utilizaban en las primeras versiones de Delphi el Experto de Datos (*Database form expert*) para crear un formula-

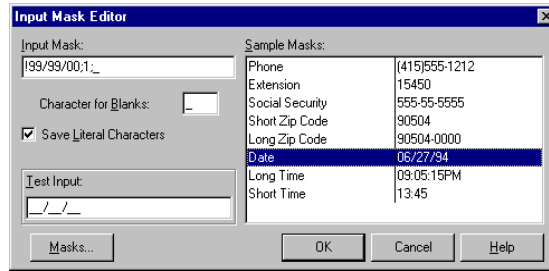
rio con los controles deseados, y luego modificar este diseño inicial. Este experto, sin embargo, tiene un par de limitaciones importantes: en primer lugar, trabaja con un tamaño fijo de la ventana, lo cual nos obliga a realizar desplazamientos cuando no hay espacio para los controles, aún cuando aumentando el tamaño de la ventana se pudiera resolver el problema. No obstante, el inconveniente principal es que siempre genera un nuevo componente *TTable* ó *TQuery*, ligado al BDE: no permite utilizar componentes existentes de otras clases. Esto es un problema, pues lo usual es definir primero los conjuntos de datos en un módulo aparte, para poder programar reglas de empresa de forma centralizada.

Podemos arrastrar campos desde el Editor de Campos sobre un formulario. Cuando lo hacemos, se crea automáticamente un control de datos asociado al campo. Junto con el control, se crea también una etiqueta, de clase *TLabel*, con el título extraído de la propiedad *DisplayLabel* del campo. Recuerde que esta propiedad coincide inicialmente con el nombre del campo, de modo que si las columnas de sus tablas tienen nombre crípticos como *NomCli*, es conveniente modificar primero las etiquetas de visualización en los campos antes de crear los controles. Adicionalmente, Delphi asigna a la propiedad *FocusControl* de los componentes *TLabel* creados el puntero al control asociado. De este modo, si la etiqueta tiene un carácter subrayado, podemos hacer uso de este carácter como abreviatura para la selección del control.

En cuanto al tipo de control creado, Delphi tiene sus reglas por omisión: casi siempre se crea un *TDBEdit*. Si el campo es un campo de búsqueda, crea un componente *TDBLookupComboBox*. Si es un memo o un campo gráfico, se crea un control *TDBMemo* o un *TDBImage*. Si el campo es lógico, se crea un *TDBChechBox*. Estas reglas implícitas pueden modificarse por medio del Diccionario de Datos, aunque sólo podemos utilizar este último con los conjuntos de datos del BDE. Si el campo que se arrastra tiene definido un conjunto de atributos, el control a crear se determina por el valor almacenado en la propiedad *TControlClass* del conjunto de atributos en el Diccionario de Datos.

## Los cuadros de edición

La forma más general de editar un campo simple es utilizar un control *TDBEdit*. Este componente puede utilizarse con campos de cadenas de caracteres, numéricos, de fecha y de cualquier tipo en general que permita conversiones desde y hacia cadenas de caracteres. Los cuadros de edición con conexión a bases de datos se derivan de la clase *TCustomMaskEdit*. Esto es así para poder utilizar la propiedad *EditMask*, perteneciente a la clase *TField*, durante la edición de un campo. Sin embargo, *EditMask* es una propiedad protegida de *TDBEdit*; el motivo es permitir que la máscara de edición se asigne directamente desde el campo asociado, dentro de la VCL, y que el programador no pueda interferir en esta asignación. Si el campo tiene una máscara de edición definida, la propiedad *IsMasked* del cuadro de edición se vuelve verdadera.



Windows no permite definir alineación para el texto de un cuadro de edición, a no ser que el estilo del cuadro sea multilineas. Si nos fijamos un poco, el control *TEdit* estándar no tiene una propiedad *Alignment*. Sin embargo, es común mostrar los campos numéricos de una tabla justificados a la derecha. Es por eso que, en el código fuente de la VCL, se realiza un truco para permitir los distintos tipos de alineación en los componentes *TDBEdit*. Es importante comprender que *Alignment* solamente funciona durante la visualización, no durante la edición. La alineación se extrae de la propiedad correspondiente del componente de acceso al campo.

Los eventos de este control que utilizaremos con mayor frecuencia son *OnChange*, *OnKeyPress*, *OnKeyDown* y *OnExit*. *OnChange* se produce cada vez que el texto en el control es modificado. Puede causar confusión el que los componentes de campos tengan un evento también nombrado *OnChange*. Este último evento se dispara cuando se modifica el contenido del campo, lo cual sucede cuando se realiza una asignación al componente de campo. Si el campo se está editando en un *TDBEdit*, esto sucede al abandonar el control, o al pulsar INTRO estando el control seleccionado. En cambio, el evento *OnChange* del control de edición se dispara cada vez que se cambia algo en el control. El siguiente evento muestra cómo pasar al control siguiente cuando se alcanza la longitud máxima admitida por el editor. Este comportamiento era frecuente en programas realizados para MS-DOS:

```
procedure TForm1.DBEdit1Change(Sender: TObject);
begin
    if Visible then
        with Sender as TDBEdit do
            if Length(Text) >= MaxLength then
                SelectNext(TDBEdit(Sender), True, True);
end;
```

## Editores de texto

Cuando el campo a editar contiene varias líneas de texto, debemos utilizar un componente *TDBMemo*. Si queremos además que el texto tenga formato y permita el uso de negritas, cursivas, diferentes colores y alineaciones podemos utilizar el componente *TDBRichEdit*.

*TDBMemo* está limitado a un máximo de 32KB de texto, además de permitir un solo tipo de letra y de alineación para todo su contenido. Las siguientes propiedades han

sido heredadas del componente *TMemo* y determinan la apariencia y forma de interacción con el usuario:

Propiedad	Significado
<i>Alignment</i>	La alineación del texto dentro del control.
<i>Lines</i>	El contenido del control, como una lista de cadenas de caracteres.
<i>ScrollBars</i>	Determina qué barras de desplazamiento se muestran.
<i>WantTabs</i>	Si está activa, el editor interpreta las tabulaciones como tales; en caso contrario, sirven para seleccionar el próximo control.
<i>WordWrap</i>	Las líneas pueden dividirse si sobrepasan el margen derecho.

La propiedad *AutoDisplay* es específica de este tipo de controles. Como la carga y visualización de un texto con múltiples líneas puede consumir bastante tiempo, se puede asignar *False* a esta propiedad para que el control aparezca vacío al mover la fila activa. Luego se puede cargar el contenido en el control pulsando INTRO sobre el mismo, o llamando al método *LoadMemo*.

El componente *TDBRichEdit*, por su parte, es similar a *TDBMemo*, excepto por la mayor cantidad de eventos y las propiedades de formato. Al estar basado en el componente *TRichEdit*, permite mostrar y editar textos en formato plano y en formato RTF.

#### ADVERTENCIA

El control *TDBRichEdit* no nota los cambios en el formato del texto asociado al campo. Si añadiésemos acciones de edición para activar y desactivar negritas en uno de estos controles, por ejemplo, notaríamos que el conjunto de datos no pasaría al estado *dsEdit* cuando cambiásemos el formato de una sección de texto ya existente.

## Textos no editables

El tipo *TLabel* tiene un equivalente *data-aware*: el tipo *TDBText*. Mediante este componente, se puede mostrar información como textos no editables. Gracias a que este control descende por herencia de la clase *TGraphicControl*, desde el punto de vista de Windows no es una ventana y no consume recursos. Si utilizamos un *TDBEdit* con la propiedad *ReadOnly* igual a *True*, consumiremos un *handle* de ventana. En compensación, con el *TDBEdit* podemos seleccionar texto y copiarlo al Portapapeles, cosa imposible de realizar con un *TDBText*.

Además de las propiedades usuales en los controles de bases de datos, *DataSource* y *DataField*, la otra propiedad interesante es *AutoSize*, que indica si el ancho del control se ajusta al tamaño del texto o si se muestra el texto en un área fija, truncándolo si la sobrepasa.

## Combos y listas con contenido fijo

Los componentes *TDBComboBox* y *TDBListBox* son las versiones *data-aware* de *TComboBox* y *TListBox*, respectivamente. Se utilizan, preferentemente, cuando hay un número bien determinado de valores que puede aceptar un campo, y queremos ayudar al usuario para que no tenga que teclear el valor, sino que pueda seleccionarlo con el ratón o el teclado. En ambos casos, la lista de valores predeterminados se indica en la propiedad *Items*, de tipo *TStrings*.

De estos dos componentes, el cuadro de lista es el menos utilizado. No permite introducir valores diferentes a los especificados; si el campo asociado del registro activo tiene ya un valor no especificado, no se selecciona ninguna de las líneas. Tampoco permite búsquedas incrementales sobre listas ordenadas. Si las opciones posibles en el campo son pocas, la mayoría de los usuarios y programadores prefieren utilizar el componente *TDBRadioGroup*, que estudiaremos en breve.

En cambio, *TDBComboBox* es más flexible. En primer lugar, nos deja utilizar tres estilos diferentes de interacción mediante la propiedad *Style*; en realidad son cinco estilos, pero dos de ellos tienen que ver más con la apariencia del control que con la interfaz con el usuario:

Estilo	Significado
<i>csSimple</i>	La lista siempre está desplegada. Se pueden teclear valores que no se encuentran en la lista.
<i>csDropDown</i>	La lista está inicialmente recogida. Se pueden teclear valores que no se encuentran en la lista.
<i>csDropDownList</i>	La lista está inicialmente recogida. No se pueden teclear valores que no se encuentren en la lista.
<i>csOwnerDrawFixed</i>	El contenido de la lista lo dibuja el programador. Líneas de igual altura.
<i>csOwnerDrawVariable</i>	El contenido de la lista lo dibuja el programador. La altura de las líneas la determina el programador.

Por otra parte, la propiedad *Sorted* permite ordenar dinámicamente los elementos de la lista desplegable de los combos. Los combos con el valor *csDropDownList* en la propiedad *Style*, y cuya propiedad *Sorted* es igual a *True*, permiten realizar búsquedas incrementales. Si, por ejemplo, un combo está mostrando nombres de países, al teclear la letra *A* nos situaremos en Abisinia, luego una *N* nos llevará hasta la Nantártida, y así en lo adelante. Si queremos iniciar la búsqueda desde el principio, o sea, que la *N* nos sitúe sobre Nepal, podemos pulsar ESC o RETROCESO ... o esperar dos segundos. Esto último es curioso, pues la duración de ese intervalo está incrustada en el código de la VCL y no puede modificarse fácilmente.

Cuando el estilo de un combo es *csOwnerDrawFixed* ó *csOwnerDrawVariable*, es posible dibujar el contenido de la lista desplegable; para los cuadros de lista, *Style* debe valer

*lbOwnerDrawFixed* ó *lbOwnerDrawVariable*. Si utilizamos alguno de estos estilos, tenemos que crear una respuesta al evento *OnDrawItem* y, si el estilo de dibujo es con alturas variables, el evento *OnMeasureItem*. Estos eventos tienen los siguientes parámetros:

```
procedure TForm1.DBComboBox1DrawItem(Control: TWinControl;
  Index: Integer; Rect: TRect; State: TOwnerDrawState);
begin
end;

procedure TForm1.DBComboBox1MeasureItem(Control: TWinControl;
  Index: Integer; var Height: Integer)
begin
end;
```

Para ilustrar el uso de estos eventos, crearemos un combo que pueda seleccionar de una lista de países, con el detalle de que cada país aparezca representado por su bandera (o por lo que más le apetezca dibujar). Vamos a inicializar el combo en el evento de creación de la ventana, y para complicar un poco el código leeremos los mapas de bits necesarios desde un fichero XML de países:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  Bmp: TBitmap;
begin
  with TClientDataSet.Create(nil) do
    try
      FileName := 'paises.xml';
      Open;
      while not Eof do
        begin
          Bmp := TBitmap.Create;
          try
            Bmp.Assign(FieldByName('Bandera'));
            DBComboBox1.Items.AddObject(
              FieldByName('Pais').AsString, Bmp);
          except
            Bmp.Free;
            raise;
          end;
          Next;
        end;
      finally
        Free;
      end;
    end;
```

El código que dibuja el mapa de bits al lado del texto es el siguiente:

```
procedure TForm1.DBComboBox1DrawItem(Control: TWinControl;
  Index: Integer; Rect: TRect; State: TOwnerDrawState);
begin
  with Control as TDBComboBox do
    begin
      Canvas.FillRect(Rect);
      Canvas.StretchDraw(
        Bounds(Rect.Left + 2, Rect.Top, ItemHeight, ItemHeight),
```

```

        TBitmap(Items.Objects[Index]));
        TextOut(Rect.Left + ItemHeight + 6, Rect.Top, Items[Index]);
    end;
end;

```



Una propiedad poco conocida de *TDBComboBox*, que éste hereda de *TComboBox*, es *DroppedDown*, de tipo lógico. *DroppedDown* es una propiedad de tiempo de ejecución, y permite saber si la lista está desplegada o no. Pero también permite asignaciones, para ocultar o desplegar la lista. Si el usuario quiere que la tecla + del teclado numérico despliegue todos los combos de una ficha, puede asignar *True* a la propiedad *KeyPreview* del formulario y crear la siguiente respuesta al evento *OnKeyDown*:

```

procedure TForm1.FormKeyDown(Sender: TObject;
    var Key: Word; Shift: TShiftState);
begin
    if (Key = VK_ADD) and (ActiveControl is TCustomComboBox) then
        begin
            TCustomComboBox(ActiveControl).DroppedDown := True;
            Key := 0;
        end;
end;

```

## Combos y listas de búsqueda

En cualquier diseño de bases de datos abundan los campos sobre los que se han definido restricciones de integridad referencial. Estos campos hacen referencia a valores almacenados como claves primarias de otras tablas. Pero casi siempre estos enlaces se realizan mediante claves artificiales, como es el caso de los códigos. ¿Qué me importa a mí que la Coca-Cola sea el artículo de código 4897 de mi base de datos particular? Soy un enemigo declarado de los códigos: en la mayoría de los casos se utilizan para permitir relaciones más eficientes entre tablas, por lo que deben ser internos a la aplicación. El usuario, en mi opinión, debe trabajar lo menos posible con códigos. ¿Qué es preferible, dejar que el usuario teclee cuatro dígitos, 4897, o que teclee el prefijo del nombre del producto?

Por estas razones, cuando tenemos que editar un campo de este tipo es preferible utilizar la clave verdadera a la clave artificial. En el caso del artículo, es mejor que el usuario pueda seleccionar el nombre del artículo que obligarle a introducir un código. Esta traducción, de código a descripción, puede efectuarse a la hora de visualizar mediante los campos de búsqueda, que ya hemos estudiado. Estos campos, no obs-

tante, son sólo para lectura; si queremos editar el campo original, debemos utilizar los controles *TDBLookupListBox* y *TDBLookupComboBox*.

Las siguientes propiedades son comunes a las listas y combos de búsqueda, y determinan el algoritmo de traducción de código a descripción:

Propiedad	Significado
<i>DataSource</i>	La fuente de datos original. Es la que se modifica.
<i>DataField</i>	El campo original. Es el campo que contiene la referencia.
<i>ListSource</i>	La fuente de datos a la que se hace referencia.
<i>KeyField</i>	El campo al que se hace referencia.
<i>ListField</i>	Los campos que se muestran como descripción.

Cuando se arrastra un campo de búsqueda desde el Editor de Campos hasta la superficie de un formulario, el componente creado es de tipo *TDBLookupComboBox*. En este caso especial, solamente hace falta dar el nombre del campo de búsqueda en la propiedad *DataField* del combo o la rejilla, pues el resto del algoritmo de búsqueda es deducible a partir de las propiedades del campo base.

Los combos de búsquedas funcionan con el estilo equivalente al de un *TDBComboBox* ordenado y con el estilo *csDropDownList*. Esto quiere decir que no se pueden introducir valores que no se encuentren en la tabla de referencia. Pero también significa que podemos utilizar las búsquedas incrementales por teclado. Y esto es también válido para los componentes *TDBLookupListBox*.

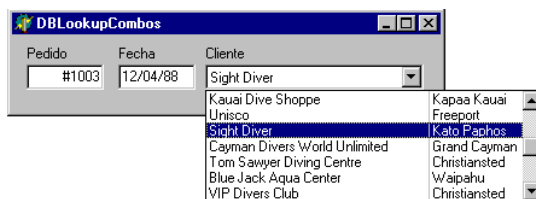
Más adelante, cuando estudiemos la comunicación entre el BDE y las bases de datos cliente/servidor, veremos que la búsqueda incremental en combos de búsqueda es una característica *muy* peligrosa. La forma en que el BDE implementa por omisión las búsquedas incrementales insensibles a mayúsculas es un despilfarro. ¿Una solución? Utilice ventanas emergentes para la selección de valores, implementando usted mismo el mecanismo de búsqueda incremental, o diseñe su propio combo de búsqueda. Recuerde que esta advertencia solamente vale para bases de datos cliente/servidor.

Tanto para el combo como para las listas pueden especificarse varios campos en *ListField*; en este caso, los nombres de campos se deben separar por puntos y comas:

```
DBLookupListBox1.ListField := 'Nombre;Apellidos';
```

Si son varios los campos a visualizar, en el cuadro de lista de *TDBLookupListBox*, y en la lista desplegable de *TDBLookupComboBox* se muestran en columnas separadas. En el caso del combo, en el cuadro de edición solamente se muestra uno de los campos: aquel cuya posición está indicada por la propiedad *ListFieldIndex*. Como por omisión esta propiedad vale 0, inicialmente se muestra el primer campo de la lista. *ListFieldIndex* determina, en cualquier caso, cuál de los campos se utiliza para realizar la búsqueda incremental en el control.





El combo de la figura anterior ha sido modificado de forma tal que su lista desplegable tenga el ancho suficiente para mostrar las dos columnas especificadas en *ListField*. La propiedad *DropDownWidth*, que por omisión vale 0, indica el ancho en píxeles de la lista desplegable. Por otra parte, *DropDownRows* almacena el número de filas que se despliegan a la vez, y *DropDownAlign* es la alineación a utilizar.

## Esencia y apariencia

Con mucha frecuencia me hacen la siguiente pregunta: ¿cómo puedo inicializar un combo de búsqueda para que no aparezca nada en él (o para que aparezca determinado valor)? Desde mi punto de vista, esta pregunta revela el mismo problema que la siguiente: ¿cómo muevo una imagen al control *TDBImage*?

Hay un defecto de razonamiento en las dos preguntas anteriores. El programador *ve* el combo, y quiere cambiar los datos *en el combo*. Pero pierde de vista que el combo es una *manifestación* del campo subyacente. Hay que buscar la esencia, no la apariencia. Así que cuando desee eliminar el contenido de un *TDBLookupComboBox* lo que debe hacer es asignar el valor nulo al campo asociado.

Lo mismo sucederá, como veremos al final del capítulo, cuando deseemos almacenar una imagen en un *TDBImage*: sencillamente cargaremos esta imagen en el campo de la tabla o consulta. Y no crea que estoy hablando de errores de programación infrecuentes: he visto programas que para mover una imagen a un campo la copian primero en el Portapapeles, fuerzan a un *TDBImage* a que pegue los datos y luego utilizan *Post* para hacer permanente la modificación. Es evidente que el contenido del Portapapeles no debe ser modificado a espaldas del usuario, por lo que se trata de una técnica pésima. Más adelante veremos cómo utilizar campos *blob* para este propósito.

## Casillas de verificación y grupos de botones

Si un campo admite solamente dos valores, como en el caso de los campos lógicos, es posible utilizar para su edición el componente *TDBCkCheckBox*. Este componente es muy sencillo, y su modo de trabajo está determinado por el tipo del campo asociado. Si el campo es de tipo *TBooleanField*, el componente traduce el valor *True* como casilla marcada, y *False* como casilla sin marcar. En ciertos casos, conviene utilizar la propiedad *AllowGrayed*, que permite que la casilla tenga tres estados; el tercer estado, la casilla en color gris, se asocia con un valor nulo en el campo.

Si el campo no es de tipo lógico, las propiedades *ValueChecked* y *ValueUnchecked* determinan las cadenas equivalentes a la casilla marcada y sin marcar. En estas propiedades se pueden almacenar varias cadenas separadas por puntos y comas. De este modo, el componente puede aceptar varias versiones de lo que la tabla considera valores “marcados” y “no marcados”:

```
DBCheckBox1.ValueChecked := 'Sí;Yes;Oui';
DBCheckBox1.ValueUnchecked := 'No';
```

Por su parte, *TDBRadioGroup* es una versión orientada a bases de datos del componente estándar *TRadioGroup*. Puede utilizarse como alternativa al cuadro de lista con contenido fijo cuando los valores que podemos utilizar son pocos y se pueden mostrar en pantalla todos a la vez. La propiedad *Items*, de tipo *TStrings*, determina los textos que se muestran en pantalla. Si está asignada la propiedad *Values*, indica qué cadenas almacenaremos en el campo para cada opción. Si no hemos especificado algo en esta propiedad, se almacenan las mismas cadenas asignadas en *Items*.

## Imágenes extraídas de bases de datos

En un campo BLOB de una tabla se pueden almacenar imágenes en los más diversos formatos. Si el formato de estas imágenes es el de mapas de bits o de metaarchivos, podemos utilizar el componente *TDBImage*, que se basa en el control *TImage*, para visualizar y editar estos datos. Las siguientes propiedades determinan la apariencia de la imagen dentro del control:

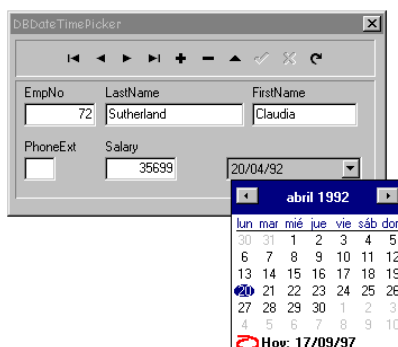
Propiedad	Significado
<i>Center</i>	La imagen aparece centrada o en la esquina superior izquierda
<i>Stretch</i>	Cuando es verdadera, la imagen se expande o contrae para adaptarse al área del control.
<i>QuickDraw</i>	Si es verdadera, se utiliza la paleta global. El dibujo es más rápido, pero puede perderse calidad en ciertas resoluciones.

De forma similar a lo que ocurre con los campos de tipo memo, el componente *TDBImage* ofrece una propiedad *AutoDisplay* para acelerar la exploración de tablas que contienen imágenes. Cuando *AutoDisplay* está activo, cada vez que el conjunto de datos cambia su fila activa, cambia el contenido del control. Si es *False*, hay que pulsar INTRO sobre el control para visualizar el contenido del campo, o llamar al método *LoadPicture*.

Estos componentes permiten trabajar con el Portapapeles, utilizando las teclas estándar CTRL+C, CTRL+V y CTRL+X, o mediante los métodos *CopyToClipboard*, *CutToClipboard* y *PasteFromClipboard*.

## La técnica del componente del pobre

Delphi nos ofrece un componente *TDateTimePicker*, que permite la edición y visualización de fechas como en un calendario. Este control se encuentra en la página *Win32* de la Paleta de Componentes. No existe ningún control similar para editar del mismo modo campos de bases de datos que contengan fechas ¿Qué hacemos en estos casos, si no existe el componente en el mercado<sup>19</sup> y no tenemos tiempo y paciencia para crear un nuevo componente? La solución consiste en utilizar los eventos del componente *TDataSource* para convertir a la pobre Cenicienta en una hermosa princesa, capaz de comunicarse con todo tipo de bases de datos:



Los eventos del componente *TDataSource* son los tres siguientes:

Evento	Se dispara...
<i>OnStateChange</i>	Cada vez que cambia el estado de la tabla asociada
<i>OnDataChange</i>	Cuando cambia el contenido del registro actual
<i>OnUpdateData</i>	Cuando hay que guardar los datos locales en el registro activo

Para ilustrar el uso del evento *OnStateChange*, cree la típica aplicación de prueba para tablas; ya sabe a qué me refiero:

- Una tabla (*TTable*) que podamos modificar sin remordimientos.
- Una fuente de datos (*TDataSource*) conectada a la tabla anterior.
- Una rejilla de datos (*TDBGrid*), para visualizar los datos. Asigne la fuente de datos a su propiedad *DataSource*.
- Una barra de navegación (*TDBNavigator*), para facilitarnos la navegación y edición. Modifique también su propiedad *DataSource*.

Vaya entonces a *DataSource1* y cree la siguiente respuesta al evento *OnStateChange*:

<sup>19</sup> En realidad existen versiones de *TDBDateTimePicker* de todo tipo y color.

```

procedure TForm1.DataSource1StateChange(Sender: TObject);
const
    Ventana: TForm = nil;
    Lista: TListBox = nil;
begin
    if not Assigned(Ventana) then
        begin
            Ventana := TForm.Create(Self);
            Lista := TListBox.Create(Ventana);
            Lista.Align := alClient;
            Lista.Parent := Ventana;
            Ventana.Show;
        end;
    // Es necesario incluir la unidad TypInfo
    Lista.Items.Add(
        GetEnumName(TypeInfo(TDatasetState), Ord(Table1.State)));
end;

```

¿Y qué hay de la promesa de visualizar campos de fechas en un *TDateTimePicker*? Es fácil: cree otra ventana típica, con la tabla, la fuente de datos, una barra de navegación y unos cuantos *TDBEdit*. Utilice la tabla *employee.db* del alias *bedemos*, por ejemplo. Esta tabla tiene un campo, *HireDate*, que almacena la fecha de contratación del empleado. Sustituya el cuadro de edición de este campo por un componente *TDateTimePicker*. Finalmente seleccione la fuente de datos y cree un manejador para el evento *OnDataChange*:

```

procedure TForm1.DataSource1DataChange(Sender: TObject;
    Field: TField);
begin
    DateTimePicker1.DateTime := Table1['HireDate'];
end;

```

Antes he mencionado que *OnDataChange* se dispara cada vez que cambia el contenido del registro activo. Esto sucede, por ejemplo, cuando navegamos por la tabla, pero también cuando alguien (un control visual o un fragmento de código) cambia el valor de un campo. En el primer caso el parámetro *Field* del evento contiene *Null*, mientras que en el segundo apunta al campo que ha sido modificado. Le propongo que intente optimizar el método anterior haciendo uso de esta característica del evento.

## Permitiendo las modificaciones

Con un poco de cuidado, podemos permitir también las modificaciones sobre el calendario. Necesitamos interceptar ahora el evento *OnUpdateData* de la fuente de datos:

```

procedure TForm1.DataSource1UpdateData(Sender: TObject);
begin
    Table1['HireDate'] := DateTimePicker1.DateTime;
end;

```

Pero aún falta algo para que la maquinaria funcione. Y es que hemos insistido antes en que solamente se puede asignar un valor a un campo si la tabla está en estado de

edición o inserción, mientras que aquí realizamos la asignación directamente. La respuesta es que necesitamos interceptar también el evento que anuncia los cambios en el calendario; cuando se modifique la fecha queremos que el control ponga en modo de edición a la tabla de forma automática. Esto se realiza mediante el siguiente método:

```
procedure TForm1.DateTimePicker1Change(Sender: TObject);
begin
    DataSource1.Edit;
end;
```

En vez de llamar al método *Edit* de la tabla o consulta asociada al control, he llamado al método del mismo nombre de la fuente de datos. Este método verifica primero si la propiedad *AutoEdit* del *data source* está activa, y si el conjunto de datos está en modo de exploración, antes de ponerlo en modo de edición.

De todos modos, tenemos un pequeño problema de activación recursiva. Cuando cambia el contenido del calendario, estamos poniendo a la tabla en modo de edición, lo cual dispara a su vez al evento *OnDataChange*, que relea el contenido del registro activo, y perdemos la modificación. Por otra parte, cada vez que cambia la fila activa, se dispara *OnDataChange*, que realiza una asignación a la fecha del calendario. Esta asignación provoca un cambio en el componente y dispara a *OnChange*, que pone entonces a la tabla en modo de edición. Esto quiere decir que tenemos que controlar que un evento no se dispare estando activo el otro. Para ello utilizaré una variable privada en el formulario, que actuará como si fuera un semáforo:

```
type
    TForm1 = class (TForm)
        // ...
    private
        FCambiando: Boolean;
    end;

procedure TForm1.DataSource1DataChange(Sender: TObject;
    Field: TField);
begin
    if not FCambiando then
        try
            FCambiando := True;
            DateTimePicker1.DateTime := Table1['HireDate'];
        finally
            FCambiando := False;
        end;
end;

procedure TForm1.Calendar1Change(Sender: TObject);
begin
    if not FCambiando then
        try
            FCambiando := True;
            DataSource1.Edit;
        end;
```

```

    finally
        FCambiando := False;
    end;
end;

```

Finalmente, podemos interceptar también el evento *OnExit* del calendario, de modo que las modificaciones realizadas en el control se notifiquen a la tabla cuando abandonemos el calendario. Esto puede ser útil si tenemos más de un control asociado a un mismo campo, o si estamos visualizando a la vez la tabla en una rejilla de datos. La clave para esto la tiene el método *UpdateRecord* del conjunto de datos:

```

procedure TForm1.DateTimePicker1Exit(Sender: TObject);
begin
    Table1.UpdateRecord;
end;

```

## Blob, blob, blob...

Las siglas BLOB quieren decir en inglés, *Binary Large Objects*: Objetos Binarios Grandes. Con este nombre se conoce un conjunto de tipos de datos con las siguientes características:

- Longitud variable.
- Esta longitud puede ser bastante grande. En particular, en la arquitectura Intel puede superar la temible “barrera” de los 64 KB.
- En el caso general, el sistema de bases de datos no tiene por qué saber interpretar el contenido del campo.

La forma más fácil de trabajar con campos blob es leer y guardar el contenido del campo en ficheros. Se pueden utilizar para este propósito los métodos *LoadFromFile* y *SaveToFile*, de la clase *TBlobField*:

```

procedure TBlobField.LoadFromFile(const Fichero: string);
procedure TBlobField.SaveToFile(const Fichero: string);

```

El ejemplo típico de utilización de estos métodos es la creación de un formulario para la carga de gráficos en una tabla, si los gráficos residen en un fichero. Supongamos que se tiene una tabla, *imagenes*, con dos campos: *Descripcion*, de tipo cadena, y *Foto*, de tipo gráfico. En un formulario colocamos los siguientes componentes:

<i>tbImagenes</i>	La tabla de imágenes.
<i>tbImagenesFoto</i>	El campo correspondiente a la columna <i>Foto</i> .
<i>OpenDialog1</i>	Cuadro de apertura de ficheros, con un filtro adecuado para cargar ficheros gráficos.
<i>bnCargar</i>	Al pulsar este botón, se debe cargar una nueva imagen en el registro actual.

He mencionado solamente los componentes protagonistas; es conveniente añadir un *DBEdit* para visualizar el campo *Descripcion* y un *DBImage*, para la columna *Foto*; por supuesto, necesitaremos una fuente de datos. También será útil incluir una barra de navegación.

El código que se ejecuta en respuesta a la pulsación del botón de carga de imágenes debe ser:

```
procedure TForm1.bnCargarClick(Sender: TObject);
begin
    if OpenFileDialog1.Execute then begin
        if not (tbImagenes.State in dsEditModes) then
            tbImagenes.Edit;
        tbImagenesFoto.LoadFromFile(OpenFileDialog1.FileName);
    end;
end;
```

Observe que este método no depende de la presencia del control *TDBImage* para la visualización del campo. Me he adelantado en el uso del método *Edit*; sin esta llamada, la asignación de valores a campos provoca una excepción.

Otra posibilidad consiste en utilizar los métodos *LoadFromStream* y *SaveToStream* para transferir el contenido completo del campo hacia o desde un flujo de datos, en particular, un flujo de datos residente en memoria. Este tipo de datos, en la VCL, se representa mediante la clase *TMemoryStream*.

## La clase *TBlobStream*

No hace falta, sin embargo, que el contenido del campo blob se lea completamente en memoria para poder trabajar con él. Para esto contamos con la clase *TBlobStream*, que nos permite acceder al contenido del campo como si estuviéramos trabajando con un fichero. Para crear un objeto de esta clase, hay que utilizar el siguiente constructor:

```
constructor TBlobStream.TBlobStream(Campo: TBlobField;
    Modo: TBlobStreamMode);
```

El tipo *TBlobStreamMode*, por su parte, es un tipo enumerativo que permite los valores *bmRead*, *bmWrite* y *bmReadWrite*; el uso de cada uno de estos modos es evidente.

¿Quiere almacenar imágenes en formato JPEG en una base de datos? Desgraciadamente, el componente *TDBImage* no permite la visualización de este formato, que permite la compresión de imágenes. Pero no hay problemas, pues podemos almacenar estas imágenes en un campo blob sin interpretación, y encargarnos nosotros mismos de su captura y visualización. La captura de la imagen tendrá lugar mediante un procedimiento idéntico al que mostramos en la sección anterior para imágenes “normales”:

```

procedure TForm1.bnCargarClick(TObject *Sender: TObject);
begin
    if OpenFileDialog1.Execute then
    begin
        if not (tbImagenes.State in dsEditModes) then
            tbImagenes.Edit;
            tbImagenesFoto.LoadFromFile(OpenDialog1.FileName);
        end;
    end;

```

La única diferencia es que asumimos que el campo *Foto* de la tabla *tbImagenes* debe ser un campo blob sin interpretación, y que hemos configurado al componente *OpenDialog1* para que sólo nos permita abrir ficheros de extensión *jpeg* y *jpg*. Debemos incluir además el fichero de cabecera *jpeg.hpp* en el formulario.

Para visualizar las imágenes, traemos un componente *TImage* de la página *Additional* de la Paleta de Componentes, e interceptamos el evento *OnDataChange* de la fuente de datos asociada a *tbImagenes*:

```

procedure TForm1.DataSource1DataChange(Sender: TObject;
    Field: TField);
var
    BS: TBlobStream;
    G: TJPEGImage;
begin
    if tbImagenesFoto.IsNull then
        Image1.Picture.Graphic := nil
    else begin
        BS := TBlobStream.Create(tbImagenesFoto, bmRead);
        try
            G := TJPEGImage.Create;
            try
                G.LoadFromStream(BS);
                Image1.Picture.Graphic := G;
            finally
                G.Free;
            end;
        finally
            BS.Free;
        end;
    end;
end;

```

La clase *TJPEGImage* está definida en la unidad *JPEG*, y descende del tipo *TGraphic*. El método anterior se limita a crear un objeto de esta clase y llenarlo con el contenido del campo utilizando un objeto *TBlobStream* como paso intermedio. Finalmente, el gráfico se asigna al control de imágenes.



## Rejillas y barras de navegación

QUIZÁS LOS CONTROLES DE BASES DE DATOS MÁS POPULARES entre los programadores de Windows sean las rejillas y las barras de navegación. Las rejillas de datos nos permiten visualizar de una forma cómoda y general cualquier conjunto de datos. Muchas veces se utiliza una rejilla como punto de partida para realizar el mantenimiento de tablas. Desde la rejilla se pueden realizar búsquedas, modificaciones, inserciones y borrados. Las respuestas a consultas *ad hoc* realizadas por el usuario pueden también visualizarse en rejillas. Por otra parte, las barras de navegación son un útil auxiliar para la navegación sobre conjuntos de datos, estén representados sobre rejillas o sobre cualquier otro conjunto de controles. En este capítulo estudiaremos este par de componentes, sus propiedades y eventos básicos, y la forma de personalizarlos.

### El uso y abuso de las rejillas

Sin embargo, es fácil utilizar incorrectamente las rejillas de datos. Pongamos por caso que una aplicación deba manejar una tabla de clientes de 1.000.000 de registros. El programador medio coloca una rejilla y ¡hala, a navegar! Si la aplicación está basada en tablas de Paradox o dBase no hay muchos problemas. Pero si tratamos con una base de datos SQL es casi seguro que se nos ahogue la red. Está claro que mostrar 25 filas en pantalla simultáneamente es más costoso que mostrar, por ejemplo, sólo un registro a la vez. Además, es peligroso dejar en manos de un usuario desaprensivo la posibilidad de moverse libremente a lo largo y ancho de un conjunto de datos. En dependencia de la interfaz de acceso a datos que utilicemos, si el usuario intenta ir al último registro del conjunto resultado, veremos cómo la red se pone literalmente al rojo vivo, mientras va trayendo al cliente cada uno de los 999.998 registros intermedios. Esto cobra importancia cuando el cliente se conecta a su servidor a través de Internet, pues el ancho de banda es mucho menor.

Quiero aclarar un par de puntos antes de continuar:

- 1 *No siempre es posible limitar el tamaño de un conjunto resultado.* Al menos, si pretendemos que la limitación tenga sentido. Existen técnicas que estudiaremos más adelante, como los filtros y rangos, que permiten reducir el número de filas de una tabla. Y está claro que utilizando una cláusula **where** en una consulta podemos

lograr el mismo efecto. Pero esta reducción de tamaño no siempre es suficiente. Por ejemplo, ¿cómo limitamos la vista de clientes? ¿Por la inicial del apellido? Vale, hay 26 letras, con lo que obtendremos una media de 40.000 registros. ¿Por ciudades? Tampoco nos sirve. ¿Limitamos el resultado a los 1000 primeros registros? Tendremos que acostumbrarnos a este último tipo de restricciones...

- 2** *Puede ser imprescindible mostrar varias filas a la vez.* Es frecuente oír el siguiente argumento: ¿para qué utilizar una rejilla sobre 1.000.000 de registros, si sólo vamos a poder ver una pequeña ventana de 25 filas a la vez? ¿Es posible sacar algo en claro de una rejilla, que no podamos averiguar navegando registro por registro? Yo creo que sí. Muchas veces olvidamos que la navegación con rejillas cobra especial importancia cuando ordenamos la tabla subyacente por alguna de sus columnas. En tal caso, el análisis del contexto en que se encuentra determinado registro puede aportarnos bastante información. Nos puede ayudar a detectar errores ortográficos en un nombre, que cierto empleado se encuentra en una banda salarial especial, etc.

Un error que comete la mayoría de los programadores consiste en sobrecargar una rejilla con más columnas de lo debido. Esto es un fallo de diseño, pues una rejilla en la que hay que realizar desplazamientos horizontales para ver todas las columnas es poco menos que inútil. Busque, por ejemplo, la libreta de direcciones de su aplicación de correo electrónico. Lo más probable es que los nombres y apellidos de los destinatarios de correo aparezcan en una lista o rejilla, y que el resto de sus datos puedan leerse de uno en uno, en otros controles de edición. Bien, ese es el modelo que le propongo de uso de rejillas. Casi siempre, las columnas que muestro en una rejilla corresponden a la clave primaria o a una clave alternativa. Es posible también que incluya alguna otra columna de la cual quiera obtener información contextual: esto implica con toda seguridad que la rejilla estará ordenada de acuerdo al valor de esa columna adicional. El resto de los campos los sitúo en controles de acceso a datos orientados a campos: cuadros de edición, combos, imágenes, etc.

## **El funcionamiento básico de una rejilla de datos**

Para que una rejilla de datos “funcione”, basta con asignarle una fuente de datos a su propiedad *DataSource*. Es todo. Quizás por causa de la sencillez de uso de estos componentes, hay muchos detalles del uso y programación de rejillas de datos que el desarrollador normalmente pasa por alto, o descuida explicar en su documentación para usuarios. Uno de estos descuidos es asumir que el usuario conoce todos los detalles de la interfaz de teclado y ratón de este control. Y es que esta interfaz es rica y compleja.

Las teclas de movimiento son las de uso más evidente. Las flechas nos permiten movernos una fila o un carácter a la vez, podemos utilizar el avance y retroceso de página; las tabulaciones nos llevan de columna en columna, y es posible usar la tabulación inversa.

La tecla INS pone la tabla asociada a la rejilla en modo de inserción. Aparentemente, se crea un nuevo registro con los valores por omisión, y el usuario debe llenar el mismo. Para grabar el nuevo registro tenemos que movernos a otra fila. Por supuesto, si tenemos una barra de navegación asociada a la tabla, el botón *Post* produce el mismo efecto sin necesidad de cambiar la fila activa. Un poco más adelante estudiaremos las barras de navegación.

Pulsando F2, el usuario pone a la rejilla en modo de edición; Delphi crea automáticamente un cuadro de edición del tamaño de la celda activa para poder modificar el contenido de ésta. Esta acción también se logra automáticamente cuando el usuario comienza a teclear sobre una celda. La *edición automática* se controla desde la propiedad *AutoEdit* de la fuente de datos (*data source*) a la cual se conecta la rejilla. Para grabar los cambios realizados hacemos lo mismo que con las inserciones: pulsamos el botón *Post* de una barra de navegación asociada o nos cambiamos de fila.

Otra combinación útil es CTRL+SUPR, mediante la cual se puede borrar el registro activo en la rejilla. Cuando hacemos esto, se nos pide una confirmación. Es posible suprimir este mensaje, que es lanzado por la rejilla, y pedir una confirmación personalizada para cada tabla interceptando el evento *BeforeDelete* de la propia tabla. Esto se explicará en el capítulo 22.

La rejilla de datos tiene una columna fija, en su extremo izquierdo, que no se mueve de lugar aún cuando nos desplazamos a columnas que se encuentran fuera del área de visualización. En esta columna, la fila activa aparece marcada, y la marca depende del estado en que se encuentre la tabla base. En el estado *dsBrowse*, la marca es una punta de flecha; cuando estamos en modo de edición, una viga I (*i-beam*), la forma del cursor del ratón cuando se sitúa sobre un cuadro de edición; en modo de inserción, la marca es un asterisco. Como veremos, esta columna puede ocultarse manipulando las opciones de la rejilla.

Por otra parte, con el ratón podemos cambiar en tiempo de ejecución la disposición de las columnas de una rejilla, manipulando la barra de títulos. Por ejemplo, arrastrando una cabecera de columna se cambia el orden de las columnas; arrastrando la división entre columnas, se cambia el tamaño de las mismas. A partir de Delphi 3 pueden incluso utilizarse las cabeceras de columnas como botones. Naturalmente, la acción realizada en respuesta a esta acción debe ser especificada por el usuario interceptando un evento.

## Opciones de rejillas

Muchas de las características visuales y funcionales de las rejillas pueden cambiarse mediante la propiedad *Options*. Aunque las rejillas de datos, *TDBGrid* y las rejillas *TDrawGrid* y *TStringGrid* están relacionadas entre sí, las opciones de estas clases son diferentes. He aquí las opciones de las rejillas de datos y sus valores por omisión:

Opción	PO	Significado
<i>dgEditing</i>	Sí	Permite la edición de datos sobre la rejilla
<i>dgAlwaysShowEditor</i>	No	Activa siempre el editor de celdas
<i>dgTitles</i>	Sí	Muestra los títulos de las columnas
<i>dgIndicator</i>	Sí	La primera columna muestra el estado de la tabla
<i>dgColumnResize</i>	Sí	Cambiar el tamaño y posición de las columnas
<i>dgColLines</i>	Sí	Dibuja líneas entre las columnas
<i>dgRowLines</i>	Sí	Dibuja líneas entre las filas
<i>dgTabs</i>	Sí	Utilizar tabulaciones para moverse entre columnas
<i>dgRowSelect</i>	No	Seleccionar filas completas, en vez de celdas
<i>dgAlwaysShowSelection</i>	No	Dejar siempre visible la selección
<i>dgConfirmDelete</i>	Sí	Permite confirmar los borrados
<i>dgCancelOnExit</i>	Sí	Cancela inserciones vacías al perder el foco
<i>dgMultiSelect</i>	No	Permite seleccionar varias filas a la vez.

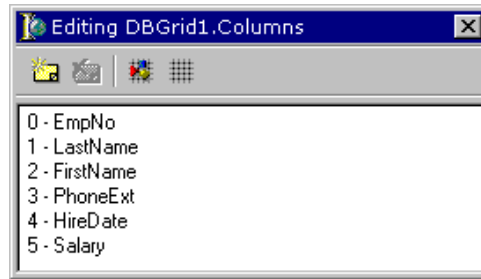
Muchas veces, es conveniente cambiar las opciones de una rejilla en coordinación con otras opciones o propiedades. Por ejemplo, cuando queremos que una rejilla se utilice sólo en modo de lectura, además de cambiar la propiedad *ReadOnly* es aconsejable eliminar la opción *dgEditing*. De este modo, cuando el usuario seleccione una celda, no se creará el editor sobre la celda y no se llevará la impresión de que la rejilla iba a permitir la modificación. Como ejemplo adicional, cuando preparamos una rejilla para seleccionar múltiples filas con la opción *dgMultiSelect*, es bueno activar también la opción *dgRowSelect*, para que la barra de selección se dibuje sobre toda la fila, en vez de sobre celdas individuales.

## Columnas a la medida

La configuración de una rejilla de datos va más allá de las posibilidades de la propiedad *Options*. Por ejemplo, es necesario indicar el orden inicial de las columnas, los títulos, la alineación de los textos dentro de las columnas... En la primer versión de la VCL, las tareas mencionadas se llevaban a cabo modificando propiedades de los componentes de campos de la tabla o consulta asociada a la rejilla. Si queríamos cambiar el título de una columna, debíamos modificar la propiedad *DisplayLabel* del campo correspondiente. La alineación de la columna se extraía de la propiedad *Alignment* del campo. Y para ocultar una columna, debíamos utilizar la propiedad *Visible* del componente de campo.

Esto ocasionaba bastantes problemas; el problema más grave era lo limitado de las posibilidades de configuración según este estilo. Por ejemplo, aunque un campo se alineara a la derecha, el título de su columna se alineaba siempre a la izquierda. En la siguiente versión las cosas hubieran podido agravarse, por causa de la aparición de los módulos de datos, que permiten utilizar el mismo componente no visual de acceso con diferentes modos de visualización. Al colocar una tabla determinada en un módulo de datos y configurar las propiedades visuales de un componente de campo en dicho módulo, cualquier rejilla que se conectara a la tabla mostraría la misma apa-

riencia. Para poder separar la parte visual de los métodos de acceso, se hizo indispensable la posibilidad de configurar directamente las rejillas de datos.



La propiedad *Columns* permite modificar el diseño de las columnas de una rejilla de datos. El tipo de esta propiedad es *TDBGridColumns*, y es una colección de objetos de tipo *TColumn*. Para editar esta propiedad podemos hacer un doble clic en el valor de la propiedad en el Inspector de Objetos, o realizar el doble clic directamente sobre la propia rejilla.

La propiedades que nos interesan de las columnas son:

Propiedad	Significado
<i>Alignment</i>	Alineación de la columna
<i>ButtonStyle</i>	Permite desplegar una lista desde una celda, o mostrar un botón de edición.
<i>Color</i>	Color de fondo de la columna.
<i>DropDownRows</i>	Número de filas desplegables.
<i>Expanded</i>	Se utiliza en los campos compuestos de Oracle.
<i>FieldName</i>	El nombre del campo que se visualiza.
<i>Font</i>	El tipo de letra de la columna.
<i>PickList</i>	Lista opcional de valores a desplegar.
<i>ReadOnly</i>	Desactiva la edición en la columna.
<i>Width</i>	El ancho de la columna, en píxeles.
<i>Title.Alignment</i>	Alineación del título de la columna.
<i>Title.Caption</i>	Texto de la cabecera de columna.
<i>Title.Color</i>	Color de fondo del título de columna.
<i>Title.Font</i>	Tipo de letra del título de la columna.

La propiedad *Expanded* se aplica a las columnas que representan campos de objetos de Oracle. Si *Expanded* es *True*, la columna se divide en subcolumnas, para representar los atributos del objeto, y la fila de títulos de la rejilla duplica su ancho, para mostrar tanto el nombre de la columna principal como los nombres de las dependientes. Esta propiedad puede modificarse tanto en tiempo de diseño como en ejecución.

Si el programador no especifica columnas en tiempo de diseño, éstas se crean en tiempo de ejecución y se llenan a partir de los valores extraídos de los campos de la

tabla; observe que algunas propiedades de las columnas se corresponden a propiedades de los componentes de campo. Si existen columnas definidas en tiempo de diseño, son éstas las que se utilizan para el formato de la rejilla.

En la mayoría de las situaciones, las columnas se configuran en tiempo de diseño, pero es también posible modificar propiedades de columnas en tiempo de ejecución. El siguiente método muestra como se pueden mostrar de forma automática en color azul las columnas de una rejilla que pertenezcan a los campos que forman parte del índice activo.

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to DBGrid1.Columns.Count - 1 do
    with DBGrid1.Columns[I] do
      if Field.IsIndexField then
        Font.Color := clBlue;
    end;
  end;
```

En este otro ejemplo tenemos una rejilla con dos columnas, que ocupa todo el espacio interior de un formulario. Queremos que la segunda columna de la rejilla ocupe todo el área que deja libre la primera columna, aún cuando cambie el tamaño de la ventana. Se puede utilizar la siguiente instrucción:

```
procedure TForm1.FormResize(Sender: TObject);
begin
  DBGrid1.Columns[1].Width := DBGrid1.ClientWidth -
    DBGrid1.Columns[0].Width - IndicatorWidth - 2;
end;
```

El valor que se resta en la fórmula, *IndicatorWidth*, corresponde a una variable global declarada en la unidad *DBGrids*, y corresponde al ancho de la columna de indicadores. He restado dos píxeles para tener en cuenta las líneas de separación. Si la rejilla cambia sus opciones de visualización, cambiará el valor a restar, por supuesto.

Para saber qué columna está activa en una rejilla, utilizamos la propiedad *SelectedIndex*, que nos dice su posición. *SelectedField* nos da acceso al componente de campo asociado a la columna activa. Por otra parte, si lo que queremos es la lista de campos de una rejilla, podemos utilizar la propiedad vectorial *Fields* y la propiedad entera *FieldCount*. Un objeto de tipo *TColumn* tiene también, en tiempo de ejecución, una propiedad *Field* para trabajar directamente con el campo asociado.

## Guardar y restaurar los anchos de columnas

Para los usuarios de nuestras aplicaciones puede ser conveniente poder mantener el formato de una rejilla de una sesión a otra, especialmente los anchos de las columnas. Voy a mostrar una forma sencilla de lograrlo, suponiendo que cada columna de la rejilla pueda identificarse de forma única por el nombre de su campo asociado. El

ejemplo utiliza ficheros de configuración, pero puede adaptarse fácilmente para hacer uso del registro de Windows.

Supongamos que el formulario *Form1* tiene una rejilla *DBGrid1* en su interior. Entonces necesitamos la siguiente respuesta al evento *OnCreate* del formulario para restaurar los anchos de columnas de la sesión anterior:

```
resourcestring
    SClaveApp = 'Software\MiEmpresa\MiAplicacion\';

procedure TForm1.FormCreate(Sender: TObject);
var
    I: Integer;
begin
    with TRegIniFile.Create(
        SClaveApp + 'Rejillas\' + Name + '.' + DBGrid1.Name) do
        try
            for I := 0 to DBGrid1.Columns.Count - 1 do
                with DBGrid1.Columns[I] do
                    Width := ReadInteger('Width', FieldName, Width);
                finally
                    Free;
                end;
            end;
        end;
```

Estamos almacenando los datos de la rejilla *DBGrid1* en la siguiente clave del registro de Windows:

```
[HKEY_CURRENT_USER\MiEmpresa\MiAplicacion\Rejillas\Form1.DBGrid1]
```

El tercer parámetro de *ReadInteger* es el valor que se debe devolver si no se encuentra la clave dentro de la sección. Este valor se utiliza la primera vez que se ejecuta el programa, cuando aún no existe el fichero de configuraciones. Este fichero se debe actualizar cada vez que se termina la sesión, durante el evento *OnClose* del formulario:

```
procedure TForm1.FormClose(Sender: TObject;
    var Action: TCloseAction);
var
    I: Integer;
begin
    with TRegIniFile.Create(
        SClaveApp + 'Rejillas\' + Name + '.' + DBGrid1.Name) do
        try
            for I := 0 to DBGrid1.Columns.Count - 1 do
                with DBGrid1.Columns[I] do
                    WriteInteger('Width', FieldName, Width);
                finally
                    Free;
                end;
            end;
        end;
```

Sobre la base de estos procedimientos simples, el lector puede incorporar mejoras, como la lectura de la configuración por secciones completas, y el almacenamiento de más información, como el orden de las columnas.

## Listas desplegables y botones de edición

Las rejillas de datos permiten que una columna despliegue una lista de valores para que el usuario seleccione uno de ellos. Puede suceder en dos contextos diferentes:

- Si el campo asociado a una columna es un campo de búsqueda (*lookup field*).
- Si el programador especifica una lista de valores en la propiedad *PickList* de una columna.

En el primer caso, el usuario puede elegir un valor de una lista que se extrae de otra tabla. El estilo de interacción es similar al que utilizan los controles *TDBLookupComboBox*, que estudiaremos más adelante; no se permite, en contraste, la búsqueda incremental mediante teclado. En el segundo caso, la lista que se despliega contiene exactamente los valores tecleados por el programador en la propiedad *PickList* de la columna. Esto es útil en situaciones en las que los valores más frecuentes de una columna se reducen a un conjunto pequeño de posibilidades: formas de pagos, fórmulas de tratamiento (Señor, Señora, Señorita), y ese tipo de cosas. En cualquiera de estos casos, la altura de la lista desplegable se determina por la propiedad *DropDownRows*: el número máximo de filas a desplegar.

PartNo	VendorNo	Vendor	Description	Cost	ListPrice
900	3820	Techniques	Dive kayak	\$1,356.75	\$3,999.95
912	3820	J.W. Luchter Mfg.	Underwater Diver Vehicle	\$504.00	\$1,680.00
1313	3511	Scuba Professionals	Regulator System	\$117.50	\$250.00
1314	5641	Techniques	Second Stage Regulator	\$124.10	\$365.00
1316	3511	Perry Scuba	Regulator System	\$119.35	\$341.00
1320	3511	Beauchat, Inc.	Second Stage Regulator	\$73.53	\$171.00
1328	3511	Scuba Professionals	Regulator System	\$154.80	\$430.00
1330	3511	Scuba Professionals	Alternate Inflation Regulator	\$85.80	\$260.00

La propiedad *ButtonStyle* determina si se utiliza el mecanismo de lista desplegable o no. Si vale *bsAuto*, la lista se despliega si se cumple alguna de las condiciones anteriores. Si la propiedad vale *bsNone*, no se despliega nunca la lista. Esto puede ser útil en ocasiones: suponga que hemos definido, en la tabla que contiene las líneas de detalles de un pedido, el precio del artículo que se vende como un campo de búsqueda, que partiendo del código almacenado en la línea de detalles, extrae el precio de venta de la tabla de artículos. En este ejemplo, no nos interesa que se despliegue la lista con todos los precios existentes, y debemos hacer que *ButtonStyle* valga *bsNone* para esa columna.

Por otra parte, si asignamos el valor *bsEllipsis* a la propiedad *ButtonStyle* de alguna columna, cuando ponemos alguna celda de la misma en modo de edición aparece en el extremo derecho de su interior un pequeño botón con tres puntos suspensivos. Lo único que hace este botón es lanzar el evento *OnEditButtonClick* cuando es pulsado:

```
procedure TForm1.DBGrid1EditButtonClick(Sender: TObject);
```



La columna en la cual se produjo la pulsación es la columna activa de la rejilla, que se puede identificar por su posición, *SelectedIndex*, o por el campo asociado, *SelectedField*. Este botón también puede activarse pulsando CTRL+INTRO.

Los puntos suspensivos aparecen también con los campos *TReferenceField* y *TDataSetField*, de Oracle 8 y MyBase. Cuando se pulsa el botón, aparece una rejilla emergente con los valores anidados dentro del campo. También se puede ejecutar el método *ShowPopupEditor* para mostrar la rejilla.

## Números verdes y números rojos

La propiedad *Columns* nos permite especificar un color para cada columna por separado. ¿Qué sucede si deseamos, por el contrario, colores diferentes por fila, o incluso por celdas? Y puestos a pedir, ¿se pueden dibujar gráficos en una rejilla? Claro que sí: para eso existe el evento *OnDrawColumnCell*.

Comencemos por algo sencillo: en la tabla de inventario *parts.db* queremos mostrar en color rojo aquellos artículos para los cuales hay más pedidos que existencias; la tabla en cuestión tiene sendas columnas, *OnOrder* y *OnHand*, para almacenar estas cantidades. Así que creamos un manejador para *OnDrawColumnCell*, y Delphi nos presenta el siguiente esqueleto de método:

```
procedure TForm1.DBGrid1DrawColumnCell(Sender: TObject;
    const Rect: TRect; DataCol: Integer; Column: TColumn;
    State: TGridDrawState);
begin
    end;
```

*Rect* es el área de la celda a dibujar, *DataCol* es el índice de la columna a la cual pertenece y *Column* es el objeto correspondiente; por su parte, *State* indica si la celda está seleccionada, enfocada y si es una de las celdas de cabecera. En ninguna parte se nos dice la fila que se va a dibujar, pero la tabla asociada a la rejilla tendrá activo el registro correspondiente durante la respuesta al evento. Así que podemos empezar por cambiar las condiciones de dibujo: si el valor del campo *OnOrder* iguala o supera al valor del campo *OnHand* en la fila activa de la tabla, cambiamos el color del tipo de letras seleccionado en el lienzo de la rejilla a rojo. Después, para dibujar el texto ... un momento, ¿no estamos complicando un poco las cosas?

La clave para evitar este dolor de cabeza es el método *DefaultDrawColumnCell*, perteneciente a las rejillas de datos. Este método realiza el dibujo por omisión de las celdas, y puede ser llamado en el interior de la respuesta a *OnDrawColumnCell*. Los parámetros de este método son exactamente los mismos que se suministran con el evento; de este modo, ni siquiera hay que consultar la ayuda en línea para llamar al método. Si el manejador del evento se limita a invocar a este método, el dibujo de la rejilla sigue siendo idéntico al original. Podemos entonces limitarnos a cambiar las condiciones iniciales de dibujo, realizando asignaciones a las propiedades del lienzo de la rejilla, antes de llamar a esta rutina. He aquí el resultado:

```

procedure TForm1.DBGrid1DrawColumnCell(Sender: TObject;
  const Rect: TRect; DataCol: Integer; Column: TColumn;
  State: TGridDrawState);
begin
  if Parts['OnOrder'] >= Parts['OnHand'] then
    TDBGrid(Sender).Canvas.Font.Color := clRed;
    TDBGrid(Sender).DefaultDrawColumnCell(
      Rect, DataCol, Column, State);
end;

```

Por supuesto, también podemos dibujar el contenido de una celda sin necesidad de recurrir al dibujo por omisión. Si estamos visualizando la tabla de empleados en una rejilla, podemos añadir desde el Editor de Columnas una nueva columna, con el botón *New*, dejando vacía la propiedad *FieldName*. Esta columna se dibujará en blanco. Añadimos también al formulario un par de componentes de imágenes, *TImage*, con los nombres *CaraAlegre* y *CaraTriste*, y mapas de bits que hagan juego; estos componentes deben tener la propiedad *Visible* a *False*. Finalmente, interceptamos el evento de dibujo de celdas de columnas:

```

procedure TForm1.DBGrid1DrawColumnCell(Sender: TObject;
  const Rect: TRect; DataCol: Integer; Column: TColumn;
  State: TGridDrawState);
begin
  with Sender as TDBGrid do
    if Column.FieldName <> '' then
      DefaultDrawColumnCell(Rect, DataCol, Column, State)
    else if Empleados['Salary'] >= 45000 then
      Canvas.StretchDraw(Rect, CaraAlegre.Picture.Graphic)
    else
      Canvas.StretchDraw(Rect, CaraTriste.Picture.Graphic);
end;

```

Dibujando gráficos en una rejilla

EmpNo	LastName	FirstName	PhoneExt	HireDate	Salary	
2	Nelson	Roberto	250	28/12/88	40000	☺
4	Young	Bruce	233	28/12/88	55500	☺
5	Lambert	Kim	22	6/02/89	25000	☹
8	Johnson	Leslie	410	5/04/89	25050	☹
9	Forest	Phil	229	17/04/89	25050	☹
11	Weston	K. J.	34	17/01/90	33292.9375	☹
12	Lee	Terri	256	1/05/90	45332	☹
14	Hall	Stewart	227	4/06/90	34482.625	☹
15	Young	Katherine	231	14/06/90	24400	☹
20	Papadopoulos	Chris	887	1/01/90	25050	☹
24	Fisher	Pete	888	12/09/90	23040	☹
28	Bennet	Ann	5	1/02/91	34482.8	☹

Otro ejemplo, que quizás sea más práctico, es mostrar el valor de un campo lógico dentro de una rejilla como una casilla de verificación. Supongamos que la tabla *Table1* contiene un campo de nombre *Activo*, de tipo lógico. Para dibujar la columna correspondiente al campo en la rejilla, utilizamos el siguiente método en respuesta al evento *OnDrawColumnCell*:

```

procedure TForm1.DBGrid1DrawColumnCell(Sender: TObject;
  const Rect: TRect; DataCol: Integer; Column: TColumn;
  State: TGridDrawState);
var

```

```

    Check: Integer;
begin
  if CompareText(Column.FieldName, 'ACTIVO') = 0 then
    begin
      Check := 0;
      if Table1['ACTIVO'] then
        Check := DFCS_CHECKED;
      DBGrid1.Canvas.FillRect(Rect);
      DrawFrameControl(DBGrid1.Canvas.Handle, Rect,
        DFC_BUTTON, DFCS_BUTTONCHECK or Check);
    end
  else
    DBGrid1.DefaultDrawColumnCell(Rect, DataCol, Column, State);
  end;
end;

```

*DrawFrameControl* es una función del API de Windows que nos permite dibujar muchos de los controles nativos. Recuerde que mientras más complicado sea el dibujo, más tardará la rejilla en dibujarse.

## Más eventos de rejillas

Delphi 3 añadió un par de nuevos eventos a las rejillas de datos: *OnCellClick*, que se dispara cuando el usuario pulsa el ratón sobre una celda de datos, y *OnTitleClick*, cuando la pulsación ocurre en alguno de los títulos de columnas. Aunque estos eventos pueden detectarse teóricamente directamente con los eventos de ratón, *OnCellClick* y *OnTitleClick* nos facilitan las cosas al pasar, como parámetro del evento, el puntero al objeto de columna donde se produjo la pulsación.

Para demostrar el uso de estos eventos, coloque en un formulario vacío una tabla con las siguientes propiedades:

Propiedad	Valor
<i>DatabaseName</i>	<i>IBLOCAL</i>
<i>TableName</i>	<i>EMPLOYEE</i>
<i>Active</i>	<i>True</i>

Es importante para este ejemplo que la tabla pertenezca a una base de datos SQL; es por eso que utilizamos los ejemplos de InterBase. Coloque en el formulario, además, un *TDataSource* y una rejilla de datos, debidamente conectados.

Luego, cree la siguiente respuesta al evento *OnTitleClick* de la rejilla:

```

procedure TForm1.DBGrid1TitleClick(Column: TColumn);
begin
  try
    if Column.Field.FieldKind = fkLookup then
      Table1.IndexFieldNames := Column.Field.KeyFields
    else
      Table1.IndexFieldNames := Column.FieldName;
    except end;
  end;
end;

```

La propiedad *IndexFieldNames* de las tablas se utiliza para indicar por qué campo, o combinación de campos, queremos que la tabla esté ordenada. Para una tabla SQL este campo puede ser arbitrario, cosa que no ocurre para las tablas locales; en el capítulo sobre índices trataremos este asunto. Nuestra aplicación, por lo tanto, permite cambiar el criterio de ordenación de la tabla que se muestra con sólo pulsar con el ratón sobre el título de la columna por la cual se quiere ordenar.

## La barra de desplazamiento de la rejilla

La otra gran diferencia entre las rejillas de datos de Delphi 1 y las de versiones posteriores consiste en el comportamiento de la barra de desplazamiento vertical que tienen asociadas. En las versiones 1 y 2 de la VCL, para desesperación de muchos programadores habituados a trabajar con bases de datos locales, la barra solamente asume tres posiciones: al principio, en el centro y al final. ¿Por qué? La culpa la tienen las tablas SQL: para saber cuántas filas tiene una tabla residente en un servidor remoto necesitamos cargar todas las filas en el ordenador cliente. ¿Y todo esto sólo para que el cursor de la barra de desplazamiento aparezca en una posición proporcional? No merece la pena, y pagan justos por pecadores, pues también se utiliza el mismo mecanismo para las tablas de Paradox y dBase.

Afortunadamente, Delphi corrigió esta situación para las tablas locales en formato Paradox aunque, por supuesto, las cosas siguen funcionando igual para el resto de los formatos.

## Rejillas de selección múltiple

Como hemos visto, si activamos la opción *dgMultiSelect* de una rejilla, podemos seleccionar varias filas simultáneamente sobre la misma. La selección múltiple se logra de dos formas diferentes:

- Extendiendo la selección mediante las flechas hacia arriba y hacia abajo, manteniendo pulsada la tecla de mayúsculas.
- Pulsando con el ratón sobre una fila, mientras se sostiene la tecla CTRL.

Si pulsamos CTRL+SUPR mientras la rejilla tiene el foco del teclado, eliminaremos todas las filas seleccionadas. Cualquier otra operación que deseemos deberá ser programada.

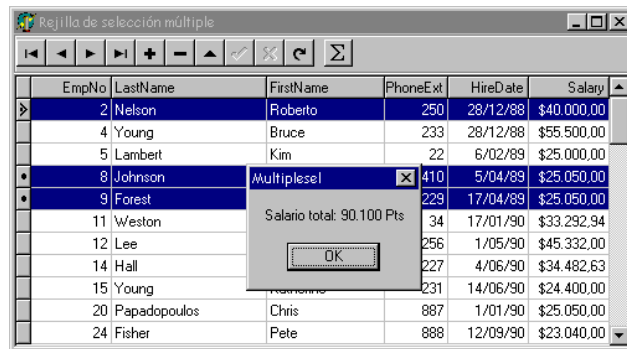
La clave para programar con una rejilla de selección múltiple es la propiedad *SelectedRows*. Esta propiedad es del tipo *TBookmarkList*, una lista de marcas de posición, cuyos principales métodos y propiedades son los siguientes:

Propiedades/Métodos	Propósito
<i>Count</i>	Cantidad de filas seleccionadas.
<i>Items[Index: Integer]</i>	Lista de posiciones seleccionadas.

Propiedades/Métodos	Propósito
<i>CurrentRowSelected</i>	¿Está seleccionada la fila activa?
<b>procedure</b> <i>Clear</i> ;	Eliminar la selección.
<b>procedure</b> <i>Delete</i> ;	Borrar las filas seleccionadas.
<b>procedure</b> <i>Refresh</i> ;	Actualizar la selección, eliminando filas borradas.

El siguiente método muestra cómo sumar los salarios de los empleados seleccionados en una rejilla con selección múltiple:

```
procedure TForm1.bnSumarClick(Sender: TObject);
var
    Total: Currency;
    BM: TBookmarkStr;
    I: Integer;
begin
    Total := 0;
    tbEmpleados.DisableControls;
    BM := tbEmpleados.Bookmark;
    try
        for I := 0 to DBGrid1.SelectedRows.Count - 1 do
            begin
                tbEmpleados.Bookmark := DBGrid1.SelectedRows[I];
                Total := Total + tbEmpleados['Salary'];
            end;
        finally
            tbEmpleados.Bookmark := BM;
            tbEmpleados.EnableControls;
        end;
    ShowMessage(Format('Salario total: %m', [Total]));
end;
```



La técnica básica consiste en controlar el recorrido desde un bucle **for**, e ir activando sucesivamente cada fila seleccionada en la lista de marcas. Para que la tabla se mueva su cursor al registro marcado en la rejilla, solamente necesitamos asignar la marca a la propiedad *Bookmark* de la tabla.

## Barras de navegación

Al igual que sucede con las rejillas de datos, la principal propiedad de las barras de navegación es *DataSource*, la fuente de datos controlada por la barra. Cuando esta propiedad está asignada, cada uno de los botones de la barra realiza una acción sobre el conjunto de datos asociado a la fuente de datos: navegación (*First*, *Prior*, *Next*, *Last*), inserción (*Insert*), eliminación (*Delete*), modificación (*Edit*), confirmación o cancelación (*Post*, *Cancel*) y actualización de los datos en pantalla (*Refresh*).



Un hecho curioso: muchos programadores me preguntan en los cursos que imparto si se pueden modificar las imágenes de los botones de la barra de navegación. Claro que se puede, respondo, pero siempre me quedo intrigado, pues no logro imaginar un conjunto de iconos más “expresivo” o “adecuado”. ¿Acaso flechas *art nouveau* verdes sobre fondo rojo? De todos modos, para el que le interese este asunto, las imágenes de los botones están definidas en el fichero *dbctrls.res*, que se encuentra en el subdirectorio *lib* de Delphi. Se puede cargar este fichero con cualquier editor gráfico de recursos, Image Editor incluido, atentar contra la estética y el buen gusto.

También he encontrado programadores que sustituyen completamente la barra de navegación por botones de aceleración. Esta técnica es correcta, y es fácil implementar tanto las respuestas a las pulsaciones de los botones como mantener el estado de activación de los mismos; hemos visto cómo se hace esto último al estudiar los eventos del componente *TDataSource* en el capítulo anterior. Sin embargo, existe un motivo interesante para que se utilice la barra de navegación de Delphi, al menos sus cuatro primeros botones, y no tiene que ver con el hecho de que ya esté programada. ¿Se ha fijado lo que sucede cuando se deja pulsado uno de los botones de navegación durante cierto tiempo? El comando asociado se repite entonces periódicamente. Implementar este comportamiento desde cero ya es bastante más complejo, y no merece la pena.

## Había una vez un usuario torpe, muy torpe...

...tan torpe que no acertaba nunca en el botón de la barra de navegación que debía llevarlo a la última fila de la tabla. No señor: este personaje siempre “acertaba” en el botón siguiente, el que inserta registros. Por supuesto, sus tablas abundaban en filas vacías... y la culpa, según él, era del programador. Que nadie piense que lo estoy inventando, es un caso real.

Para estas situaciones tenemos la propiedad *VisibleButtons* de la barra de navegación. Esta propiedad es la clásica propiedad cuyo valor es un conjunto. Podemos, por ejemplo, esconder todos los botones de actualización de una barra, dejando solamente los cuatro primeros botones. Esconder botones de una barra provoca un de-

sagradable efecto secundario: disminuye el número de botones pero el ancho total del componente permanece igual, lo que conduce a que el ancho individual de cada botón aumente. Claro, podemos corregir la situación posteriormente reduciendo el ancho general de la barra.

A propósito de botones, las barras de navegación tienen una propiedad *Flat*, para que el borde tridimensional de los botones esté oculto hasta que el ratón pase por encima de uno de ellos. La moda ejerce su dictadura también en las pantallas de nuestros ordenadores.

## Ayudas para navegar

Aunque la barra de navegación tiene una propiedad *Hint* como casi todos los controles, esta propiedad no es utilizada por Delphi. Las indicaciones por omisión que muestran los botones de una barra de navegación se encuentran definidas en la unidad *dbconsts.pas*, utilizando declaraciones de cadenas de recursos (**resourcestring**).

Para personalizar las indicaciones de ayuda, es necesario utilizar la propiedad *Hints*, en plural, que permite especificar una indicación por separado para cada botón. *Hints* es de tipo *TStrings*, una lista de cadenas. La primera cadena corresponde al primer botón, la segunda cadena, que se edita en la segunda línea del editor de propiedades, corresponde al segundo botón, y así sucesivamente. Esta correspondencia se mantiene incluso cuando hay botones no visibles. Por supuesto, las ayudas asociadas a los botones ausentes no tendrán efecto alguno.

## El comportamiento de la barra de navegación

Una vez modificada la apariencia de la barra, podemos también modificar parcialmente el comportamiento de la misma. Para esto contamos con el evento *OnClick* de este componente:

```
procedure TForm1.DBNavigator1Click(Sender: TObject;
    Button: TNavigateButton);
begin
    end;
```

Podemos ver que, a diferencia de la mayoría de los componentes, este evento *OnClick* tiene un parámetro adicional que nos indica qué botón de la barra ha sido pulsado. Este evento se dispara *después* de que se haya efectuado la acción asociada al botón; si se ha pulsado el botón de editar, el conjunto de datos asociado ya ha sido puesto en modo de edición. Esto se ajusta al concepto básico de tratamiento de eventos: un evento es un contrato sin obligaciones. No importa si no realizamos acción alguna en respuesta a un evento en particular, pues el mundo seguirá girando sin nuestra cooperación.

Mostraré ahora una aplicación de este evento. Según mi gusto personal, evito en lo posible que el usuario realice altas y modificaciones directamente sobre una rejilla. Prefiero, en cambio, que estas modificaciones se efectúen sobre un cuadro de diálogo modal, con los típicos botones de aceptar y cancelar. Este diálogo de edición debe poderse ejecutar desde la ventana en la que se efectúa la visualización mediante la rejilla de datos. Si nos ceñimos a este estilo de interacción, no nos vale el comportamiento normal de las barras de navegación. Supongamos que *Form2* es el cuadro de diálogo que contiene los controles necesarios para la edición de los datos visualizados en el formulario *Form1*. Podemos entonces definir la siguiente respuesta al evento *OnClick* de la barra de navegación existente en *Form1*:

```
procedure TForm1.DBNavigator1Click(Sender: TObject;
  Button: TNavigateButton);
begin
  if Button in [nbEdit, nbInsert] then
    // La tabla está ya en modo de edición o inserción
    Form2.ShowModal;
end;
```

De este modo, al pulsar el botón de edición o el de inserción, se pone a la tabla base en el estado correspondiente y se activa el diálogo de edición. Hemos supuesto que este diálogo tiene ya programadas acciones asociadas a los botones para grabar o cancelar los cambios cuando se cierra. Podemos incluso crear métodos de clase, como los que mostraremos en el capítulo 23, para crear dinámicamente este cuadro de diálogo:

```
procedure TForm1.DBNavigator1Click(Sender:TObject;
  Button: TNavigateButton);
begin
  if Button in [nbEdit, nbInsert] then
    // La tabla está ya en modo de edición o inserción
    TForm2.Mostrar(Button = nbEdit); // Creación dinámica
end;
```

Un método útil es el siguiente:

```
procedure TDBNavigator.BtnClick(Index: TNavigateBtn);
```

Este método simula la pulsación del botón indicado de la barra de navegación. Supongamos que, en el ejemplo anterior, quisiéramos que una doble pulsación del ratón sobre la rejilla de datos activase el diálogo de edición para modificar los datos de la fila actual. En vez de programar a partir de cero esta respuesta, lo más sensato es aprovechar el comportamiento definido para la barra de navegación. Interceptamos de la siguiente forma el evento *OnDblClick* de la rejilla de datos:

```
procedure TForm1.DBGrid1DblClick(Sender: TObject);
begin
  DBNavigator1.BtnClick(nbEdit);
end;
```



Observe que la llamada al método *BtnClick* va a disparar también el evento asociado a *OnClick* de la barra de navegación.

El evento *BeforeAction* es disparado cuando se pulsa un botón, pero antes de que se produzca la acción asociada al mismo. El prototipo del evento es similar al de *OnClick*. A veces yo utilizo este evento para cambiar la acción asociada al botón de inserción. Este botón *inserta* visualmente una fila entre la fila actual y la anterior, pero en muchos casos es más interesante que la fila vaya al final de la rejilla, directamente. Es decir, quiero que el botón ejecute el método *Append*, no *Insert*. Bueno, ésta es una forma de lograrlo:

```
procedure TForm1.DBNavigator1.BeforeAction(Sender: TObject);
begin
    TDBNavigator(Sender).DataSource.DataSet.Append;
    SysUtils.Abort;
end;
```

## Rejillas de controles

Un *TDBGrid* de Delphi no puede editar, al menos directamente, un campo lógico como si fuera una casilla de verificación. Es igualmente cierto que, mediante los eventos *OnDrawColumnCell*, *OnCellClick* y una gran dosis de buena voluntad, podemos simular este comportamiento. Pero también podemos utilizar el componente *TDBCtrlGrid*, que nos permite, mediante la copia de controles de datos individuales, mostrar varios registros a la vez en pantalla.

En principio, un *TDBCtrlGrid* aparece dividido en paneles, y uno de estos paneles acepta otros controles en su interior. En tiempo de ejecución, los otros paneles, que aparecen inactivos durante el diseño, cobran vida y repiten en su interior controles similares a los colocados en el panel de diseño. En cada uno de estos paneles, por supuesto, se muestran los datos correspondientes a un registro diferente. Las propiedades que tienen que ver con la apariencia y disposición de los paneles son:

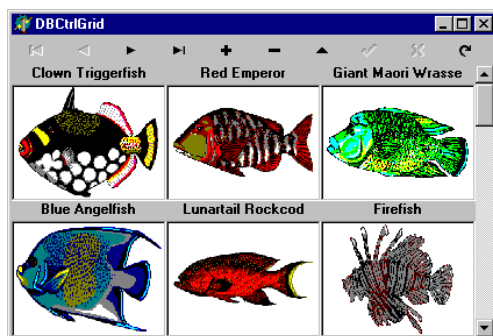
Propiedad	Significado
<i>Orientation</i>	Orientación del desplazamiento de paneles.
<i>ColCount</i>	Número de columnas.
<i>RowCount</i>	Número de filas.
<i>PanelBorder</i>	¿Tiene borde el panel? <sup>20</sup>
<i>PanelWidth</i>	Ancho de cada panel, en píxeles.
<i>PanelHeight</i>	Altura de cada panel, en píxeles.

Las rejillas de controles tienen la propiedad *DataSource*, que indica la fuente de datos a la cual están conectados los diferentes controles de su interior. Cuando un control de datos se coloca en este tipo de rejilla debe “renunciar” a tener una fuente de datos diferente a la del *TDBCtrlGrid*. De este modo pueden incluirse cuadros de edición,

<sup>20</sup> ¿Y sueñan los androides con ovejas eléctricas?

combos, casillas de verificación, memos, imágenes o cualquier control que nos dicte nuestra imaginación. La excepción son los *TDBRadioGroup* y los *TDBRichEdit*.

En las primeras versiones de Delphi no se podían colocar directamente componentes *DBMemo* y *DBImage* en una rejilla de controles. La causa inmediata era que estos controles carecían de la opción *csReplicable* dentro de su conjunto de opciones *ControlStyle*. La causa verdadera era que el BDE no permitía el uso de caché para campos BLOB. No obstante, se pueden crear componentes derivados que añadan la opción de replicación durante la creación.



La implementación de componentes duplicables en la VCL es algo complicada y no está completamente documentada. Aunque el tema sale fuera del alcance de este libro, mencionaré que estos componentes deben responder al mensaje interno *CM\_GETDATALINK*, y tener en cuenta el valor de la propiedad *ControlState* para dibujar el control, pues si el estado *csPaintCopy* está activo, hay que leer directamente el valor a visualizar desde el campo.

## Indices, filtros y búsqueda

**M**YBASE ES UN SISTEMA PEQUEÑO, PERO muy bien diseñado. Hasta el momento sólo hemos arañado la superficie de todas las posibilidades que ofrece. En este capítulo agruparé varias técnicas de trabajo con conjuntos de datos clientes que van más allá de la simple navegación: el uso de índices en memoria, los métodos de búsqueda y varias técnicas de trabajo con filtros.

Algunos de estos recursos son específicos del componente *TClientDataSet*, pero otros se encuentran también disponibles en otras interfaces de acceso, por lo que su interés sobrepasa al simple estudio de MyBase.

### Indices en memoria

Aunque los conjuntos de datos clientes sean tablas almacenadas en memoria, sigue siendo útil poder disponer de índices sobre sus filas. El objetivo principal no es acelerar la búsqueda de valores, naturalmente, sino ordenar las filas de acuerdo a diferentes criterios. Existen tres formas de crear índices en *TClientDataSet*:

- Utilizando la propiedad *IndexDefs*, antes de crear el conjunto de datos.
- Dinámicamente, cuando ya está creado el conjunto de datos, utilizando el método *AddIndex*. Estos índices sobreviven mientras el conjunto de datos está activo, pero no se almacenan en disco.
- Dinámicamente, especificando un criterio de ordenación mediante la propiedad *IndexFieldNames*. Estos índices desaparecen al cambiar el criterio de ordenación.

Cuando utilizamos alguna de las dos primeras técnicas, podemos utilizar opciones que solamente están disponibles para este tipo de índices:

- Para claves compuestas, algunos campos pueden ordenarse ascendentemente y otros en forma descendente.
- Para claves compuestas, algunos campos pueden distinguir entre mayúsculas y minúsculas, y otros no.
- Puede indicarse un nivel de agrupamiento. Esto nos permitirá definir, más adelante, estadísticas a nivel de grupos.

Veamos primero el prototipo del método *AddIndex*:

```
procedure TCustomClientDataSet.AddIndex(
  const Name, Fields: string;
  Options: TIndexOptions;
  const DescFields: string = '';
  const CaseInsFields: string = '';
  const GroupingLevel: Integer = 0);
```

La declaración de *TIndexOptions* es la siguiente:

```
type
  TIndexOption = (
    ixPrimary, ixUnique, ixDescending,
    ixCaseInsensitive, ixExpression, ixNonMaintained);

  TIndexOptions = set of TIndexOption;
```

Los nombres de los campos que formarán parte del índice se pasan en el parámetro *Fields*, separados entre sí por puntos y comas. Si incluimos *ixDescending* en las opciones, se utilizará el orden descendente para todos los campos. Ahora bien, podemos pasar en *DescFields* un subconjunto de los campos, para que solamente ellos sean ordenados a la inversa. La misma relación existe entre la opción *ixCaseInsensitive* y el parámetro *CaseInsFields*, respecto a ignorar las mayúsculas en los campos alfanuméricos. El siguiente índice ordena ascendentemente por el nombre, teniendo en cuenta las mayúsculas y minúsculas, de modo ascendente por el apellido, pero sin distinguir los tipos de letra, y de modo descendente por el salario:

```
ClientDataSet1.AddIndex('Indice1', 'Nombre;Apellidos;Salario',
  [], 'Salario', 'Apellidos');
```

Una cosa es tener un índice disponible, y otra es activarlo, para que las filas del conjunto de datos aparezcan ordenadas. Ya he presentado a *IndexFieldNames*; hay otra propiedad para activar un índice, llamada *IndexName*, en la que debemos asignar el nombre de un índice existente. *IndexName* e *IndexFieldNames* son propiedades mutuamente excluyentes. Si hay algo asignado en una de ellas y se modifica el valor de la otra, se pierde el valor de la primera.

#### NOTA

Cuando se trabaja con Paradox y dBase, utilizando el BDE, la activación de un índice tiene mucha importancia, porque permite el uso de algunas operaciones especiales, como la búsqueda con los métodos *FindKey* y *FindNearest*, o que especifiquemos rangos de filas. Pero cuando se trabaja con MyBase, o con conjuntos de datos SQL, existen operaciones alternativas que no requieren la presencia de un índice activo, y que generalmente son más fáciles de utilizar.

Hagamos una prueba muy sencilla: vamos a crear una pequeña aplicación que muestre el contenido de la tabla *customers.xml* en una rejilla. Quiero que cuando pulsemos el ratón sobre una de las cabeceras de columna, el conjunto de datos se ordene por el campo asociado a ella. La primera vez que se pulse, de forma ascendente, pero si se

vuelve a hacer clic en la misma columna, el orden debe ser descendente. Este es el código necesario:

```
procedure TwndPrincipal.DBGrid1TitleClick(Column: TColumn);
begin
  if SameText(Column.FieldName, Clientes.IndexFieldNames) then
  begin
    Clientes.AddIndex(Column.FieldName, Column.FieldName, [],
      Column.FieldName);
    Clientes.IndexName := Column.FieldName;
  end
  else
  begin
    Clientes.IndexFieldNames := Column.FieldName;
  end;
end;
```

Como *IndexFieldNames* no permite especificar la ordenación descendente, hay que crear un índice dinámicamente para este caso. Observe que le damos a estos índices el mismo nombre de la columna. Si quiere, puede ampliar el ejemplo permitiendo criterios de ordenación que involucren a más de una columna.

El uso de *IndexDefs* para crear índices permanentes es igual de sencillo. El tipo *TIndexDef* utiliza las propiedades *DescFields*, *CaseInsFields* y *GroupingLevel* para obtener el mismo efecto que los parámetros correspondientes de *AddIndex*.

## Campos calculados internos

¿Podemos indicar una expresión para crear un índice en MyBase? No, al menos no directamente. Pero hay un truco a nuestro alcance: definir *campos calculados internos*, para crear índices sobre ellos. Estos se diferencian de los campos calculados tradicionales en que sus valores se almacenan internamente junto con los campos normales “de datos”, en el formato interno de registros de MyBase. Así logramos tres objetivos:

- 1 No es necesario recalcular el valor del campo cada vez que cambiemos la fila activa. El cálculo se realiza la primera vez que se selecciona una fila, y sólo es necesario repetirlo si se modifica la fila.
- 2 Como he mencionado, podemos usarlos para definir índices.
- 3 Más adelante veremos que también es posible usarlos para definir *filtros*.

Pondré como ejemplo una variante de la aplicación de la sección anterior, pero utilizando la tabla de empleados. Después de que tenga configurados los campos de datos de la misma, ejecute el comando *New field* en el Editor de Campos. El tipo del nuevo campo debe ser *InternalCalc*. Bautícelo como *Meses*, porque contendrá el número de meses que lleva contratado el empleado, y asigne *Integer* como su tipo de datos:

Luego tendremos que interceptar el evento *OnCalcFields* del conjunto de datos:

```
procedure TwndPrincipal.EmpleadosCalcFields(DataSet: TDataSet);
begin
    if Empleados.State = dsInternalCalc then
        EmpleadosMeses.Value :=
            RestarMeses(Date, EmpleadosHireDate.Value);
end;
```

¡Preste atención al nuevo valor de la propiedad *State*! El evento *OnCalcFields* puede dispararse por dos motivos diferentes: que el estado del conjunto de datos sea *dsCalcFields* o *dsInternalCalc*. Sólo es necesario evaluar los campos calculados internos en el estado *dsInternalCalc*. Si lo desea, puede poner un punto de ruptura dentro del método anterior, o utilizar la función *OutputDebugString* para comprobar que el estado *dsInternalCalc* se activa en menos ocasiones que *dsCalcFields*.

Nº	Apellidos	Nombre	Extensión	Contrato	Salario	Meses
29	De Souza	Roger	288	18/feb/1991	25.500,00 €	132
28	Bennet	Ann	5	01/feb/1991	34.482,00 €	132
24	Fisher	Pete	888	12/sep/1990	23.040,00 €	137
15	Young	Katherine	231	14/jun/1990	24.400,00 €	140
14	Hall	Stewart	227	04/jun/1990	34.482,00 €	141
12	Lee	Terri	256	01/may/1990	45.332,00 €	142
11	Weston	K. J.	34	17/ene/1990	33.292,00 €	145
20	Papadopoulos	Chris	887	01/ene/1990	25.050,00 €	146
9	Forest	Phil	229	17/abr/1989	25.050,00 €	154
8	Johnson	Leslie	410	05/abr/1989	25.050,00 €	155
5	Lambert	Kim	22	06/feb/1989	25.000,00 €	157
4	Young	Bruce	233	28/dic/1988	55.500,00 €	158
2	Nelson	Roberto	250	28/dic/1988	40.000,00 €	158

Por su parte, la función *RestarMeses* hace exactamente lo que su nombre sugiere:

```
function RestarMeses(Date1, Date2: TDateTime): Integer;
var
    Y1, M1, D1, Y2, M2, D2: Word;
begin
    DecodeDate(Date1, Y1, M1, D1);
    DecodeDate(Date2, Y2, M2, D2);
```

```
Result := (Y1 - Y2) * 12 + M2 - M1;
end;
```

Finalmente, recuerde interceptar el evento *OnTitleClick* de la rejilla para ordenar por la columna que seleccionemos, al igual que hicimos en la aplicación anterior.

## Búsquedas

Otros dos métodos importantes de *TClientDataSet* son *Locate* y *Lookup*, y se utilizan para la búsqueda de registros según el valor de la columna que indiquemos. Ambos métodos se introducen en *TDataSet* y están disponibles en todas las clases derivadas de ésta. La implementación, por supuesto, varía según la interfaz de acceso a datos que se utilice. Como los conjuntos de datos clientes almacenan sus datos en memoria, el algoritmo exacto de su implementación no es tan importante. Este es el prototipo de *Locate*:

```
function TDataSet.Locate(const Columnas: string;
    const Valores: Variant; Opciones: TLocateOptions): Boolean;
```

En el primer parámetro se pasa una lista de nombres de columnas; el formato de esta lista es similar al que hemos encontrado en la propiedad *IndexFieldNames*: las columnas se separan entre sí por puntos y comas. Para cada columna especificada hay que suministrar un valor. Si se busca por una sola columna, necesitamos un solo valor, el cual puede pasarse directamente, por ser el segundo parámetro de tipo *Variant*. Si se especifican dos o más columnas, tenemos que pasar una matriz variante; en breve veremos ejemplos de estas situaciones. Por último, el conjunto de opciones del tercer parámetro puede incluir las siguientes:

Opción	Propósito
<i>loCaseInsensitive</i>	Ignorar mayúsculas y minúsculas
<i>loPartialKey</i>	Permitir búsquedas parciales en columnas alfanuméricas

Cuando *Locate* puede encontrar una fila con los valores deseados en las columnas apropiadas, devuelve *True* como resultado, y cambia la fila activa del conjunto de datos. Si, por el contrario, no se localiza una fila con tales características, la función devuelve *False* y no se altera la posición del cursor sobre la tabla.

Ahora veamos un par de ejemplos sencillos de traspaso de parámetros con *Locate*. El uso más elemental de *Locate* es la localización de una fila dado el valor de una de sus columnas. Digamos:

```
if not Clientes.Locate('CustNo', Pedidos['CustNo'], []) then
    ShowMessage('Se nos ha perdido un cliente en el bosque...');
```

En este caso, el valor a buscar ha sido pasado directamente como un valor variante, pero podíamos haber utilizado un entero con el mismo éxito, suponiendo que el campo *Código* es de tipo numérico:

```

if not Clientes.Locate('CustNo', 007, []) then
    ShowMessage('... o se lo ha tragado la tierra');

```

Se puede aprovechar este algoritmo para crear un campo calculado en la tabla de clientes, que diga si el cliente ha realizado alguna compra. Suponiendo que el nuevo campo, *HaComprado*, es de tipo lógico, necesitamos la siguiente respuesta al evento *OnCalcFields* en el conjunto de datos *Clientes*:

```

procedure TForm1.ClientesCalcFields(DataSet: TDataSet);
begin
    Clientes['HaComprado'] :=
        Pedidos.Locate('CustNo', Clientes['CustNo'], []);
end;

```

Consideremos ahora que queremos localizar un empleado, dados el nombre y el apellido. La instrucción necesaria es la siguiente:

```

Empleados.Locate('LastName;FirstName',
    VarArrayOf([Apellido, Nombre]), []);

```

En primer término, hay que mencionar los nombres de ambas columnas en el primer parámetro, separadas por punto y coma. Después, hay que pasar los dos valores correspondientes como una matriz variante; la función *VarArrayOf* es útil para esta última misión. En este ejemplo, las variables *Apellido* y *Nombre* son ambas de tipo **string**, pero pueden pertenecer a tipos diferentes, en el caso más general.

Cuando la búsqueda se realiza con el objetivo de recuperar el valor de otra columna del mismo registro, se puede aprovechar el método *Lookup*:

```

function TDataSet.Lookup(const Columnas: string;
    const Valores: Variant; const ColResultados: string): Variant;

```

*Lookup* realiza primero un *Locate*, utilizando los dos primeros parámetros. Si no se puede encontrar la fila correspondiente, *Lookup* devuelve el valor variante especial *Null*; por supuesto, la fila activa no cambia. Por el contrario, si se localiza la fila adecuada, la función extrae los valores de las columnas especificadas en el tercer parámetro. Si se ha especificado una sola columna, ese valor se devuelve en forma de variante; si se especificaron varias columnas, se devuelve una matriz variante formada a partir de estos valores. A diferencia de *Locate*, en este caso no se cambia la fila activa original de la tabla al terminar la ejecución del método. La siguiente función, por ejemplo, localiza el nombre de un cliente dado su código:

```

function TDataModule1.ObtenerNombre(Codigo: Integer): string;
begin
    Result := VarToStr(Clientes.Lookup('CustNo', Codigo, 'Company'));
end;

```

He utilizado la función *VarToStr* para garantizar la obtención de una cadena de caracteres, aún cuando no se encuentre el código de la compañía; en tal situación,



*Lookup* devuelve el variante *Null*, que es convertido por *VarToStr* en una cadena vacía.

También se puede utilizar *Lookup* eficientemente para localizar un código de empleado dado el nombre y el apellido del mismo:

```
function TDataModule1.ObtenerCodigo(  
    const Apellido, Nombre: string): Integer;  
var  
    V: Variant;  
begin  
    V := Empleados.Lookup('LastName;FirstName',  
        VarArrayOf([Apellido, Nombre]), 'CustNo');  
    if VarIsNull(V) then  
        DatabaseError('Empleado no encontrado');  
    Result := V;  
end;
```

Para variar, he utilizado una excepción para indicar el fallo de la búsqueda; esto equivale a asumir que lo normal es que la función *ObtenerCodigo* deba encontrar el registro del empleado. Note nuevamente el uso de la función *VarArrayOf*, además del uso de *VarIsNull* para controlar la presencia del valor nulo.

Por último, presentaremos la función inversa a la anterior: queremos el nombre completo del empleado dado su código. En este caso, necesitamos especificar dos columnas en el tercer parámetro de la función. He aquí una posible implementación:

```
function TDataModule1.NombreDeEmpleado(Codigo: Integer): string;  
var  
    V: Variant;  
begin  
    V := Empleados.Lookup('CustNo', Codigo, 'FirstName;LastName');  
    if VarIsNull(V) then  
        DatabaseError('Empleado no encontrado');  
    Result := V[0] + ' ' + V[1];  
end;
```

## Filtros

Los filtros nos permiten limitar las filas visibles de un conjunto de datos mediante una condición arbitraria establecida por el programador. Es un recurso disponible desde la clase *TDataSet*, pero cada interfaz de acceso a datos lo implementa, o no, a su manera, con distintas sintaxis para las condiciones y algoritmos de evaluación.

En MyBase, concretamente, los filtros enmascaran las filas que no cumplen la condición que establezca el programador, pero no las elimina de la memoria. Si desactivamos el filtro, volveremos a ver el conjunto original de filas. De este modo, los filtros sustituyen en MyBase algunos de los tipos de consultas que sí permitiría un sistema SQL.

**NOTA IMPORTANTE**

En la gran mayoría de interfaces de acceso a SQL soportadas por Delphi, los filtros se evalúan también en el lado cliente, no en el servidor. Solamente en un caso muy particular de conjunto de datos del BDE, las tablas, y siempre que se cumplan varias condiciones adicionales, el filtro se evalúa en el servidor. De hecho, lo que sucede es que el componente *TTable* intenta generar una consulta **select** utilizando la condición del filtro para una cláusula **where**.

Existen dos formas principales de establecer un filtro: la más general consiste en utilizar el evento *OnFilterRecord*; pero de esta técnica hablaremos más adelante. Lo más sencillo es usar la propiedad *Filter* para establecer la condición del filtro, y activarla mediante otra propiedad, llamada *Filtered*:

Propiedad	Significado
<i>Filter</i>	Contiene una cadena de caracteres con la condición de filtrado
<i>Filtered</i>	De tipo <i>Boolean</i> , indica si el filtro está “activo” o “latente”
<i>FilterOptions</i>	Las posibles opciones son <i>foCaseInsensitive</i> y <i>foNoPartialCompare</i>

No se preocupe demasiado por las opciones de filtrado. Son una herencia del Paleozoico, cuando Paradox y dBase se alimentaban de algas, amebas y otras porquerías, y ni siquiera existían las plantas gimnospermas. En cuanto a *Filtered*, más adelante veremos cómo se puede aprovechar la existencia de filtros incluso cuando esta propiedad está desactivada.

Una agradable sorpresa: las expresiones de filtros de MyBase son razonablemente potentes. Soportan incluso características no permitidas por los restantes tipos de conjuntos de datos de Delphi. Estos, en su mayor parte, se limitan a sencillas comparaciones entre campos, combinadas con los típicos operadores lógicos:

```
Pais = 'Conchinchina'
(Provincia <> '') or (UltimaFactura > '4/07/96')
Salario >= 30000 and Salario <= 100000
Precio = Coste
```

Incluso hay problemas con la última condición en Paradox y dBase, que exigen que uno de los operadores de la comparación sea obligatoriamente una constante.

MyBase, ¡faltaría más!, reconoce todas las condiciones mencionadas. Podemos, además, saber si un campo es nulo o no utilizando la misma sintaxis que en SQL:

```
Direccion is null
```

Se admiten expresiones aritméticas:

```
Descuento * Cantidad * Precio < 100
```

Se han añadido las siguientes funciones de cadenas:

```
upper, lower, substring, trim, trimleft, trimright
```

Y disponemos de estas funciones para trabajar con fechas:

`day, month, year, hour, minute, second, date, time, getdate`

La función *date* extrae la parte de la fecha de un campo de fecha y hora, mientras que *time* aísla la parte de la hora. La función *getdate* devuelve el momento actual. También se permite la aritmética de fechas:

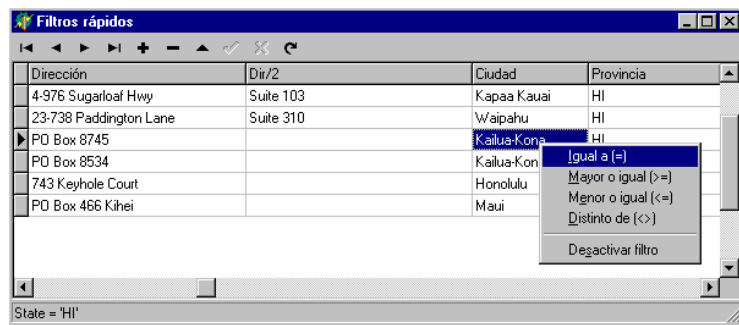
`getdate - HireDate > 365`

Por último, podemos utilizar los operadores **in** y **like** de SQL:

Nombre **like** '%soft'  
year(FechaVenta) **in** (1994,1995)

## Filtros rápidos

Es fácil diseñar un mecanismo general para la aplicación de filtros por el usuario de una rejilla de datos. La clave consiste en restringir el conjunto de datos de acuerdo al valor de la celda seleccionada en la rejilla. Si elegimos, en la columna *Provincia*, una celda con el valor *Madrid*, podemos seleccionar todos los registros cuyo valor para esa columna sea igual o diferente de Madrid. Si está seleccionada la columna *Edad*, en una celda con el valor 30, se puede restringir la tabla a las filas con valores iguales, mayores o menores que este valor. Pero también queremos que estas restricciones sean *acumulativas*. Esto significa que después de limitar la visualización a los clientes de Madrid, podamos entonces seleccionar los clientes con más de 30 años que viven en Madrid. Y necesitamos poder eliminar todas las condiciones de filtrado.



Por lo tanto, comenzamos con la ficha clásica de consulta: una rejilla de datos y una barra de navegación conectada a un *TClientDataSet*; si quiere experimentar, le recomiendo conectar la rejilla a la tabla *customer.xml*, que tiene columnas de varios tipos diferentes. A esta ficha básica le añadimos un menú emergente, *PopupMenu1*, que se conecta a la propiedad *PopupMenu* de la rejilla; basta con esto para que el menú se despliegue al pulsar el botón derecho del ratón sobre la rejilla.

Para el menú desplegable especificamos las siguientes opciones:

Comando de menú	Nombre del objeto de menú
Igual a (=)	<i>miIgual</i>
Distinto de (<>)	<i>miDistinto</i>
Mayor o igual (>=)	<i>miMayorIgual</i>
Menor o igual (<=)	<i>miMenorIgual</i>
Desactivar filtro	<i>miDesactivar</i>

Observe que he utilizado las relaciones mayor o igual y menor igual en lugar de las comparaciones estrictas; la razón es que las condiciones individuales se van a conectar entre sí mediante conjunciones lógicas, el operador *and*, y las comparaciones estrictas pueden lograrse mediante una combinación de las presentes. De todos modos, es algo trivial aumentar el menú y el código correspondiente con estas relaciones estrictas.

Ahora debemos crear un manejador de evento compartido por las cuatro primeras opciones del menú:

```

procedure TwndPrincipal.Filtrar(Sender: TObject);
var
    Operador, Valor, Campo: string;
begin
    if Sender = miIgual then Operador := '='
    else if Sender = miMayorIgual then Operador := '>='
    else if Sender = miMenorIgual then Operador := '<='
    else Operador := '<>';
    // Extraer el nombre del campo y formatear el valor
    Campo := DBGrid1.SelectedField.FieldName;
    Valor := QuotedStr(DBGrid1.SelectedField.AsString);
    // Combinar la nueva condición con las anteriores
    if Clientes.Filter = '' then
        Clientes.Filter := Format('%s %s %s',
            [Campo, Operador, Valor])
    else
        Clientes.Filter := Format('%s AND %s %s %s',
            [Clientes.Filter, Campo, Operador, Valor]);
    // Activar directamente el filtro
    miDesactivar.Enabled := True;
    Clientes.Filtered := True;
end;

```

Y desactivamos el filtro mediante este otro manejador de eventos:

```

procedure TwndPrincipal.miDesactivarClick(Sender: TObject);
begin
    Clientes.Filtered := False;
    Clientes.Filter := '';
    miDesactivar.Enabled := False;
end;

```

## Filtros latentes

Hay que darle al público lo que el público espera. Si un usuario está acostumbrado a actuar de cierta manera frente a cierto programa, esperará la misma implementación

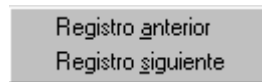
de la técnica en nuestros programas. En este caso, me estoy refiriendo a las técnicas de búsqueda en procesadores de textos. Generalmente, el usuario dispone al menos de un par de comandos: *Buscar* y *Buscar siguiente*; a veces, también hay un *Buscar anterior*. Cuando se ejecuta el comando *Buscar*, el usuario teclea lo que quiere buscar, y este valor es utilizado por las restantes llamadas a *Buscar siguiente* y *Buscar anterior*. Y nuestro problema es que este tipo de interacción es difícil de implementar utilizando el método *Locate*, que solamente nos localiza la primera fila que contiene los valores deseados.

La solución a nuestro problema la tienen, curiosamente, los filtros otra vez. Por lo que sabemos hasta el momento, hace falta activar la propiedad *Filtered* para esconder del conjunto de datos las filas que no cumplen la condición. La novedad consiste en que teniendo *Filtered* el valor *False*, es posible recorrer a saltos los registros que satisfacen la condición del filtro, para lo cual contamos con las funciones *FindFirst*, *FindLast*, *FindNext* y *FindPrior*.

```
function TDataSet.FindFirst: Boolean;
function TDataSet.FindPrior: Boolean;
function TDataSet.FindNext: Boolean;
function TDataSet.FindLast: Boolean;
```

Las cuatro funciones devuelven un valor lógico para indicarnos si la operación fue posible o no. Además, para mayor comodidad, los conjuntos de datos tienen una propiedad *Found*, que almacena el resultado de la última operación sobre filtros.

Para demostrar el uso de este recurso utilizaremos una aplicación muy similar a la de la sección anterior. Pero esta vez utilizaremos un menú emergente diferente:



Si nos situamos sobre una celda de la rejilla y ejecutamos alguno de los comandos anteriores, debe seleccionarse automáticamente el registro anterior o siguiente que tenga en la columna activa el mismo valor que el del registro inicial. Por ejemplo, me sitúo sobre la columna *State* de un registro correspondiente a un hawaiano. Al ejecutar *Registro siguiente* debemos saltar al siguiente hawaiano de la lista. Y si no existe, haremos que el ordenador emita un pitido. Este método debe compartirse por el evento *OnClick* de los dos comandos del menú emergente:

```
procedure TwndPrincipal.Filtrar(Sender: TObject);
var
    BM: TBookmarkStr;
begin
    Clientes.Filter :=
        DBGrid1.SelectedField.FieldName + '=' +
        QuotedStr(DBGrid1.SelectedField.AsString);
    // ----- Inicio de una pequeña chapuza -----
    BM := Clientes.Bookmark;
    Clientes.FindFirst;
```

```

Clientes.Bookmark := BM;
// ----- Fin de la pequeña chapuza -----
if Sender = miSiguiente then
  Clientes.FindNext
else
  Clientes.FindPrior;
if not Clientes.Found then Beep;
end;

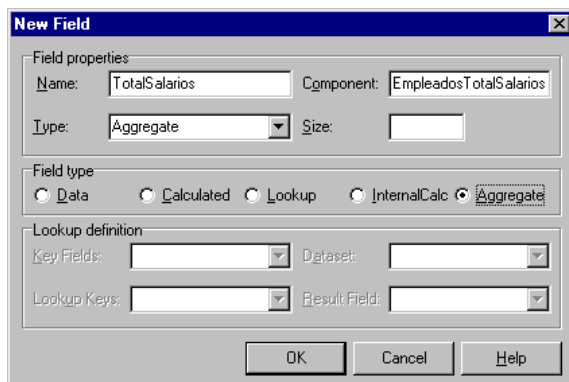
```

La explicación es evidente, excepto la zona que he delimitado como “pequeña chapuza”. Si la desactiva, verá que el mecanismo de navegación a saltos se quedará fijado a la primera condición de filtro que establezcamos. Al llamar a *FindFirst*, sin embargo, parece que se restauran las condiciones iniciales de este tipo de navegación.

## Campos de estadísticas

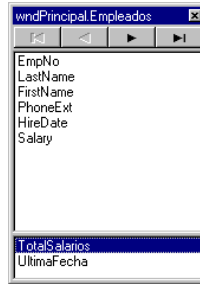
Una de las posibilidades más sorprendentes de los conjuntos de datos clientes es el mantenimiento de estadísticas sobre los valores de campos, a petición del programador. A grandes rasgos, podemos definir un campo utilizando la opción *fkAggregate* en su propiedad *FieldKind*. La clase correspondiente, *TAggregateField*, tiene una propiedad *Expression* en la que debe indicar una expresión SQL que haga uso de las tradicionales funciones de conjuntos: **sum**, **avg**, **min**, **max** y **count**. Entonces el *TClientDataSet* se ocupará de mantener actualizado el valor del campo, de forma automática, incluso cuando realicemos modificaciones sobre las filas del conjunto de datos.

Ahora conviene un ejemplo para ver los detalles: inicie una aplicación, para mostrar en una rejilla el contenido de la tabla *employee.xml*. Configure los campos del componente como es habitual. Luego, dentro del Editor de Campos, pulse el botón derecho del ratón y ejecute el comando *New field*, como si fuese a añadir un campo calculado:



Llame *TotalSalarios* al nuevo campo, y en el panel *Field type* seleccione *Aggregate*. Observará que el tipo del campo pasa a ser inmediatamente *Aggregate*; es difícil para Delphi deducir el tipo de datos exacto de la expresión que teclearemos más adelante. Una vez que haya terminado con este primer campo, añada otro más del mismo tipo,

y llámelo *UltimaFecha*. Como detalle folklórico, observe que Delphi añade un nuevo panel en el Editor de Campos para agrupar los nuevos campos:



Seleccione la propiedad *Expression* de cada uno de ellos, y configúrelos para que evalúen las siguientes expresiones:

```
sum(salary)
max(hiredate)
```

Se trata de expresiones sencillas, pero debe saber que MyBase admite operaciones dentro y fuera de las funciones de conjuntos, como la siguiente:

```
max(salary) - min(salary)
```

Pero no basta indicar la expresión: tenemos también que activar el cálculo de los campos. Primero hay que asignar *True* en la propiedad *Active* de cada uno de los campos, y después tenemos que seleccionar el propio conjunto de datos que los cobija para activar su propiedad *AggregatesActive*. También debemos asignar *True* en la propiedad *Visible* de los dos campos. Si no hacemos esto último, Delphi no incluirá su nombre en la lista desplegable de la propiedad *DataField* común a los controles de datos orientados a campos, como *TDBEdit*.

Ya tenemos los campos, y es el momento de mostrar su contenido... en algún lugar. Digo esto porque no tiene sentido alguno incluir estos campos en la rejilla. De hecho, aunque pongamos *Visible* a *True*, Delphi no permite elegir un campo agregado en el editor de la propiedad *FieldName* de una columna de una rejilla. Sin embargo, si tecleamos el nombre del campo directamente, no protestará. Pruébelo, añadiendo una nueva columna a la rejilla, y comprobará que todas las filas mostrarán el mismo valor.

Vamos entonces a mostrar la última fecha de contrato en un control *TDBEdit*, que en el ejemplo del disco he situado dentro de la barra de herramientas:

N°	Apellidos	Nombre	Extensión	Contrato	Salario
2	Nelson	Roberto	250	28/dic/1988	40.000,00 €
4	Young	Bruce	233	28/dic/1988	55.500,00 €
5	Lambert	Kim	22	06/feb/1989	25.000,00 €
8	Johnson	Leslie	410	05/abr/1989	25.050,00 €
9	Forest	Phil	229	17/abr/1989	25.050,00 €
11	Weston	K. J.	34	17/ene/1990	33.292,00 €
12	Lee	Terri	256	01/may/1990	45.332,00 €
14	Hall	Stewart	227	04/jun/1990	34.482,00 €
15	Young	Katherine	231	14/jun/1990	24.400,00 €
20	Papadopoulos	Chris	887	01/ene/1990	25.050,00 €

1.386.198,00 €

Como puede apreciar, he modificado la propiedad *DisplayFormat* del campo *UltimaFecha* para mostrar el nombre del mes. No obstante, si intentamos hacer lo mismo con *TotalSalarios*, veremos que el truco no funciona. Hay un par de pequeños problemas, que le adelanto:

- 1 Para estos campos no se pueden utilizar las propiedades habituales de recuperación de valores: *AsString*, *AsInteger*, *AsCurrency*, etc. Estas propiedades suelen extraer sus valores del *buffer* donde se almacena el registro activo del conjunto de datos, y está muy claro que la implementación de las estadísticas es muy diferente. Para obtener el valor de un *TAggregateField* debemos siempre interrogar su propiedad *Value*, que es de tipo *Variant*.
- 2 Por el mismo motivo, tampoco se puede utilizar *IsNull*. La solución consiste en pasar el valor en *Value* a la función *VarIsNull*, que en Delphi 6 se define dentro de la unidad *Variants*.
- 3 La propiedad *DisplayFormat* no tiene efecto sobre estos campos, cuando el valor devuelto es de tipo numérico. Es curioso, sin embargo, que sí funciona cuando se trata de fechas. Si necesita dar formato a estos valores, deberá interceptar el evento *OnGetText* de los campos correspondientes.

De todos modos, asignamos una cadena de formato en la propiedad *DisplayFormat* de *TotalSalarios*, e interceptamos su evento *OnGetText*:

```
procedure TwndPrincipal.EmpleadosTotalSalariosGetText(
  Sender: TField; var Text: string; DisplayText: Boolean);
var
  V: Variant;
begin
  V := Sender.Value;
  if not VarIsNull(V) then
    Text := FormatFloat(TAggregateField(Sender).DisplayFormat, V);
end;
```

Por último, para mostrar el total de los salarios en la barra de estado de la aplicación, he interceptado el evento *OnDataChange* de la fuente de datos asociada a *Empleados*:



```
procedure TwndPrincipal.dsEmpleadosDataChange (
    Sender: TObject; Field: TField);
begin
    StatusBar.Panels[0].Text := EmpleadosTotalSalarios.DisplayText;
end;
```

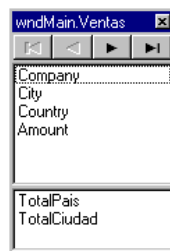
## Grupos y valores agregados

¿Y qué hay acerca del nivel de agrupamiento? Este valor indica al índice que dispare ciertos eventos internos cuando alcance el final de un grupo de valores repetidos. Tomemos como ejemplo el conjunto de datos que mostramos en la siguiente imagen:

Grupos y agregados					
Compañía	Ciudad	País	Total	Total ciudad	Total país
Shangri-La Sports Center	Freeport	Bahamas	11.954,85 €	97.598,45 €	190.343,05 €
Unisco			85.643,60 €		
SCUBA Heaven	Nassau		66.208,65 €	92.744,60 €	
Tora Tora Tora			26.535,95 €		
Adventure Undersea	Belize City	Belize	92.494,85 €	92.494,85 €	92.494,85 €
Norwest'er SCUBA Limited	Paget	Bermuda	76.698,75 €	76.698,75 €	99.052,75 €
Underwater SCUBA Company	Somerset		22.354,00 €	22.354,00 €	
Cayman Divers World Unlimited	Grand Cayman	British West Indies	59.660,05 €	76.511,80 €	76.511,80 €
Fisherman's Eye			12.022,00 €		
Safari Under the Sea			4.829,75 €		
Marmot Divers Club	Kitchener	Canada	12.223,25 €	12.223,25 €	97.358,90 €
Davy Jones' Locker	Vancouver		44.073,65 €	44.073,65 €	
On-Target SCUBA	Winnipeg		41.062,00 €	41.062,00 €	
Fantastique Aquatica	Bogota	Columbia	90.143,40 €	90.143,40 €	90.143,40 €
Sight Diver	Kato Paphos	Cyprus	261.575,80 €	261.575,80 €	261.575,80 €

En realidad, solamente las cuatro primeras columnas pertenecen al conjunto de datos: el país, la ciudad, el nombre de un cliente y el estado de su cuenta con nosotros. El índice que está activo, llamado *Jerarquia*, ordena por los tres primeros campos, y su *GroupingLevel* es igual a 2. De esta manera, podremos mantener estadísticas o agregados por los dos primeros campos del índice: el país y la ciudad.

Para obtener el resultado anterior, hay que entrar en el Editor de Campos del conjunto de datos y ejecutar el comando *New field*. Como recordará de la sección anterior, ahora tenemos la opción *Aggregate* en el tipo de campo. Como nombre, utilizaremos *TotalPorPaís*. Repetiremos la operación para crear un segundo campo agregado: *TotalPorCiudad*. Nuevamente, los dos campos se mostrarán en un panel situado en la parte inferior del Editor de Campos:



Debemos cambiar un par de propiedades para los objetos recién creados:

	TotalPorPais	TotalPorCiudad
Expression	sum(Saldo)	sum(Saldo)
IndexName	Jerarquia	Jerarquia
GroupingLevel	1	2

Vamos ahora a la rejilla de datos, y creamos un par de columnas adicionales, a la que asignamos explícitamente el nombre de los nuevos campos en la propiedad *FieldName*; recuerde que Delphi no muestra los campos agregados en la lista desplegable de la propiedad *FieldName* de un *TDBCColumn*. Esta vez, la respuesta que compartiremos para los eventos *OnGetText* de ambos campos será más sofisticada:

```

procedure TwndMain.AggregateGetText(Sender: TField;
  var Text: string; DisplayText: Boolean);
begin
  Text := '';
  if gbFirst in Ventas.GetGroupState(
    TAggregateField(Sender).GroupingLevel) then
    if not VarIsNull(Sender.Value) then
      Text := FormatCurr('0,.00 €', Sender.Value);
end;

```

La novedad consiste en el uso de la función *GetGroupState* del conjunto de datos. Esta devuelve un conjunto con los valores *gbFirst*, *gbMiddle* y *gbLast*, para indicar si estamos al inicio, al final o en mitad de un grupo. Solamente mostramos un texto no vacío cuando estamos al principio del grupo. Observe que tenemos que indicar el nivel del grupo para llamar a *GetGroupState*.

```

procedure TwndMain.FieldGetText(Sender: TField;
  var Text: string; DisplayText: Boolean);
begin
  if gbFirst in Ventas.GetGroupState(Sender.Tag) then
    Text := Sender.AsString
  else
    Text := '';
end;

```

Por último, compartimos el método anterior como respuesta a los eventos *OnGetText* de los campos que contienen el nombre de la ciudad y el país. Para determinar el nivel de agrupamiento de estos campos, que son campos “normales”, he modificado el valor de sus propiedades *Tag*, asignando 1 para el país y 2 para la ciudad.

## Relaciones maestro/detalles

**S**I LE GUSTAN LAS SIGLAS, AHÍ TIENE UNA nueva para su colección: *NF2*. Se trata de un término técnico derivado de *Non-First Normal Form*: relaciones que no satisfacen el criterio conocido como Primera Forma Normal. ¿Necesita almacenar datos de clientes con varias direcciones asociadas? Si hace caso a la ortodoxia relacional predicada por Codd, necesitará dos tablas diferentes. Pero si no le importa pecar, puede utilizar una sola tabla; siempre que cada registro de cliente tenga un campo *Direcciones...* que almacene todo un conjunto de registros de direcciones.

Los conjuntos de datos de MyBase permiten definir tablas *NF2*, que no satisfacen la primera forma normal. Es una característica muy potente, que solamente comparte con Oracle, y que resuelve algunos problemas importantes que existen en otras interfaces de acceso, como BDE o ADO. Antes, sin embargo, veremos otro mecanismo más tradicional para representar objetos complejos mediante los conjuntos de datos de Delphi.

### Representación de entidades complejas

Es muy fácil perder de vista los defectos del modelo de datos relacional. El defecto con el que más frecuentemente tropieza el programador es la incapacidad del modelo para representar directamente entidades que no sean “planas”. Pongamos por caso que tenemos que almacenar una lista de direcciones para cada uno de nuestros clientes: debemos entonces utilizar dos tablas separadas. Es cierto que podemos utilizar las restricciones de integridad referencial para sugerir la relación existente entre ambas tablas.

Pero nuestro problema ahora consiste en cómo recuperar datos pertenecientes a entidades complejas como la descrita desde una aplicación tradicional. Para ayudarnos, Delphi nos ofrece la posibilidad de asociar dos conjuntos de datos diferentes mediante una técnica conocida como *relación maestro/detalles*. Uno de los conjuntos de datos recibe el nombre de conjunto de datos *maestro*; al otro lo denominaremos conjunto *dependiente*, en general. La técnica de enlace es relativamente independiente de la clase del componente maestro, pero varía mucho respecto a la clase del conjunto dependiente. En este capítulo sólo mostraré la configuración aplicable a conjuntos de datos clientes.

La esencia de la relación se explica con facilidad. Cada vez que se cambia la fila activa en el conjunto maestro, se restringe el conjunto de filas de la tabla dependiente a los registros relacionados con la fila maestra activa. Si la tabla maestra es la de clientes y la tabla dependiente es la de pedidos, la última debe mostrar en cada momento sólo los pedidos realizados por el cliente activo:

CLIENTES	
Código	Nombre
1221	Kauai Dive Shoppe
1231	Unisco
1351	Sight Diver

PEDIDOS		
Número	Cliente	Fecha
1023	1221	2/Jul/88
1076	1221	26/Abr/89
1123	1221	24/Ago/93

CLIENTES	
Código	Nombre
1221	Kauai Dive Shoppe
1231	Unisco
1351	Sight Diver

PEDIDOS		
Número	Cliente	Fecha
1060	1231	1/Mar/89
1073	1231	15/Abr/89
1102	1231	6/Jun/92

Para establecer una relación *master/detail* entre dos conjuntos de datos sólo hay que hacer cambios en el que va a funcionar como conjunto de datos dependiente. En el caso de *TClientDataSet*, las propiedades que hay que modificar son las siguientes:

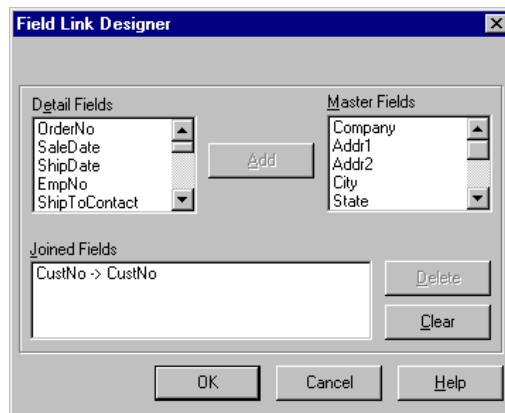
Propiedad	Propósito
<i>MasterSource</i>	Un <i>datasource</i> asociado a la tabla maestra
<i>IndexName</i> ó <i>IndexFieldNames</i>	Criterio de ordenación en la tabla dependiente
<i>MasterFields</i>	Campos de la tabla maestra que forman la relación

Es necesario configurar una de las propiedades *IndexName* ó *IndexFieldNames*; ya hemos visto que son modos alternativos y excluyentes de establecer un orden sobre los registros del conjunto de datos. Este criterio de ordenación es el que se aprovecha para restringir eficientemente el cursor sobre los detalles. En el ejemplo que mostramos antes, los pedidos deben estar ordenados por la columna *Cliente*.

Sin embargo, no tenemos que modificar directamente estas propiedades en tiempo de diseño, pues el editor de la propiedad *MasterFields* se encarga automáticamente de ello. Este editor, conocido como el Editor de Enlaces (*Links Editor*) se ejecuta cuando realizamos una doble pulsación sobre la propiedad *MasterFields*, y su aspecto depende de la clase del conjunto de detalles. Si se trata de una tabla en formato Paradox o dBase, el diálogo tiene un combo, con la leyenda *Available Indexes* para que indiquemos el nombre del índice por el cual se ordena la tabla dependiente. Si la relación *master/detail* está determinada por una restricción de integridad referencial, la mayoría de los sistemas de bases de datos crean de forma automática un índice secundario sobre la columna de la tabla de detalles. Suponiendo que las tablas del ejemplo anterior sean tablas Paradox con integridad referencial definida entre ellas, la tabla de pedidos debe tener un índice secundario, de nombre *Clientes*, que es el que

debemos seleccionar. Una vez que se selecciona un índice para la relación, en el cuadro de lista de la izquierda aparecen las columnas pertenecientes al índice elegido. Para este tipo de tablas, el Editor de Enlaces modifica la propiedad *IndexName*: el nombre del índice escogido.

En el caso de una tabla SQL o de MyBase, no aparece la lista de índices, y en el cuadro de lista de la izquierda aparecen todas las columnas de la tabla dependiente. Para probarlo, traiga dos componentes *TClientDataSet* sobre un formulario. Llame *Clientes* al primero, y asócielo al fichero *customer.xml*. Renombre el segundo como *Pedidos*, y asócielo a *orders.xml*. Añada entonces un *TDataSource*, llámelo *dsClientes* y modifique su propiedad *DataSet* para que apunte a la tabla de clientes. Lo nuevo viene ahora. Seleccione *Pedidos*, modifique su *MasterSource* para que apunte a *dsClientes* y haga doble clic en su propiedad *MasterFields*:



Ni las tablas SQL ni los conjuntos de datos clientes tienen, en principio, limitaciones en cuanto al orden en que se muestran las filas, por lo que basta con especificar las columnas de la tabla dependiente para que la tabla quede ordenada por las mismas. En este caso, la propiedad que modifica el Editor de Enlaces es *IndexFieldNames*: las columnas por las que se ordena la tabla. Pero con independencia del origen de los datos, las columnas de la lista de la derecha corresponden siempre al conjunto de datos maestro, y son las que se asignan realmente a la propiedad *MasterFields*. En el ejemplo anterior, *MasterFields* recibirá el valor *CustNo* al aceptar el diálogo.

Para poder visualizar el resultado, deberá traer un segundo *TDataSource*, y asociarlo a la tabla de pedidos. Para no complicar las cosas, traiga dos rejillas de datos y sepárelas con un control *TSplitter*, de la página *Additional* de la Paleta de Componentes.

#### NOTA

Observe que *dsClientes*, el primer *TDataSource* que añadimos, tiene dos funciones diferentes: avisar a *Pedidos* cada vez que cambie la fila activa de *Clientes*, para que éste muestre sólo las filas correspondientes, y avisar además al componente *TDBGrid* que muestra los datos de clientes, o a cualquier otro control asociado.

CustNo	Company	Addr1	Addr2
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy	Suite 103
1231	Unisco	PO Box Z-547	
1351	Sight Diver	1 Neptune Lane	
1354	Cayman Divers World Unlimited	PO Box 541	
1356	Tom Sawyer Diving Centre	632-1 Third Frydenhoj	
1380	Blue Jack Aqua Center	23-738 Paddington Lane	Suite 310
1384	VIP Divers Club	32 Main St.	

OrderNo	CustNo	SaleDate	ShipDate	EmpNo	ShipToContact	ShipToAddr1
1173	1231	16/07/94	16/07/94	127		
1178	1231	02/08/94	02/08/94	24		
1160	1231	01/06/94	01/06/94	110		
1202	1231	06/10/94	06/10/94	145		

El resultado final debe ser parecido a la imagen anterior. En la barra de herramientas he puesto cuatro botones asociados a las acciones estándar de navegación. Estas acciones ejercen su influencia sobre el control que esté activo en cada momento. Si nos situamos sobre la primera rejilla, la navegación afecta a la tabla de clientes, pero si movemos el foco a la segunda rejilla, navegaremos sobre los pedidos asociados.

## Navegación sobre detalles

Quiero aprovechar este mismo ejemplo para mostrarle algunas peculiaridades sobre el funcionamiento de *TDDataSource* y los conjuntos de datos. Supongamos que necesitamos conocer el total facturado por clientes que no viven en los Estados Unidos. Preste mucha atención:

*“Si estuviésemos trabajando con bases de datos SQL,  
la única forma decente de realizar esta suma sería enviar una consulta al servidor,  
jamás recorriendo las filas involucradas”*

... a no ser que sean pocas filas y ya las hayamos leído, por supuesto. En nuestro ejemplo no hay servidor SQL a la vista, por lo que tendremos que recorrer manualmente los registros. Pero incluso en este caso, tendríamos que decidir si vamos a realizar esta suma con cierta frecuencia, porque es posible incluso que sea más eficiente dejar que MyBase mantenga actualizado el valor de un campo de estadísticas.

Volvamos al ejemplo, y ponga un botón en algún sitio del formulario para efectuar la suma de los pedidos. Pruebe este algoritmo inicial, en respuesta al evento *OnExecute* de la acción correspondiente:

```
// PRIMERA VARIANTE: ¡¡¡MUY INEFICIENTE!!!
procedure TwndPrincipal.SumarExecute(Sender: TObject);
var
  Tiempo: Cardinal;
  Total: Currency;
begin
  Tiempo := GetTickCount; Total := 0;
```

```

Clientes.First;
while not Clientes.Eof do begin
  if not SameText(Clientes['COUNTRY'], 'US') then begin
    Pedidos.First;
    while not Pedidos.Eof do begin
      Total := Total + Pedidos['ITEMSTOTAL'];
      Pedidos.Next;
    end;
  end;
  Clientes.Next;
end;
ShowMessage(Format('%m: %d', [Total, GetTickCount - Tiempo]));
end;

```

Note que estoy accediendo a los valores de los campos de las tablas mediante el método más sencillo ... y más ineficiente. Me refiero a estas instrucciones:

```

if not SameText(Clientes['COUNTRY'], 'US') then
// ...
  Total := Total + Pedidos['ITEMSTOTAL'];

```

Sólo se lo advierto para que no sirva de precedente.

Habría observado el comentario que encabeza el primer listado de esta sección. Si ha preparado el ejemplo y pulsado el botón, comprenderá por qué es ineficiente. ¡Cada vez que se mueve una fila, las rejillas siguen el movimiento! Y lo que más tiempo consume es el dibujo en pantalla. Sin embargo, usted ya conoce los métodos *EnableControls* y *DisableControls*, que desactivan las notificaciones a los controles visuales ¿Por qué no utilizarlos?

```

// SEGUNDA VARIANTE: ;;;INCORRECTA Y DESASTROSA!!!
procedure TwndPrincipal.SumarExecute(Sender: TObject);
var
  Tiempo: Cardinal;
  Total: Currency;
begin
  Tiempo := GetTickCount; Total := 0;
  Clientes.DisableControls;           // ← NUEVO
  Pedidos.DisableControls;           // ← NUEVO
  try
    Clientes.First;
    while not Clientes.Eof do begin
      if not SameText(Clientes['COUNTRY'], 'US') then begin
        Pedidos.First;
        while not Pedidos.Eof do begin
          Total := Total + Pedidos['ITEMSTOTAL'];
          Pedidos.Next;
        end;
      end;
      Clientes.Next;
    end;
  finally
    Pedidos.EnableControls;           // ← NUEVO
    Clientes.EnableControls;          // ← NUEVO
  end;
  ShowMessage(Format('%m: %d', [Total, GetTickCount - Tiempo]));
end;

```

Ahora sí va rápido el algoritmo, ¡pero devuelve un resultado a todas luces incorrecto! Cuando llamamos a *DisableControls* estamos desconectando el mecanismo de notificación de cambios de la tabla a sus controles visuales... y también a las tablas que dependen en relaciones *master/detail*. Por lo tanto, se mueve la tabla de clientes, pero no se modifica el conjunto de filas de pedidos activas para el cliente seleccionado.

Hay una primera solución muy sencilla: utilice dos componentes *TDDataSource* acoplados a la tabla de clientes. Traiga un nuevo componente *DataSource1*, y cambie su propiedad *DataSet* para que apunte a *Clientes*. Entonces, modifique la propiedad *DataSource* de *DBGrid1* para que apunte a *DataSource1* en vez de a *dsClientes*. Por último, retoque el algoritmo de iteración del siguiente modo:

```
// TERCERA VARIANTE: ¡¡¡AL FIN BIEN!!!
procedure TwndPrincipal.SumarExecute(Sender: TObject);
var
    Tiempo: Cardinal;
    Total: Currency;
begin
    Tiempo := GetTickCount; Total := 0;
    DataSource1.Enabled := False;      // ← NUEVO
    Pedidos.DisableControls;
    try
        Clientes.First;
        while not Clientes.Eof do begin
            if not SameText(Clientes['COUNTRY'], 'US') then begin
                Pedidos.First;
                while not Pedidos.Eof do begin
                    Total := Total + Pedidos['ITEMSTOTAL'];
                    Pedidos.Next;
                end;
            end;
            Clientes.Next;
        end;
    finally
        Pedidos.EnableControls;
        DataSource1.Enabled := True;    // ← NUEVO
    end;
    ShowMessage(Format('%m: %d', [Total, GetTickCount - Tiempo]));
end;
```

Así el flujo de notificaciones se corta al nivel de una fuente de datos particular, no al nivel general del conjunto de datos. *Clientes* sigue enviando notificaciones a sus dos fuentes de datos asociadas. La fuente *dsClientes* propaga estas notificaciones a los objetos que hacen referencia a ella: en este caso, la tabla dependiente *dsPedidos*. Pero *DataSource1* se inhabilita temporalmente para que la rejilla asociada no reciba notificaciones de cambio.

¿Probamos otra solución? En Delphi 4 se introdujo la propiedad *BlockReadSize* en los conjuntos de datos. Cuando su valor es mayor que cero, el conjunto de datos entra en un estado especial: la propiedad *State* toma el valor *dsBlockRead*. En dicho estado, las notificaciones de movimiento se envían solamente a las relaciones *master/detail*, pero no a los controles de datos. Parece ser que también se mejora la eficiencia de las



lecturas, porque se leen simultáneamente varios registros por operación. Hay que tener en cuenta, sin embargo, dos inconvenientes:

- La única operación de navegación que funciona es *Next*.
- Para ciertos conjuntos de datos del BDE, parece ser que la modificación de esta propiedad en una tabla de detalles no funciona correctamente.

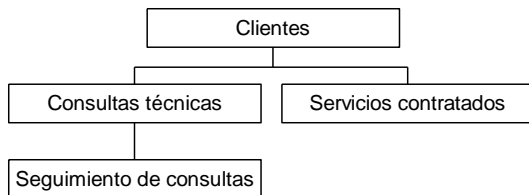
Teniendo en cuenta estas advertencias, nuestro algoritmo podría escribirse en esta forma alternativa:

```
// CUARTA VARIANTE: ;;;TAMBIEN CORRECTA!!!
procedure TwndPrincipal.SumarExecute(Sender: TObject);
var
    Tiempo: Cardinal;
    Total: Currency;
begin
    Tiempo := GetTickCount; Total := 0;
    // Se llama a First antes de modificar BlockReadSize
    Clientes.First;
    Clientes.BlockReadSize := 10;      // ← NUEVO
    Pedidos.DisableControls;          // Se mantiene
    try
        while not Clientes.Eof do begin
            if not SameText(Clientes['COUNTRY'], 'US') then begin
                Pedidos.First;
                while not Pedidos.Eof do begin
                    Total := Total + Pedidos['ITEMSTOTAL'];
                    Pedidos.Next;
                end;
            end;
            Clientes.Next;
        end;
    finally
        Pedidos.EnableControls;
        Clientes.BlockReadSize := 0;    // ← NUEVO
    end;
    ShowMessage(Format('%m: %d', [Total, GetTickCount - Tiempo]));
end;
```

Repito: ahora que ya sabe cómo realizar un doble recorrido sobre un par de tablas en relación *master/detail*, le aconsejo que solamente programe este tipo de algoritmos cuando esté trabajando con bases de datos de escritorio. Si está utilizando una base de datos cliente/servidor, la ejecución de una consulta es incomparablemente más rápida. Puede comprobarlo.

## Uso y abuso de la relación maestro/detalles

Gracias a que las relaciones *master/detail* se configuran en la tabla dependiente, y no en la maestra, es fácil crear estructuras complejas basadas en esta relación:



En el diagrama anterior, la tabla de clientes controla un par de tablas dependientes. A su vez, la tabla de consultas técnicas, que depende de la tabla de clientes, controla a la de seguimiento de consultas.

Si quiere practicar lo aprendido, puede probar a añadir la tabla de líneas de pedidos, almacenada en el fichero *items.xml*, al ejemplo que hemos desarrollado.

Semánticamente, las relaciones maestro/detalles suelen estar relacionadas con restricciones de integridad referencial, aunque no estén definidas explícitamente. Por ejemplo, en casi todos los dialectos de SQL, la tabla de pedidos que para este ejemplo hemos leído desde un fichero XML, tendría en su definición una restricción similar a ésta:

```

create table ORDERS (
    /* ... */
    foreign key (CustNo) references CUSTOMER (CustNo),
    /* ... */
);
  
```

Como podrá deducir, el grupo de columnas a continuación de las palabras **foreign key** son las que van a parar a la propiedad *IndexFieldNames*, y las columnas que siguen al nombre de la tabla de referencia son las que se mencionan en *MasterFields*. Es cierto que en este ejemplo hay una sola columna en la relación, pero ya sabe que SQL permite que sean varias.

No obstante, esto no significa que una relación maestro/detalles sea *equivalente* a una restricción de integridad referencial de SQL. Ni siquiera significa que haya una rela-

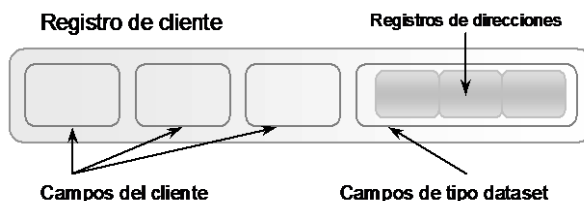
ción uno/muchos entre las tablas involucradas, porque también puede utilizarse para las relaciones muchos/uno. Podemos designar como tabla maestra la tabla de pedidos, y como tabla de detalles la tabla de clientes. En este caso, por cada fila de la primera tabla debe haber exactamente una fila en la tabla de clientes. Si se aprovecha esta relación en una ficha de entrada de pedidos, cuando el usuario introduce un código de cliente en la tabla de pedidos, automáticamente la tabla de clientes cambia su fila activa al cliente cuyo código se ha tecleado. La siguiente imagen, correspondiente a una de las demostraciones de Delphi, muestra esta técnica.

Los cuadros de edición que aparecen con fondo gris en la parte superior izquierda del formulario pertenecen a la tabla de clientes, que está configurada como tabla de detalles de la tabla principal de pedidos. Observe que la rejilla muestra también datos de otra tabla de detalles: las líneas correspondientes al pedido activo.

## Conjuntos de datos anidados

La técnica de relacionar dos conjuntos de datos que acabamos de ver funciona para cualquier interfaz de acceso soportada por Delphi. En cambio, la técnica que estudiaremos ahora es exclusiva de MyBase, y de Oracle, que la soporta a partir de su versión 8. En vez de utilizar dos tablas diferentes para representar un registro con detalles, utilizaremos un solo conjunto de datos con registros de estructura compleja.

La siguiente imagen muestra un posible registro de cliente:



En vez de utilizar una tabla separada para almacenar los registros de direcciones asociadas, en el propio registro del cliente se define un campo adicional, con un tipo especial. Como esta vez no se trata de una característica de SQL, lo que nos importa es la constante que se utiliza en Delphi para el nuevo tipo: *ftDataSet*. El contenido de la columna puede ser interpretado como una lista de registros. Evidentemente, los registros anidados pueden estructurarse según las necesidades del programador, aunque en este momento todos deben tener las mismas columnas. O son manzanas, o son melones<sup>21</sup>, pero no se pueden mezclar las frutas polimórficamente.

Creo que será más fácil de explicar si vemos un ejemplo. Vamos a programar una libreta de direcciones de clientes, con la peculiaridad de que cada cliente puede tener varias direcciones registradas. Previsible, ¿verdad? Para que sea un poco más “real”, vamos a situar los componentes de acceso a datos en un módulo de datos, en el que definiremos el siguiente método público:

```
procedure TmodDatos.CrearLibreta(const AFileName: string);
begin
    with TClientDataSet.Create(nil) do
        try
            FieldDefs.Add('Nombre', ftString, 20, True);
            FieldDefs.Add('Apellidos', ftString, 20, True);
            FieldDefs.Add('Movil', ftString, 15, False);
            with FieldDefs.AddFieldDef do begin
                Name := 'Direcciones';
                DataType := ftDataSet;
                ChildDefs.Add('Descripcion', ftString, 20);
                ChildDefs.Add('Direccion1', ftString, 30, True);
                ChildDefs.Add('Direccion2', ftString, 30, False);
                ChildDefs.Add('CP', ftSmallInt, 0, True);
                ChildDefs.Add('Ciudad', ftString, 25, True);
                ChildDefs.Add('Telefono', ftString, 15, False);
            end;
            CreateDataSet;
            SaveToFile(AFileName, dfXML);
        finally
            Free;
        end;
    end;
```

Ya conocíamos la propiedad *FieldDefs* de *TClientDataSet*, pero no habíamos visto usar la propiedad *ChildDefs* de un *TFieldDef* aislado. Es fácil: *CrearLibreta* añade una definición de columna, le asigna un nombre y le asigna el tipo *ftDataSet*, que ya he mencionado. A continuación, utiliza la propiedad *ChildDefs* de esa definición de columna para crear columnas anidadas, exactamente igual que si estuviésemos definiendo una tabla “tradicional” de direcciones. Cuando todos está listo, llamamos a *CreateDataSet*, y guardamos en un fichero XML el conjunto de datos recién creado... y vacío. Es decir, que nos quedamos sólo con su esquema relacional.

---

<sup>21</sup> Je, je, je, me gustan los melones.

**NOTA**

Recordará que *FieldDefs* es una propiedad publicada, ¿cierto? Entonces, ¿es posible tener estas definiciones preparadas en tiempo de diseño? ¡Sí, es perfectamente posible! Pero he encontrado algunos problemas al hacerlo así, por lo que he preferido ir sobre seguro, y crear el conjunto de datos en tiempo de ejecución.

Como *CrearLibreta* trabaja con un *TClientDataSet* temporal, necesitaremos otro componente de este tipo para mostrar el contenido de una libreta. Añada entonces un *TClientDataSet* al módulo de datos, y llámelo *Clientes*. Ahora tenemos un problema: ¿cómo crearemos los campos del nuevo conjunto de datos? Mejor le digo como lo resolví: puse un botón en la ventana principal para llamar a *CrearLibretas*. Ejecuté la aplicación, creé un fichero vacío, regresé a Delphi y lo asocié a *Clientes*, y sólo entonces añadí y configuré sus objetos de acceso a campos:



Observe que hay un campo llamado *Direcciones*, para el que Delphi ha creado una variable llamada *ClientesDirecciones*, del tipo *TDataSetField*. Y que los cuatro botones de navegación del borde superior de la ventana están apagados, porque el conjunto de datos está vacío.

En realidad, *CrearLibreta* debe ser ejecutado por una acción *FileNew* que debemos traer a la ventana principal. En la respuesta a su evento *OnExecute* pedimos el nombre del fichero que se va a crear y se llama al método mencionado. Una vez que ya existe el fichero, se cierra *Clientes* y se asocia con él, para luego activar el conjunto de datos.

```
procedure TwndPrincipal.FileNewExecute(Sender: TObject);
begin
    if SaveDialog.Execute then begin
        modDatos.CrearLibreta(SaveDialog.FileName);
        modDatos.Clientes.Close;
        modDatos.Clientes.FileName := SaveDialog.FileName;
        modDatos.Clientes.Open;
    end;
end;
```

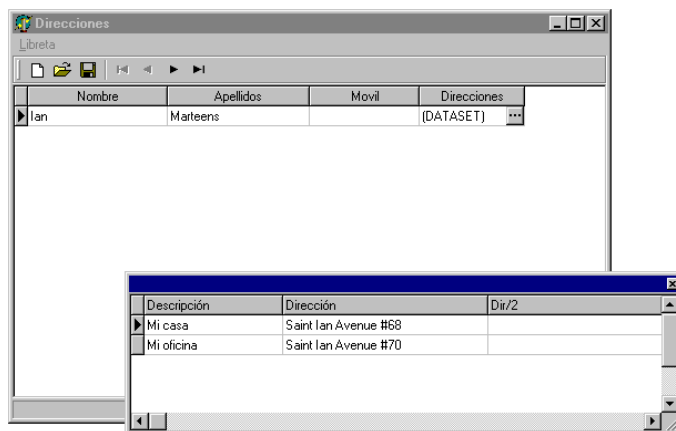
Para abrir un fichero XML ya existente, he traído una acción estándar *FileOpen*. Las acciones de esta clase poseen un diálogo de apertura como subcomponente. Hay que seleccionar ese diálogo, en la propiedad *Dialog* de la acción, para modificar las opciones y el filtro de tipos de fichero. Y en vez de interceptar el evento *OnExecute*, debemos asociar una respuesta a *OnAccept*:

```

procedure TwndPrincipal.FileOpenAccept(Sender: TObject);
begin
    modDatos.Clientes.Close;
    modDatos.Clientes.FileName := FileOpen.Dialog.FileName;
    modDatos.Clientes.Open;
end;

```

Para rematar la faena... pues ya sabe: traiga una rejilla a la ventana principal y una fuente de datos, y configúrelas para que muestren el contenido de *Clientes*. Echele un vistazo al resultado:



Como puede apreciar, la columna que corresponde al campo de tipo *dataset* se muestra de forma similar a como se mostraría un campo blob. Pero si selecciona la celda, verá que aparece automáticamente un botón con puntos suspensivos. Y si pulsa sobre él, saltará sobre su cara una rejilla emergente para que introduzcamos las direcciones del cliente activo. Para poder hacer la foto, tuve que cambiar el tamaño y posición de la rejilla emergente. Y prometerle que iba a llevarla de paseo al parque si se quedaba quieta al menos dos segundos.

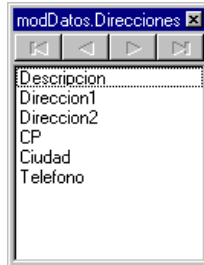
#### ADVERTENCIA

Tenga cuidado al teclear direcciones, porque Delphi intentará grabar primero el registro de cabecera. Recuerde que el nombre y los apellidos del cliente han sido marcados como requeridos.

## Anclando los detalles

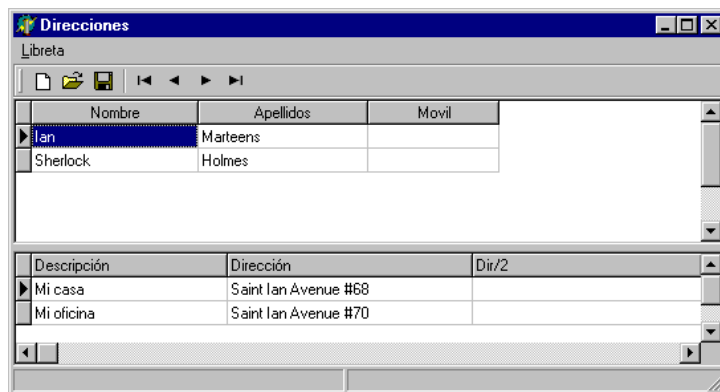
Si tuviéramos que depender siempre de la rejilla flotante para trabajar con un conjunto de datos anidados, le aseguro que no estaría hablando bien de este recurso. Para terminar el ejemplo, hay que traer un segundo *TClientDataSet* al módulo de datos; cambie su nombre a *Direcciones*. Sólo tendremos que modificar una de sus propiedades, *DataSetField*, para que apunte al campo desde donde extraerá sus registros, *ClientesDirecciones*. Después, haga doble clic para mostrar el Editor de Campos de *Direccio-*

nes, y verá que puede añadir componentes de acceso a campo para las columnas del conjunto de datos anidado.



Pruebe a activar *Clientes*, estando *Direcciones* inactivo; verá que la activación se propaga en cascada. Lo mismo sucederá si cerramos *Clientes*: también se cierra *Direcciones*.

Por último, traiga una segunda rejilla y otro *TDataSource* a la ventana principal, y haga que apunten a *Direcciones*. Sería un detalle por su parte si quita entonces la innecesaria cuarta columna de la rejilla de clientes. No se preocupe, que no le va a pasar lo que a Sansón.







## Actualizaciones en MyBase

**N**ADA SURGE DE LA NADA, Y LAS TABLAS de una base de datos se llenan porque alguien se empeña en añadirles registros. En este capítulo aprenderemos los misterios y secretos de la actualización directa de los conjuntos de datos de Delphi. Utilizo el adjetivo “directa” porque existe también una actualización indirecta: si el conjunto de datos proviene de una base de datos SQL, es también posible ejecutar instrucciones SQL de actualización sobre la misma.

Aunque describiremos, en concreto, las actualizaciones sobre conjuntos de datos clientes en MyBase, la mayoría de estas técnicas pueden aplicarse a los conjuntos de datos del BDE, de ADO Express y de IB Express. Pero quiero dejar algo muy claro: las técnicas de actualización “directa” sobre el cursor que vamos a estudiar en este capítulo no son recomendables para los conjuntos de datos SQL, en la mayoría de los escenarios. DB Express va más allá y simplemente las prohíbe. Existen casos, por supuesto, como la presencia de una caché de actualizaciones o aplicaciones que se ejecutan en redes de área local con poco volumen de datos, en que se puede relajar esta norma. Pero es una decisión que debe meditarse mucho antes de tomarla.

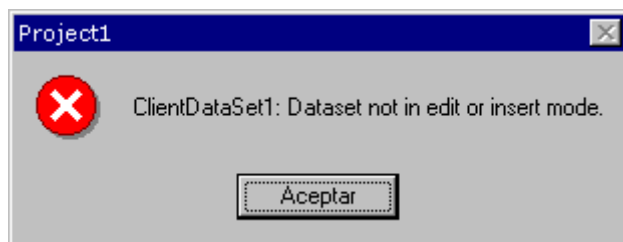
### El diagrama de estados

Hagamos un tonto experimento: traiga un *TClientDataSet* sobre un formulario, modifique su propiedad *FileName* para que apunte a un fichero de MyBase y actívelo. Supondré, para mayor concreción, que ha elegido *employee.xml*. Si lo desea, para ver más claro lo va a pasar, traiga un *TDataSource*, modifique su *DataSet* para que apunte al conjunto de datos y añada una *TDBGrid*, cambiando su propiedad *DataSource* para que apunte al objeto conveniente. Finalmente, ponga un botón en alguna zona libre de la ventana, e intercepte su evento *OnClick*:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    // ;Código incorrecto!
    with ClientDataSet1.FieldByName('Salary') do
        AsFloat := 2 * AsFloat;
end;
```

En pocas palabras, cuando pulsemos el botón intentaremos duplicar el salario del afortunado empleado que esté seleccionado en ese momento. Aunque no será tanta

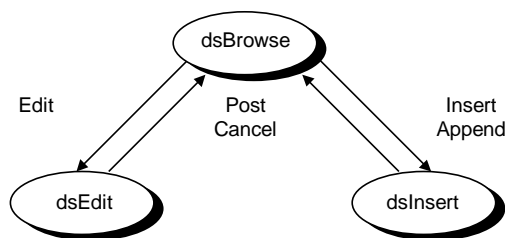
su fortuna, porque cuando ejecute el programa y pulse el botón, obtendrá la siguiente excepción:



Para el novato, esto contradice su experiencia con la edición sobre controles de datos. Digamos que estamos mostrando un conjunto de datos abierto sobre una rejilla. Nos situamos sobre cualquier fila y columna y comenzamos a teclear. Cambiamos de fila y vemos cómo los cambios se hacen definitivos. Esto lleva a muchos a pensar que, en cualquier momento, podemos asignar un nuevo valor a campo, sin más. Sin embargo, lo que sucede durante la edición sobre controles de datos es que éstos realizan cierto “trabajo sucio” a nuestras espaldas. Veamos en qué consiste.

La clave está en la propiedad *State* del conjunto de datos, que mencioné al presentar los métodos de apertura y cierre. En aquel momento, adelanté que un conjunto de datos cerrado se encuentra en el estado *dsInactive*, y que al abrirse pasa al estado *dsBrowse*. En el capítulo anterior, además vimos la existencia de dos estados internos: *dsCalcFields* y *dsInternalCalc*. Y vimos que podíamos asignar un valor en el buffer interno de un campo calculado solamente cuando el estado activo era uno de estos dos estados especiales.

Lo mismo sucede con las asignaciones sobre campos de datos. Para que el campo no las rechace, el conjunto de datos tiene que estar en uno de los dos nuevos estados: *dsEdit*, si es para modificar un registro ya existente, o *dsInsert*, cuando estamos añadiendo un nuevo registro.



El “problema” de *State* es que se trata de una propiedad de sólo lectura. Si queremos que el conjunto de datos pase del estado normal, *dsBrowse*, al estado *dsEdit* no podemos modificar directamente *State*. Las transiciones de datos se logran ejecutando métodos del conjunto de datos. En particular, para llegar al estado *dsEdit* es necesario

ejecutar el método *Edit*. Para comprobar su funcionamiento, modifique el manejador de eventos que mostré antes en la siguiente manera:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    // ¡Ya funciona!
    ClientDataSet1.Edit;
    with ClientDataSet1.FieldByName('Salary') do
        AsFloat := 2 * AsFloat;
    end;
end;
```

Ejecute la aplicación y pulse el botón; el salario del afortunado empleado se duplicará inmediatamente. Pero como la felicidad nunca dura demasiado, observe lo que pasa si pulsa la tecla ESC: ¡el registro vuelve a su estado inicial, y se pierde la modificación! Lo que sucede es que las modificaciones realizadas en campos mientras el conjunto de datos se encuentra en modo de edición no se graban hasta que regresamos al estado inicial, *dsBrowse*. Existen dos métodos que provocan esa transición: *Post*, que de paso graba las modificaciones, y *Cancel*, que es el método disparado por la rejilla cuando pulsamos ESC.

El siguiente método muestra la forma “canónica” de programar una modificación sobre la fila activa de un conjunto de datos:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    // Un "buen ejemplo" de grabación
    ClientDataSet1.Edit;
    try

        // -- Modificaciones sobre el registro activo --
        with ClientDataSet1.FieldByName('Salary') do
            AsFloat := 2 * AsFloat;
        // -- Fin de las modificaciones -----

        ClientDataSet1.Post;    // Todo ha funcionado bien
    except
        ClientDataSet1.Cancel; // Detectamos un problema
        raise;                // Relanzamos la excepción
    end;
end;
```

Es muy importante que no olvidemos cancelar las modificaciones si se detecta un problema; en caso contrario, el conjunto de datos permanece en modo de edición, y al tratar de cambiar la fila activa se reintentará grabar el registro. Por otra parte, es igualmente importante volver a lanzar la excepción original dentro de la cláusula **except**. La única concesión se admite cuando queremos cambiar el mensaje de error, y terminamos el algoritmo lanzando nuestro propio objeto de excepción.

## Autoedición

¿Cómo es entonces que, cuando tecleamos datos sobre una rejilla, no tenemos que pulsar explícitamente el botón de edición, para llamar al método *Edit*? La respuesta está en la forma en que se implementan los controles de datos, de acuerdo a los convenios establecidos en Delphi. Cada vez que uno de estos controles recibe una notificación que indique que su contenido está a punto de cambiar, el control intenta que el conjunto de datos al que está asociado pase al modo de edición. La transición, sin embargo, se realiza a través de un intermediario: el componente de tipo *TDataSource* que sirve de puente entre el conjunto de datos y el control, y es posible activar o desactivar la transición automática por medio de una de sus propiedades:

```
property TDataSource.AutoEdit: Boolean;
```

Normalmente, *AutoEdit* contiene el valor *True*. Podemos cambiar este valor a *False* si queremos obligar a nuestro cliente para que pulse el botón de edición antes de que toque los controles y fastidie el contenido del conjunto de datos. Es una buena medida contra usuarios torpes. Y valga la redundancia.

Y ya que hablamos de autoedición, debemos mencionar la existencia de la *grabación* automática. Esta tiene lugar cuando abandonamos la fila activa y existen cambios en la misma.

## Inserción

Las inserciones directas sobre conjuntos de datos se programan de forma similar a las modificaciones. La diferencia consiste en utilizar uno de los métodos *Insert* o *Append*, en vez de *Edit*. Ambos métodos crean una fila ficticia en la lista de registros en memoria, y ponen el conjunto de datos en el estado *dsInsert*. A partir de ese momento, ya podemos realizar las modificaciones en los valores de los campos y, para grabarlos definitivamente en donde corresponda, se llama también al método *Post*. De esta manera, el algoritmo canónico de inserción directa es como el siguiente:

```
// Un "buen ejemplo" de inserción
ClientDataSet1.Append;
try

    // -- Modificaciones sobre el registro activo --
    // ...
    // -- Fin de las modificaciones -----

    ClientDataSet1.Post;    // Todo ha funcionado bien
except
    ClientDataSet1.Cancel;  // Detectamos un problema
    raise;                 // Relanzamos la excepción
end;
```

Y ahora viene la pregunta del millón de euros: ¿cuál es la diferencia entre *Insert* y *Append*? Muy simple: tiene que ver con la posición, dentro de la lista de registros, en que se sitúa la nueva fila “virtual”. Si es *Insert*, la nueva fila abre un hueco entre el registro activo y el registro anterior, si es que existe. Cuando se ejecuta *Append*, por el contrario, la nueva fila va a parar siempre al final de la lista de registros. Sin embargo, cuando al final llamamos a *Post*, vemos que el resultado de ambos métodos es idéntico: la fila se ubica definitivamente en la posición que le corresponde según el índice activo.

¿Hay siempre un índice activo en un conjunto de datos? Casi siempre, aunque no siempre se trate de algo explícito.

## Métodos abreviados

Existen métodos para simplificar la inserción de registros en tablas y consultas. Se llaman *InsertRecord* y *AppendRecord*:

```
procedure TDataSet.InsertRecord(const Values: array of const);  
procedure TDataSet.AppendRecord(const Values: array of const);
```

En principio, por cada columna del conjunto de datos donde se realiza la inserción hay que suministrar un elemento en el vector de valores. El primer valor se asigna a la primera columna, y así sucesivamente. Pero también puede utilizarse como parámetro un vector con menos elementos que la cantidad de columnas de la tabla. En ese caso, las columnas que se quedan fueran se inicializan con el valor por omisión. El valor por omisión depende de la definición de la columna; si no se ha especificado otra cosa, se utiliza el valor nulo de SQL.

Si una tabla tiene tres columnas, y queremos insertar un registro tal que la primera y tercera columna tengan valores no nulos, mientras que la segunda columna sea nula, podemos pasar la constante *Null* en la posición correspondiente:

```
CDS1.InsertRecord(['Valor1', Null, 'Valor3']);  
// ...  
CDS1.AppendRecord([  
    RandomString(CDS.FieldName('Cadena').Size),  
    Random(MaxInt)])];
```

Delphi también ofrece el método *SetFields*, que asigna valores a los campos de una tabla a partir de un vector de valores:

```
CDS1.Edit;  
CDS1.SetFields(['Valor1', Null, 'Valor3']);  
CDS1.Post;
```

El inconveniente de estos métodos abreviados se ve fácilmente: nos hacen dependientes del orden de definición de las columnas de la tabla. Se reestructura la tabla y ¡adiós inserciones!

## Eliminando registros

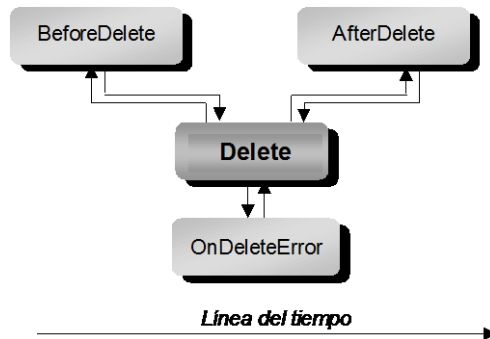
La operación de actualización más sencilla sobre tablas y consultas es la eliminación de filas. Esta operación se realiza con un solo método, *Delete*, que actúa sobre la fila activa del conjunto de datos:

```
procedure TDataSet.Delete;
```

Después de eliminar una fila, se intenta dejar como fila activa la siguiente. Si ésta no existe, se intenta activar la anterior. Por ejemplo, para borrar todos los registros que satisfacen cierta condición necesitamos este código:

```
procedure TForm1.BorrarTodosClick(Sender: TObject);
begin
    CDS1.First;
    while not CDS1.Eof do
        if Condicion(CDS1) then
            CDS1.Delete
            // No se llama a Next en este caso
        else
            CDS1.Next;
end;
```

Como puede ver, para borrar un registro no hace falta pasar por una sucesión de estados intermedios, como sucede con las inserciones y modificaciones. No obstante, durante la ejecución de *Delete* se disparan algunos eventos útiles:



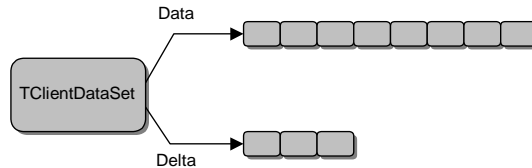
Podemos aprovechar *BeforeDelete* para borrar registros de detalles en cascada, o para impedir el borrado si existen tales registros. Por ejemplo, suponiendo que *Proveedores* actúa como conjunto maestro de un segundo componente llamado *Productos*:

```
procedure TmodDatos.ProveedoresBeforeDelete(DataSet: TDataSet);
begin
    if not Productos.IsEmpty then
        raise Exception.Create(
            'No se puede eliminar un proveedor con productos');
end;
```

Para detener la operación nos ha bastado lanzar una excepción.

## El registro de actualizaciones

Hace ya un buen centenar de páginas, cuando presenté el componente *TClientDataSet*, mencioné qué éste almacenaba sus registros en memoria, y mostré el siguiente diagrama para explicar cómo se estructuraba esta información:



Los datos “originales” se almacenan, como podrá suponer, en *Data*, que es una propiedad de tipo *OleVariant*, e internamente se organizan como un vector de bytes. *Data* es una propiedad de lectura y escritura. En cambio, *Delta* es de sólo lectura, y almacena también un vector dentro de un valor *OleVariant*. Dentro de *Delta* encontraremos... bueno, ¿por qué no lo descubre usted mismo?

Cree una aplicación típica, con un conjunto de datos cliente (*Empleados*, en el ejemplo del CD) y una rejilla. Luego sitúe una segunda rejilla donde se le antoje, y engánchela a un conjunto de datos cliente sin configurar; por supuesto, a través de un *TDataSource*. Al segundo conjunto de datos lo llamaremos *Delta*. Finalmente, añada una acción *VerDelta* al formulario y manéjela de la siguiente forma:

```
procedure TwndPrincipal.VerDeltaExecute(Sender: TObject);
begin
  try
    Delta.Data := Empleados.Delta;
  except
    Delta.Data := Null;
  end;
end;
```

Lo que hacemos es considerar que la propiedad *Delta* tiene en el fondo un formato similar a *Data*, y que son compatibles por asignación. Modifique algunos registros en el fichero de empleados y pulse el botón de la acción, para ver qué obtenemos:

Deltas					
Nº emp	Apellidos	Nombre	Extensión	Contrato	Salario
1	Ian	Marteens	111	01/ene/2001	100.000,00 €
2	Nelson	Roberto	250	28/dic/1988	40.000,00 €
4	Lee	Bruce	233	28/dic/1988	55.500,00 €
5	Lambert	Kim	22	06/feb/1989	25.000,00 €
8	Johnson	Leslie	410	05/abr/1989	25.050,00 €
9	Forest	Phil	229	17/abr/1989	25.050,00 €
11	Weston	K. J.	34	17/ene/1990	33.292,00 €

EmpNo	LastName	FirstName	PhoneExt	HireDate	Salary
4	Young	Bruce	233	28/12/88	55500
	Lee				
1	Ian	Marteens	111	01/01/01	100000

En el ejemplo de la imagen, había modificado un registro, el de *Bruce Young*, cambiando su apellido a *Lee*. Y me había insertado yo mismo, ganando un salario decente. Al mostrar el contenido de *Delta*, he encontrado tres registros: los dos primeros corresponden a la modificación, por ser la primera operación en ser realizada. El primero contiene los valores originales de la fila en cuestión, y el segundo solamente muestra algo en las columnas modificadas. En el caso de la inserción, veremos un solo registro, con los valores insertados.

Y ahora le propondré algunas preguntas bizantinas, para que experimente un poco. Por ejemplo, ¿qué pasa si modificamos un registro ya insertado? Para empezar, podría duplicarme el salario. Respuesta: sigue habiendo un solo registro en *Delta* para la inserción, pero se modifican los valores nuevos. ¿Y si realiza más modificaciones en el registro de Mr. Lee (Young de soltero)? Nada, que las nuevas modificaciones se mezclan con las anteriores.

Pero la pregunta más importante es, ¿qué pasa si cerramos la aplicación y volvemos a ejecutarla? Pues que si pedimos nuevamente el *Delta*, ¡comprobaremos que las modificaciones se siguen almacenando por separado! Esto sucede a pesar de que los registros se han grabado en el fichero XML al cerrarse el conjunto de datos. Si quisiéramos mezclar definitivamente el contenido de *Delta* y *Data*, dejando *Delta* vacío, por supuesto, tendríamos que ejecutar el siguiente método:

```
procedure TCustomClientDataSet.MergeChangeLog;
```

Tenga mucho cuidado, no obstante, porque esta operación es irreversible.

### ADVERTENCIA

Cuando estudiemos el uso de *TClientDataSet* como caché de otros conjuntos de datos, veremos que no se debe utilizar *MergeLogChanges* en esa configuración del componente. Para hacer efectivos los cambios en la base de datos SQL asociada se utiliza simplemente el método *ApplyUpdates*, que presentaremos en el momento oportuno.

## El estado de una fila

¿Cómo distingue el *TClientDataSet* entre los distintos tipos de registros que se almacenan en *Delta*? La respuesta es un valor que se asocia internamente a cada fila, no solamente de *Delta*, sino también de *Data*. En especial, es con esta última propiedad con la que podemos probar el uso de la siguiente función:

```
type
    TUpdateStatus = (usUnmodified, usModified,
                    usInserted, usDeleted);

function TDataSet.UpdateStatus: TUpdateStatus; virtual;
```

*UpdateStatus* devuelve siempre el tipo de la fila activa: si se trata de una fila que no se ha modificado, si se han realizado modificaciones, si es una fila nueva... o si se trata



de una fila borrada. ¿Le extraña esto último? Le voy a dar dos razones para lo contrario. En primer lugar, porque si usted borra un registro, éste se guarda en Delta marcado con el estado *usDeleted*. En segundo lugar, dentro de poco presentaré una propiedad que nos permitirá mostrar los registros borrados. Antes de que ejecutemos *MergeChangeLog*, por supuesto.

En el proyecto *Deltas* he creado un manejador compartido por los eventos *OnDrawColumnCell* de las dos rejillas. He llamado *DibujarCelda* al método, y en su interior cambio el estilo o el color del texto de acuerdo al tipo de fila que se esté dibujando:

```
procedure TwndPrincipal.DibujarCelda(Sender: TObject;
const Rect: TRect; DataCol: Integer; Column: TColumn;
State: TGridDrawState);
begin
    with TDBGrid(Sender) do begin
        with Canvas.Font do
            case DataSource.DataSet.UpdateStatus of
                usModified: Style := [fsBold];
                usInserted: Color := clBlue;
                usDeleted: Color := clRed;
            end;
        DefaultDrawColumnCell(Rect, DataCol, Column, State);
    end;
end;
```

La propiedad que sirve para especificar los tipos de registros que se muestran en los controles visuales es la siguiente:

```
property TCustomClientDataSet.StatusFilter: TUpdateStatusSet;
```

Note que *StatusFilter* admite un conjunto de valores del tipo *TUpdateStatus*. Sin embargo, la función no es tan útil como puede parecer a simple vista. He creado tres acciones en el proyecto *Deltas*: *VerModificadas*, *VerInsertadas* y *VerEliminadas*. En las tres he activado la propiedad *AutoCheck*, y he asignado en sus *Tags* los valores 1, 2 y 3, respectivamente, coincidiendo con el ordinal de las constantes a las que van asociadas. Luego he puesto las tres opciones en un menú emergente, que he asociado a un botón de la barra de herramientas, y he creado un manejador compartido para el evento *OnExecute* de todas:

```
procedure TwndPrincipal.CambiarFiltro(Sender: TObject);
var
    US: TUpdateStatusSet;
begin
    US := Empleados.StatusFilter;
    if TAction(Sender).Checked then
        Include(US, TUpdateStatus(TAction(Sender).Tag))
    else
        Exclude(US, TUpdateStatus(TAction(Sender).Tag));
    Empleados.StatusFilter := US;
end;
```

Observará que, efectivamente, si marca la opción de ver las filas modificadas, aparecerán sólo las filas modificadas... pero acompañadas por los fantasmas de los registros originales. Como Dorian Gray y su puñetero retrato.

## Distintos grados de arrepentimiento

Ya que *TClientDataSet* mantiene una estructura tan complicada para seguir la pista a los cambios, ¿sería mucho pedir aprovecharla para deshacerlos? No se preocupe, que existe una amplia oferta de métodos relacionados con la contricción y el arrepentimiento de los pecados.

En primer lugar, debemos saber si hay cambios para deshacer. Para eso tenemos la propiedad *ChangeCount*, de sólo lectura:

```
property TCustomClientDataSet.ChangeCount: Integer;
```

Hay que tener un poco de cuidado, porque *ChangeCount* representa el número de registros cambiados, no el número de modificaciones sobre esos registros. Además, como veremos en breve, si queremos saber si hay cambios en un conjunto de datos clientes, muchas veces es apropiado añadir los cambios no confirmados sobre el registro activo, preguntando por el valor de la propiedad *Modified*.

- 1 Si *Modified* es *False*, y *ChangeCount* es cero, el conjunto de datos está intacto.
- 2 Si *Modified* es *True* y *ChangeCount* es cero, el usuario está destrozando el registro activo, pero todavía no ha llamado a *Post*.
- 3 Si *Modified* es *False*, pero *ChangeCount* es mayor que cero, el usuario ha confirmado la actualización de uno o más registros, aunque no esté editando ninguno en este preciso momento.
- 4 Y si *Modified* es *True* y *ChangeCount* es mayor que cero, se trata de un usuario reincidente, que ya ha modificado algún otro registro y vuelve a las andadas.

Ahora vienen los métodos que deshacen cambios. El acto más drástico de arrepentimiento es ejecutar *CancelUpdates*. Su efecto consiste en vaciar todo el contenido del vector *Delta*:

```
procedure TCustomClientDataSet.CancelUpdates;
```

Hay otra muestra de contricción más oportunista y taimada: sólo deshacemos los cambios sufridos por el registro activo del conjunto de datos; si es que se ha cambiado algo en él, por supuesto. El método necesario es *RevertRecord*:

```
procedure TCustomClientDataSet.RevertRecord;
```

Pero sin lugar a dudas, el método más popular será *UndoLastChange*. Su prototipo es el siguiente:

```
function UndoLastChange(FollowChange: Boolean): Boolean;
```

Con *UndoLastChange* anulamos el último cambio aplicado, según indica su nombre. El parámetro *FollowChange* tiene que ver con la posición del cursor después de su ejecución. Si pasamos *True*, el cursor se desplazará al registro que se ha visto afectado, o a sus cercanías; tenga en cuenta que si lo último que hemos hecho ha sido insertar un registro, la llamada a este método lo eliminará del conjunto de datos. Si *FollowChange* vale *False*, el cursor no se inmuta. Pero mi recomendación es permitir que el cursor se mueva, porque así se asemeja más a la forma en que se deshace el último cambio en un procesador de texto.

## Puntos de control

La estrella del espectáculo es una propiedad llamada *SavePoint*. Es de lectura y escritura, y ésta es su declaración:

```
property TCustomClientDataSet.SavePoint: Integer;
```

El valor de *SavePoint* viene a ser, más o menos, como el tamaño actual del vector almacenado en *Delta*. De esta forma, podemos guardar el valor de *SavePoint* en una variable local, realizar unas cuantas operaciones sobre el conjunto de datos y, si algo huele mal, deshacer todos los cambios de golpe, restaurando en *SavePoint* el valor que habíamos salvado:

```
var
    LocalSave: Integer;
begin
    LocalSave := ClientDataSet1.SavePoint;
    try
        // ... cambios y más cambios ...
    except
        ClientDataSet1.SavePoint := LocalSave;
        raise;
    end;
end;
```

Así que, en cierto modo, podemos imitar en parte el comportamiento de las transacciones sobre un conjunto de datos cliente. Ahora bien, reconozco que *SavePoint* es más importante desde el punto de vista técnico, pues nos da una pista sobre la implementación de *TClientDataSet*, que práctico. He tenido que rebuscar un poco para dar con un ejemplo de uso real.

En la aplicación cuya imagen muestro en la página siguiente, un cliente solicita un préstamo; como tiene algunos préstamos o hipotecas activos en ese momento, pide que volvamos a negociar las condiciones, y que consolidemos todos los préstamos en un solo contrato. En el cuadro de diálogo *Propuesta de refinanciación* se edita un *TClientDataSet* complejo, con datos anidados; entre ellos, una lista con los préstamos activos. Oculto tras la ventana activa (*Selección de productos*) se encuentra el botón que ha disparado este otro diálogo. Cada vez que seleccionamos un producto, estamos modificando un campo lógico de un registro de detalles.

Propuesta de refinanciación

Producto: L Fuente: Todos los demás

Fecha efectiva: 01/08/2000 Plazo/frecuencia: 6 1

Primer vencimiento: 10/08/2000 Sólo int./plazo pago: 0 0

Fecha vencimiento: 10/01/2001 Interés diferencial: 26.00% 0.00%

Plan del seguro: Prima única 1 asegurado Sexo: Ciudad:

Importe solicitado: 500.000 pts

Saldo refinanciado: 0 pts

Total financiado: 520.750 pts

Intereses totales: 32.298 pts

Seguro total: 20.750 pts

Contrato total: 553.048 pts

Tipo Código Estado Fecha

L 2 Refinanciado 06/

L 3 Refinanciado 06/

L 4 Refinanciado 18/

L 5 Activo 20/

Selección de productos para refinanciación

Producto	Estado	Saldo	Impagado	Morosidad
L 05	Activo	49.815 pts	42.184 pts	90

Descripción	Importe	Importe a cobrar
Principal pendiente	49.815 pts	49.815 pts
Intereses impagados	2.391 pts	2.391 pts
Intereses hasta la fecha	783 pts	783 pts
Seguro impagado	921 pts	921 pts
Seguro próxima cuota	307 pts	307 pts
Gastos devolución impagados	4.000 pts	4.000 pts
Gastos de mora impagados	5.031 pts	5.031 pts
<b>Totales del producto</b>	<b>64.742 pts</b>	<b>64.742 pts</b>
<b>Totales productos seleccionados</b>	<b>64.742 pts</b>	<b>64.742 pts</b>

Sucursal: 100 SUP99 01/08/2000

Puede que, después de haber marcado varias filas, el usuario decida dejar los datos tal como estaban antes de mostrar el diálogo. La forma en que resolví el problema fue quedarme con el valor de *SavePoint* en el momento de activar el diálogo, y restaurar el punto de control si la ejecución modal se cancela.

## RECOMENDACIÓN

La única precaución aconsejable: si el usuario ha realizado cambios en el registro de cabecera, tenemos que recordar que esas modificaciones no pasan a *Delta* hasta que se ejecute *Post*. Por lo tanto, antes de leer el valor de *SavePoint* es necesario que llamemos a *CheckBrowseMode*, para quedarnos con un punto estable adonde retroceder.

## Validaciones a nivel de campo

Ahora que ya sabemos cómo modificar los datos de un *TClientDataSet*, vamos a ocuparnos de la validación de las actualizaciones. En general, hay tres tipos de validaciones que se podrían ejecutar durante la entrada de datos, de acuerdo al momento en que se disparan:

- Cada vez que el usuario teclea un carácter.
- Cuando el usuario ha terminado de teclear el valor de un campo.
- Cuando el usuario ha terminado con el registro.

El primer tipo de validación es bastante problemático, porque no toda cadena parcial tecleada por el usuario representa un valor correcto. Pero, de todos modos, es posible realizar este tipo de validación en la mayoría de los casos. Ya hemos presentado la propiedad *ValidChars*, común a todos los tipos de campos. Y Delphi permite utilizar máscaras de edición para campos de tipo cadena o fecha. Ahora bien, esas máscaras suelen ser demasiado rígidas, porque exigen una equivalencia biyectiva con los valo-

res que se tecleen. Por ejemplo, la siguiente máscara es la “recomendada” por Delphi para teclear fechas:

```
!99/99/00;1;_
```

Haga el experimento de asignarla en la propiedad *EditMask* de un campo de fecha, e intente teclear el cuatro de julio, para que vea por qué detesto las máscaras de Delphi. ¿La solución? O creas tus propios componentes con validación incorporada... o utilizas alguna de las muchas colecciones de componentes que hay en Internet. Como toda técnica que empleemos será obligatoriamente “a la medida”, no insistiré más en esta modalidad de validación.

Delphi sí nos da mucha libertad y facilidades para que implementemos los otros dos tipos de validaciones. Antes de comenzar la presentación de propiedades y eventos, quiero dejar claro algo importante: si la condición que desea verificar afecta a un solo campo, utilice la validación a nivel de campo, si afecta a varios campos, debe usar obligatoriamente la verificación a nivel de registro. Parece elemental, ¿no?

Lo que sucede es que algunos programadores, acostumbrados a desarrollar aplicaciones con la rígida interfaz de usuario de MS-DOS, intentan forzar las validaciones a nivel de campo para que involucren varias columnas. Eso sólo es posible cuando hay un orden muy estricto de entrada de los datos. Pero la interfaz gráfica de Windows se caracteriza por todo lo contrario: el usuario debe poder manipular los controles de edición en el orden que mejor le parezca. ¿Se puede forzar a Windows para que trabaje como MS-DOS? Hombre, seguro que sí. Pero no vale la pena el esfuerzo.

## Propiedades de validación en campos

Las propiedades de este primer grupo existen en todos los tipos de campos:

Propiedad	Propósito
<i>Required</i>	Indica si el campo es obligatorio o no
<i>CustomConstraint</i>	Expresión SQL que debe cumplirse en las modificaciones
<i>ConstraintErrorMessage</i>	Error a mostrar si falla <i>CustomConstraint</i>

*Required* se utiliza para indicar si un campo admite valores nulos o no. En un *TClientDataSet* que trabaja directamente con un fichero de MyBase, la propiedad se puede definir al crear el fichero, o puede activarse posteriormente. En algunas interfaces de acceso a SQL, el propio sistema propaga el atributo correspondiente desde la base de datos al componente de acceso a campos, pero a otras, como es el caso de ADO Express, hay que echarles una mano. Cuando intentamos asignar un valor nulo a un campo marcado como *Required*, se produce el siguiente mensaje de error:

*“Field YoQueSe must have a value”*

La reacción instintiva de muchos programadores es no utilizar la validación predefinida... simplemente porque el mensaje sale en inglés. No debería ser así, porque este “problema” puede resolverse traduciendo los mensajes de la aplicación, sin necesidad de usar una sola línea adicional de código.

El otro pretexto que he escuchado para no usar este mecanismo es: “...sí, pero yo quiero además que se seleccione el control donde el usuario debe teclear el valor...” Pero se trata también de otra queja sin fundamento. En la clase *TField* existe el siguiente método:

```
procedure TField.FocusControl;
```

Cuando falla alguna de las validaciones impuestas por Delphi a nivel de campos, la propia VCL aplica *FocusControl* sobre el campo conflictivo. La implementación de *FocusControl* busca entonces un control visual asociado a ese campo para pasarle el foco del teclado.

Mucho más interesante es el uso de *CustomConstraint*, pues en ella podemos teclear una condición que debe cumplir el campo. Supongamos que tenemos un campo llamado *Email* en el que almacenamos una dirección de correo electrónico. Podríamos utilizar la siguiente expresión en *CustomConstraint*:

```
x like '%%.____' or x like '%%.____'
```

¿Por qué *x*? Simplemente porque *x* no significa nada en SQL; podemos elegir cualquier otro identificador para representar el valor del campo, siempre que seamos coherentes en su uso. Notará también que he utilizado el operador **like**. Esto es posible porque estamos utilizando un *TClientDataSet*, y la sintaxis aceptada para los campos de los conjuntos de datos clientes es la misma que se utiliza en las condiciones de filtro que ya hemos estudiado.

#### **NOTA**

El BDE es la otra interfaz de acceso que permite el uso de *CustomConstraint* en sus campos. No obstante, la sintaxis permitida es diferente, porque es el propio BDE quien evalúa las expresiones. De todos modos, la mayoría de las aplicaciones de interfaz interactiva que se programan actualmente, utilizan *TClientDataSet* para la capa visual.

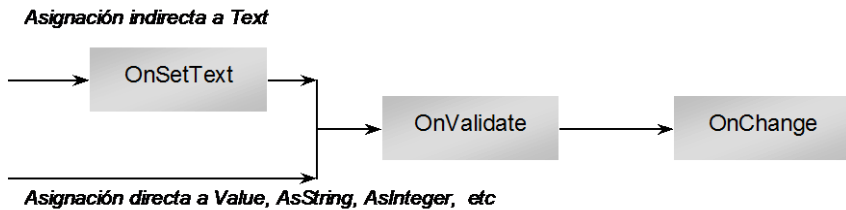
Si ha hecho la prueba de activar *CustomConstraint* en algún campo, habrá visto que el mensaje que aparece por omisión es demasiado “técnico”. Por este motivo existe la propiedad *ConstraintErrorMessage*, para que digamos el mensaje que queremos que aparezca.

Dije al principio de la sección que las propiedades anteriores estaban presentes en todas las clases derivadas de *TField*. En cambio, los campos de tipo numérico ofrecen dos propiedades en exclusiva: *Min* y *Max*; puede imaginar para qué se utilizan.

En realidad, existe otra propiedad paralela a *CustomConstraint*, llamada *Imported-Constraint*. Es una herencia de los tiempos del BDE, y se utilizaba en conjunción con el Diccionario de Datos del BDE. Ahora que el BDE está en la Unidad de Cuidados Intensivos, casi podemos olvidarnos de ella.

## Eventos de validación en campos

Hay verificaciones a nivel de campo que no pueden realizarse solamente con las propiedades explicadas. En esos casos, debemos recurrir al evento *OnValidate* del campo. Pero antes de poner ejemplos, quiero que analice este diagrama, que muestra el orden de disparo de los eventos en un campo:



Si asignamos directamente un valor al campo, mediante las propiedades *Value*, *AsInteger*, *AsString*... internamente se realizará una asignación “tentativa”: las propiedades que representan el valor cambiarán su valor, pero se disparará el evento *OnValidate*, para ver si estamos satisfechos. Durante esa llamada podemos disparar una excepción. El campo la interpretará como un gruñido de desaprobación, restaurará su valor inicial, y dejará que la excepción se propague, para que aparezca el mensaje de error que usted haya utilizado para la excepción. Entonces disparará el evento *OnChange*, para avisar que el campo tiene un nuevo valor, esta vez definitivo.

Algunos programadores creen que *OnChange* se dispara cada vez que pulsamos una tecla en un control de datos. Está claro que no es así. Normalmente, la cadena que se teclea en un control no se asigna al campo subyacente hasta que el usuario abandona el campo, o cuando pulsa INTRO.

Precisamente, si el campo recibe su valor por medio de un control visual, lo que sucede en realidad es que el control asigna la cadena tecleada por el usuario en la propiedad *Text* del campo. En tal caso se dispara un evento adicional que ya conocemos: *OnSetText*. Por supuesto, también pueden producirse excepciones durante el procesamiento de este evento, que interrumpirían la asignación. Pero hay que saber distinguir entre los errores que deben señalarse en *OnSetText* y en *OnValidate*. Recuerde que *OnSetText* no se disparará en todas las ocasiones en que cambie el valor del campo.

Supongamos que hay que verificar que la fecha de una entrevista de trabajo no caiga en un fin de semana. Esto habría que comprobarlo en *OnValidate*. Al ejecutarse este evento, sin embargo, podemos asumir que ya existe un valor de tipo fecha en la pro-

propiedad *Value* del campo. Las verificaciones del tipo “el mes vale entre 1 y 12” y “el día sólo puede ser 31 para ciertos meses” deben haber sido ejecutadas durante la asignación a la propiedad *Text*, en el evento *OnSetText* o internamente.

### ACLARACION

Quiero también aclarar que *OnValidate* solamente se ejecuta cuando se intentan actualizaciones. Si especificamos una condición de validación sobre una tabla, y ciertos registros ya existentes violan la condición, Delphi no protestará al visualizar los campos. Lo mismo puede decirse acerca de las propiedades de validación de campos e incluso de registros.

El siguiente método, programado como respuesta a un evento *OnValidate*, verifica que un nombre propio debe contener solamente caracteres alfabéticos, porque no somos robots:

```
procedure TmodDatos.VerificarNombrePropio(Sender: TField);
var
  S: string;
  I: Integer;
begin
  S := Sender.AsString;
  for I := Length(S) downto 1 do
    if (S[I] <> ' ') and not IsCharAlpha(S[I]) then
      DatabaseError('Carácter no permitido en nombre propio');
  end;
```

Este manejador puede ser compartido por varios componentes de acceso a campos, como pueden ser el campo del nombre y el del apellido. Es por esto que se extrae el valor del campo del parámetro *Sender*. Si falla la verificación, la forma de impedir el cambio es interrumpir la operación elevando una excepción; en caso contrario, no se hace nada especial. Para lanzar la excepción he utilizado la función *DatabaseError*. La llamada a esta función es equivalente a la siguiente instrucción:

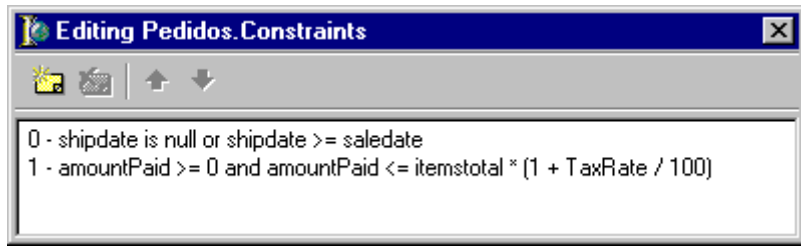
```
raise EDatabaseError.Create(
  'Carácter no permitido en nombre propio');
```

El problema de utilizar **raise** directamente es que se consume más código, pues también hay que crear el objeto de excepción en línea. La excepción *EDatabaseError*, por convenio, se utiliza para señalar los errores de bases de datos producidos por el usuario o por la VCL, no por el BDE.

## Validaciones a nivel de registro

Si tenemos que comprobar que se cumple una condición que afecte a más de una columna, estamos obligados entonces a realizar la verificación a nivel de registro. Al igual que sucede con las validaciones de campos, existen propiedades y eventos para facilitarnos la tarea. La propiedad se llama *Constraints*, y es una colección de objetos de tipo *TCheckConstraint*:



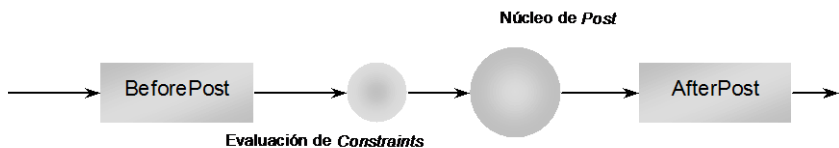


Cada *TCheckConstraint* individual tiene las siguientes propiedades:

Propiedad	Significado
<i>CustomConstraint</i>	La expresión SQL que debe cumplirse siempre
<i>ErrorMessage</i>	El mensaje de error para cuando no se cumpla
<i>FromDictionary</i>	¿Ha sido importada desde el Diccionario de Datos?
<i>ImportedConstraint</i>	La expresión, cuando viene del Diccionario de Datos

Volvemos a tropezar con el Diccionario de Datos; como el BDE está a punto de pasar a mejor vida, podemos ignorar las últimas dos propiedades... mientras Borland no diga lo contrario.

¿Por qué una lista de condiciones? En principio, podríamos mezclarlas todas en una gigantesca expresión, utilizando operadores **and**. Pero así perderíamos la posibilidad de mostrar un error diferente para cada cláusula que fallase. Observe que esta vez no podemos usar variables arbitrarias como *x* e *y*, sino que hay que utilizar los nombres de columnas verdaderos.



Por otra parte, si queremos realizar una comprobación por código, el evento apropiado es *BeforePost*. En el diagrama anterior se muestra la secuencia de acontecimientos que se producen al ejecutar el método *Post*. Primero, se dispara el evento *BeforePost*, si es que hay un manejador asociado. Solamente entonces es que se realiza la evaluación de las condiciones especificadas en *Constraints*. Después se ejecuta el “núcleo” del método, el código que realmente realiza las grabaciones. Y para terminar, se dispara el evento *AfterPost*, si no se han producido errores en los pasos anteriores.

Para señalar un error dentro de *BeforePost* hacemos lo mismo que en *OnValidate*: lanzar una excepción:

```

procedure TmodDatos.EmpleadosBeforePost(DataSet: TDataSet);
begin
    if (Empleados.State = dsEdit) and
        (EmpleadosSalary.Value > 2 * EmpleadosSalary.OldValue) then

```

```
DatabaseError('¡Desmesurado incremento salarial!');
end;
```

Observe que tenemos la posibilidad de consultar el estado en que se encuentra el conjunto de datos. Es útil hacerlo porque *Post* se utiliza tanto para las inserciones como para las modificaciones de registros existentes. En este ejemplo concreto, sólo me interesan las modificaciones, porque voy a consultar el *valor anterior* del campo salario, en la propiedad *OldValue* del componente de campo. Si el nuevo salario es más del doble del salario original, deniego la grabación del registro.

### ADVERTENCIA

¡Tenga mucho cuidado! No todos los tipos de conjuntos de datos permiten el uso de *OldValue* en sus campos. Además de MyBase, se puede utilizar esta propiedad en los conjuntos de datos del BDE, pero es necesario que el componente se encuentre en el modo conocido como *actualizaciones en caché*. Mi consejo es que, si necesita consultar el valor anterior de un campo, compruebe primero si su interfaz de acceso lo permite. En caso negativo, es fácil guardar este valor en variables locales al módulo si interceptamos los eventos *AfterEdit* y *AfterInsert*.

## Inicialización de campos

Otro asunto a tener en cuenta cuando se configura un conjunto de datos para realizar inserciones a través de él, son los valores por omisión que deben tomar sus campos. También aquí quiero hacer una aclaración, aunque solamente vale para componentes vinculados a bases de datos SQL:

- 1 Si usted ha definido valores por omisión al crear la tabla, pero no los ha propagado a la aplicación (usando las propiedades y eventos que veremos en breve), el usuario no sabrá de la existencia de esos valores hasta que grabe el registro.
- 2 Si, por el contrario, usted especifica valores por omisión para los campos dentro de la aplicación, la regla establecida en el servidor nunca llegará a aplicarse... ¡porque cada vez que el usuario envíe un nuevo registro al servidor, lo enviará con valores en cada una de sus columnas!

Para que comprenda mejor lo que está en juego, pongamos un ejemplo muy concreto. Usted define una tabla de pedidos, con un campo para la fecha de entrada, y especifica que el valor por omisión para esta columna sea la fecha actual; claro, se trata de la fecha del servidor. Supongamos ahora que creamos un formulario para que un usuario teclee datos de pedidos. Si no especificamos, en Delphi, un valor por omisión para el componente de campo que accede a la fecha del pedido, el usuario verá algo curioso: aunque deje vacío el control de la fecha de alta, cuando se produzca la grabación el campo asumirá un valor como por arte de magia. Por supuesto, no hay nada malo en esto.

Ahora, ¿qué pasaría si configurásemos el campo de fecha en Delphi para que se inicialice con la fecha actual? Simplemente, que los pedidos se van a crear no con la hora real de grabación en el servidor, sino con la hora del momento en que el usuario

llama al método *Insert* o *Append*; la hora del cliente, en otras palabras. Además, la regla que tiene el servidor dice, más o menos: “si ves que llega un pedido con fecha nula, pon la fecha del sistema”. Pero ahora todos los pedidos llegan con la fecha inicializada. ¿Es bueno o es malo? Por lo general, da lo mismo. Pero si obligatoriamente la hora debe ser la del servidor, tendrá que elegir entre estas alternativas:

- 1 Quitar la regla de inicialización en el lado cliente. Es lo más sencillo y recomendable. Claro, el cliente no sabrá con qué fecha se queda el registro mientras no lo grabe.
- 2 Si quiere que el cliente se haga una idea de la hora, puede dejar la regla de inicialización. Pero entonces tendrá que modificar esa hora en el servidor, posiblemente mediante un *trigger*. Como ve, es una complicación innecesaria.

Si queremos un valor por omisión para un campo, debemos intentarlo primero con la propiedad *DefaultExpression* del campo:

```
property TField.DefaultExpression: string;
```

Se trata, otra vez, de una expresión que es evaluada por el *TClientDataSet*, y que por lo tanto debe cumplir con la misma sintaxis que los filtros. Si el conjunto de datos pertenece al BDE o a ADO, claro está, la sintaxis será diferente. Normalmente, los valores por omisión son simples constantes. Sólo hay que recordar encerrar entre comillas simples las constantes de cadenas y fechas:

```
0,5  
'Aprobado'  
'15/1/2001'
```

Observe que hay algo de peligro con las fechas y las constantes reales con decimales, porque el separador decimal y el orden de los elementos de fechas deben suministrarse de acuerdo a las convenciones locales.

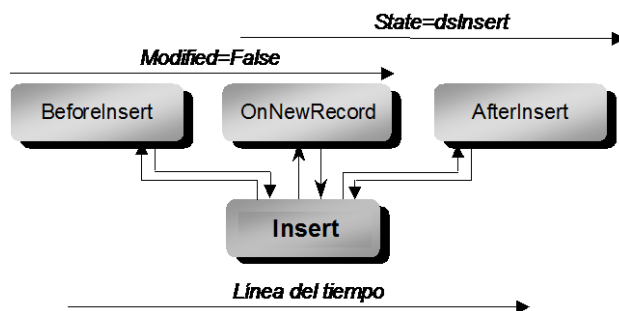
¿Y si el campo es de tipo fecha, o fecha y hora, y queremos que el valor por omisión sea la fecha o la hora actual? ¿Podemos especificar expresiones que no sean simples constantes? Voy a ser sincero: no descubrí cómo lograrlo hasta que, preparando los ejemplos para este capítulo, se me encendió la chispa. En primer lugar, la documentación no deja claro cuál es la sintaxis de las expresiones que puedes utilizar en los valores de omisión. Uno sospecha que es la misma de los filtros, aunque sea por economía de los medios de evaluación. Pero si pruebas la función *GetDate*, que según la ayuda devuelve la fecha y hora actual, ves que no funciona... El problema es que en la documentación, *GetDate* aparece escrita como en Pascal, sin los paréntesis. Sólo cuando añadí los paréntesis, en un momento de inspiración, logré que todo cobrase sentido:

```
GetDate()  
GetDate() + 7  
Date(GetDate())  
Time(GetDate())
```

La primera de las expresiones anteriores devuelve la fecha y hora actual. La segunda devuelve la fecha y hora, pero dentro de una semana. En los dos últimos ejemplos, se extrae solamente la fecha o la parte de la hora.

## El evento de inicialización

Pero por mucho que nos empeñemos, siempre habrá alguna inicialización que no pueda lograrse mediante *DefaultExpression*. Y en ese momento podremos utilizar el evento *OnNewRecord*, del conjunto de datos. Este evento se dispara durante la ejecución de los métodos *Insert* y *Append*. El siguiente diagrama muestra el orden de disparo de éste y otros eventos relacionados:



En primer lugar, quiero recordarle que *Insert...* no inserta nada, a pesar de su nombre. Lo que hace es crear un “hueco” en la caché de registros del conjunto de datos, y limpiar el *buffer* del registro activo. Quien realmente “inserta”, o graba, es *Post*. Como consecuencia, en *BeforeInsert* no podemos efectuar asignaciones de valores a campos, porque el conjunto de datos no se halla todavía en el estado *dsInsert*. En cambio, en *AfterInsert* el conjunto de datos ya ha pasado a ese estado, y podríamos asignar valores iniciales a sus campos. ¿Por qué entonces hay un evento *OnNewRecord* adicional, que se dispara también después que el conjunto de datos ha entrado en el modo de inserción?

La diferencia entre *OnNewRecord* y *AfterInsert* consiste en que las asignaciones realizadas a los campos durante el primer evento *no* marcan el registro actual como modificado. ¿Y en qué notamos esta diferencia? Tomemos el fichero *employee.xml*, y visualicémoslo en una rejilla. Interceptemos el evento *OnNewRecord* y asociémosle el siguiente procedimiento:

```
procedure TForm1.Table1NewRecord(DataSet: TDataSet);
begin
    DataSet['HireDate'] := Date;
end;
```

Estamos asignando a la fecha de contratación la fecha actual, algo lógico en un proceso de altas. Ahora ejecute el programa y realice las siguientes operaciones: vaya a la última fila de la tabla y utilizando la tecla de cursor FLECHA ABAJO cree una nueva fila

moviéndose más allá de este último registro. Debe aparecer un registro vacío, con la salvedad de que el campo que contiene la fecha de contratación ya tiene asignado un valor. Ahora dé marcha atrás sin tocar nada y el nuevo registro desaparecerá, precisamente porque, desde el punto de vista de Delphi, no hemos modificado nada en el mismo. Recuerde que esta información nos la ofrece la propiedad *Modified* de la clase *TDataSet*.

Para experimentar la diferencia, desacople este manejador de eventos de *OnNewRecord* y asócielo al evento *AfterInsert*. Si realiza estas mismas operaciones en la aplicación resultante verá cómo si se equivoca y crea un registro por descuido, no puede deshacer la operación con sólo volver a la fila anterior.



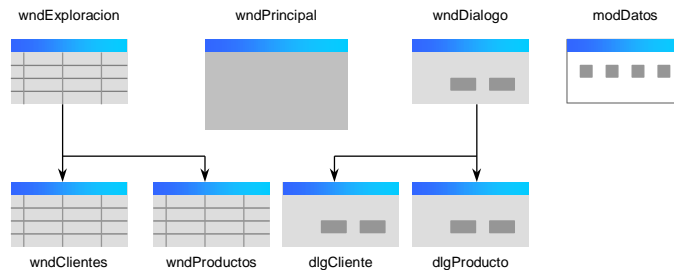
## Herencia visual y prototipos

LA CLAVE PARA DESARROLLAR APLICACIONES, y no perder dinero en el proceso, es terminirlas lo antes posible. Y cuando se programa en Delphi, una de las principales ayudas la ofrece un recurso conocido como *herencia visual*, que permite definir prototipos de formularios con el diseño visual y el código fuente común de varias ventanas. En este capítulo presentaré un prototipo de aplicación muy sencillo, con el propósito de que el lector se familiarice con las posibilidades de esta técnica.

### Decisiones iniciales de diseño

La primera decisión: ¿será, o no, una aplicación MDI? Veremos que es fácil saltar de un modelo a otro, pero prefiero utilizar aplicaciones MDI. Principalmente, porque prefiero también trabajar con ventanas no modales, ya que permiten más libertad al usuario. Y cuando se trabaja con ventanas no modales, la organización de estas se simplifica al residir todas ellas dentro del espacio delimitado por la ventana principal. Si ha escuchado alguna leyenda negra sobre las aplicaciones MDI, sepa que se trata de calumnias.

Este dibujo muestra las clases que van a existir en nuestro proyecto de ejemplo, de acuerdo a su relación de herencia:



Sólo las ventanas de la fila superior pertenecen realmente al prototipo de proyecto. La ventana principal del proyecto se llama *wndPrincipal*, y su propiedad *FormStyle* vale *fsMDIForm*; naturalmente, esta ventana no se usa como prototipo para herencia. A la

derecha del diagrama tenemos el módulo de datos, *modDatos*, del que se creará una sola instancia global.

El papel del tradicional módulo de datos global será mayor o menos en dependencia de la segunda decisión de diseño: ¿permitiremos sólo una o más de una ventana de búsqueda y navegación por entidad? Seré más concreto. Seleccionemos una de las entidades con las que trabajará la aplicación, digamos que *clientes*. Tendremos una ventana para teclear una condición de búsqueda, obtener como resultado una lista de clientes y navegar sobre ellos. ¿Una o varias? Prefiero tener una sola ventana de búsqueda y navegación por entidad. Eso significa que necesitaremos una única instancia del conjunto de búsqueda de clientes, y que podremos ubicarla sin remordimientos en el módulo global. En nuestro proyecto, todas esas ventanas de búsqueda y navegación descenderán de una misma clase, que en el dibujo aparece señalada como *wndExploracion*. Será un formulario con su propiedad *FormStyle* igual a *fsMDIChild*.

Para tomar la última decisión hay que pensar un poco más. Suponga que ya tenemos una ventana de búsqueda y navegación sobre clientes. Seleccionamos un cliente de la lista y hacemos doble clic para mostrar un cuadro de diálogo. Así podemos ver más detalles del registro, e incluso aprovechar para modificarlo. Esa ventana de detalles y edición, ¿se ejecutará de forma modal? En caso afirmativo, solamente podremos ver los datos de un solo cliente cada vez. O, por el contrario, podríamos mostrar los detalles en una ventana no modal, y permitir que haya varias instancias de ese tipo activas simultáneamente. Para simplificar, voy a utilizar ventanas de detalles modales; reconozco, sin embargo, que para ciertas aplicaciones es obligatorio que los detalles aparezcan en ventanas no modales. Dentro del proyecto de ejemplo, todos los cuadros de diálogos con detalles de registros descenderán de la clase *TwndDialogo*.

## Iniciando el proyecto

Comencemos creando y configurando los elementos globales del proyecto. Creamos un nuevo proyecto; el ejemplo que viene en el CD se llama *Prototipos*, pero usted puede llamarlo como le venga en ganas. Cambie el nombre de la ventana principal a *wndPrincipal*, modifique su propiedad *FormStyle* a *fsMDIForm*, y asigne *True* en *ShowHint*. Es también recomendable hacer que *WindowState* valga *wsMaximized*. Guarde la unidad con el nombre de *Principal*.

Para crear un módulo de datos, ejecute el comando de menú *File|New|Data module*. Ponga en su interior un componente *TImageList*, para garantizar la disponibilidad global de una lista de imágenes. Llame *modDatos* al módulo, y guárdelo como *Datos*. Regrese a la ventana principal, ejecute el comando *File|Use unit* para incluir la unidad *Datos*, y añada:

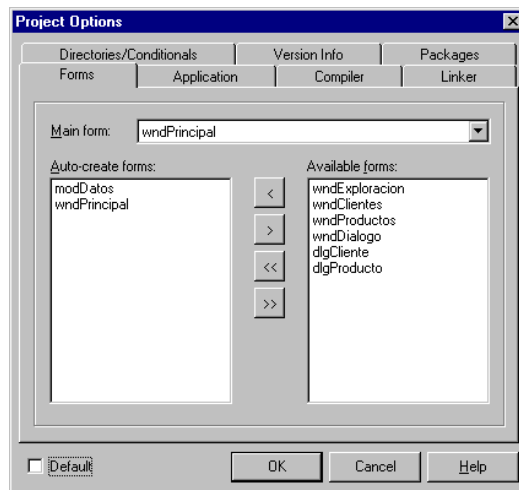
- Una lista de acciones, *ActionList*. Enlace su propiedad *ImageList* con la lista de imágenes del módulo de datos.



- Un menú, *MainMenu*. Haga que su propiedad *Images* apunte también a la lista de imágenes global.
- Un componente *CoolBar*, dentro del cual deberá ubicar dos controles *ToolBar*. Para el *CoolBar*, asigne *True* en su *DocSite*, y active *AutoSize*. En cuanto a las dos barras de herramientas, modifique las propiedades *Images*, *AutoSize*, *Flat*, y elimine ese molesto borde superior retocando *EdgeBorders*.

Deberá también traer algunas acciones predefinidas, como las cuatro acciones de navegación, de la categoría *DataSet*, las acciones de manejo de ventanas, y quizás alguna que otra de la categoría *File*. No le aburriré contando los detalles de la configuración.

Lo que sí debe hacer ahora es ejecutar el comando de menú *Project|Options*:



Por supuesto, no tenemos tantas ventanas todavía, pero asegúrese de que el módulo de datos y la ventana principal se creen automáticamente al iniciarse la aplicación, y que el módulo de datos se cree *antes* que la ventana principal, tal como muestro en la imagen anterior.

## Creación y destrucción automática

Lo siguiente será crear y configurar el prototipo de ventana de exploración. Añada un nuevo formulario al proyecto, llámelo *wndExploracion*, guarde el fichero como *Exploracion*, a secas, y asegúrese de que no esté en la lista de ventanas con creación automática. Cambie su *FormStyle* a *fsMDIChild*, y asigne *True* en su *ShowHint*.

Y ahora comenzamos a escribir código. Escribiremos un método que cree una ventana de exploración, cuando no haya ninguna otra de su mismo tipo, o que ponga esa ventana ya existente en primer plano, si logra encontrarla. Como comprenderá, se

trata de un método que a veces creará un objeto, y a veces no. No puede ser un constructor, que siempre reserva memoria, ni un método “normal”, que nunca lo hace. Lo que haremos es definir un método de clase:

```
class function TwndExploracion.Lanzar: TwndExploracion;
var
  I: Integer;
  F: TForm;
begin
  for I := Screen.FormCount - 1 downto 0 do
    begin
      F := Screen.Forms[I];
      if F.ClassType = Self then
        begin
          if F.WindowState = wsMinimized then
            F.WindowState := wsNormal;
          F.BringToFront;
          Result := TwndExploracion(F);
          Exit;
        end;
      end;
    Result := Self.Create(Application);
  end;
```

Hay que garantizar también que la ventana se destruya al ser cerrada. Para lograrlo, interceptaremos su evento *OnClose*:

```
procedure TwndExploracion.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  Action := caFree;
end;
```

## Apertura controlada de conjuntos de datos

Tenemos que tomar otra decisión importante: ¿añadimos directamente una rejilla dentro de la ventana de exploración? La alternativa sería crear un marco o *frame*, y situar un componente *TDBGrid* en su interior. En la clase del marco programaríamos las funciones y eventos comunes a todas las rejillas de navegación que usemos en el proyecto. En verdad, así es como me gustaría hacerlo, pero como se trata de un sistema más complicado, voy a meter directamente la rejilla en la propia ventana. En realidad, traeremos primero un *TDataSource*, al que llamaremos *dsBase*, y un *TDBGrid*, que enlazaremos al primer componente mediante su propiedad *DataSource*. Finalmente, cambiaremos la propiedad *Align* de la rejilla a *alClient*.

Como sabemos, *dsBase* tiene una propiedad *DataSet*, que apuntará al conjunto de datos relacionado con la ventana. Vamos a crear un manejador para el evento *OnCreate* del formulario, y ampliaremos la respuesta actual al evento *OnClose*:

```
procedure TwndExploracion.FormCreate(Sender: TObject);
var
  I: Integer;
begin
```

```

    if Assigned(dsBase.DataSet) then
        with dsBase.DataSet do
            begin
                Open;
                Tag := Tag + 1;
            end;
        for I := DBGrid.Columns.Count - 1 downto 0 do
            with DBGrid.Columns[I] do
                Title.Alignment := Alignment;
            end;
        end;

    procedure TwndExploracion.FormClose(Sender: TObject;
        var Action: TCloseAction);
    begin
        Action := caFree; // Ya estaba en el paso anterior
        if Assigned(dsBase.DataSet) then
            with dsBase.DataSet do
                begin
                    if Tag = 1 then Close;
                    Tag := Tag - 1;
                end;
            end;
    end;
end;

```

Lo que hemos hecho es controlar la apertura del conjunto de datos asociado al *TDataSource* mediante un contador de referencias, que mantendremos en la propiedad *Tag* del conjunto de datos que sea. Observe, además, que al crear la ventana he dedicado un par de líneas para que los títulos de las columnas de la rejilla tengan la misma alineación que el contenido de la columna.

## Asociando una ventana de edición

Espero que no haya perdido de vista el objetivo de la ventana anterior: a partir de ella definiremos las ventanas “concretas” de exploración, como *wndClientes* y *wndProductos*. Debe recordar que queremos asociar un diálogo de detalles y edición a cada una de ellas, de manera que con un doble clic sobre la rejilla, se cree el diálogo correspondiente y se ejecute en forma modal. Vaya entonces al editor de código, y añada las siguientes declaraciones a la clase de la ventana de exploración general:

```

type
    TwndExploracion = class(TForm)
        // ...
        procedure DBGridDblClick(Sender: TObject);
    public
        Editor: TFormClass;
        procedure Insertar; virtual;
        procedure Modificar; virtual;
        // ...
    end;

```

La variable *Editor* ha sido definida mediante el tipo *TFormClass*, que sirve para almacenar referencias de clases. Esta es su declaración, en la unidad *Forms*:

```

type
    TFormClass = class of TForm;

```

Cada vez que definamos por herencia una ventana derivada concreta, tendremos que interceptar su evento *OnCreate* para asignar la clase de editor asociada. Por ejemplo, para *wndClientes* el editor donde veremos los detalles de un cliente será un formulario de la clase *dlgCliente*, por lo que trataremos el evento *OnCreate* de *wndClientes* así:

```
procedure TwndClientes.FormCreate(Sender: TObject);
begin
    inherited;
    Editor := TdlgCliente;
end;
```

Hemos definido dos métodos públicos virtuales, *Insertar* y *Modificar*. La implementación de *Insertar* debe ser:

```
procedure TwndExploracion.Insertar;
begin
    if Assigned(Editor) then
    begin
        dsBase.DataSet.Insert;
        Editor.Create(Self).ShowModal;
    end;
end;
```

Primer comprobamos que no se nos ha olvidado asignar una clase de edición a la ventana de navegación. Si está todo bien, ponemos al conjunto de datos asociado en modo de inserción, creamos una instancia del editor, e inmediatamente la mostramos en forma modal. Esto es muy importante: no se destruye explícitamente la instancia del editor, porque asumiremos que todos los cuadros de edición tienen programada la destrucción automática, a través de *OnClose*. Por su parte, *Modificar* es similar:

```
procedure TwndExploracion.Modificar;
begin
    if Assigned(Editor) then
    begin
        dsBase.DataSet.Edit;
        Editor.Create(Self).ShowModal;
    end;
end;

procedure TwndExploracion.DBGridDblClick(Sender: TObject);
begin
    Modificar;
end;
```

La respuesta a un doble clic sobre la rejilla consiste en llamar directamente a *Modificar*, para mostrar los detalles del registro activo. Observe que, tanto en el caso de *Insertar* como en el de *Modificar*, no nos preocupamos por devolver el conjunto de datos al estado *dsBrowse*. Estamos asumiendo también que el diálogo será responsable de este asunto.

## Automatizando la entrada de datos

Ahora programaremos el prototipo de todos los cuadros de diálogo de actualización. Cree un nuevo formulario dentro del proyecto, llámelo *wndDialogo*, asegúrese de que Delphi no lo instancie automáticamente al iniciar la ejecución, y guarde la unidad con el nombre de *Dialogo*. Asigne *True* a *ShowHint*, *poScreenCenter* a *Position*, y *bsDialog* a *BorderStyle*. Traiga también un *TDataSource*, y bautícelo como *dsBase*. Seleccione entonces el código de la declaración de ventana, y modifíquelo de este modo:

```
type
  TwndDialogo = class(TForm)
    dsBase: TDataSource;
    procedure FormClose(Sender: TObject;
      var Action: TCloseAction);
    procedure FormCloseQuery(Sender: TObject;
      var CanClose: Boolean);
  protected
    procedure Guardar; virtual;
    procedure Cancelar; virtual;
    function Modificado: Boolean; virtual;
    function Confirmar: Boolean; virtual;
  end;
```

Primero trataremos el evento *OnClose*, para garantizar la destrucción automática:

```
procedure TwndDialogo.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  Action := caFree;
end;
```

Pero la parte verdaderamente importante es la implementación de la respuesta al evento *OnCloseQuery* que, como recordará, se dispara antes de cerrar una ventana, para preguntarle al programador si puede cerrarla o no:

```
procedure TwndDialogo.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
begin
  if Assigned(dsBase.DataSet) then
    if ModalResult = mrOk then
      Guardar
    else if not Modificado or Confirmar then
      Cancelar
    else
      CanClose := False;
end;
```

Como puede ver, este algoritmo es suficientemente general como para acomodar cualquier técnica de grabación de datos. Además, todos los métodos que utiliza han sido declarados virtuales, por lo que si surge alguna excepción en el tratamiento de las modificaciones, puede que no sea necesario modificar todo el algoritmo, sino que baste con redefinir la implementación de alguno de los métodos virtuales.

Veamos primero cómo averiguamos si el registro activo ha sido modificado o no:

```
function TwndDialogo.Modificado: Boolean;
begin
    Result := dsBase.DataSet.Modified;
end;
```

Fácil, ¿verdad? Esta es la implementación de *Grabar*:

```
procedure TwndDialogo.Grabar;
begin
    dsBase.DataSet.CheckBrowseMode;
end;
```

Quizás usted haya pensado que iba a utilizar *Post* directamente. El problema de *Post* es que exige como condición que el conjunto de datos se encuentre en alguno de los modos de edición; si no se cumple este requisito, lanza una excepción. *CheckBrowseMode*, en cambio, verifica el estado del conjunto de datos, e incluso comprueba si hay modificaciones, antes de llamar a *Post*.

Por suerte, *Cancel* no es tan quisquilloso, y podemos programar *Cancelar* de esta simple manera:

```
procedure TwndDialogo.Cancelar;
begin
    dsBase.DataSet.Cancel;
end;
```

Por último, he aquí la implementación del método de confirmación de abandono de los cambios:

```
resourcestring
    SAbandonarCambios = 'Este registro ha sido modificado'#13 +
                        '¿Desea abandonar los cambios?';

function TwndDialogo.Confirmar: Boolean;
begin
    Result := MessageDlg(SAbandonarCambios, mtConfirmation,
                        [mbYes, mbNo], 0) = mrYes;
end;
```

Observe que he utilizado *MessageDlg*, y que he declarado el mensaje mediante una sección **resourcestring**. Mi propósito ha sido facilitar la traducción posterior de la aplicación a otros idiomas.

## La ventana principal

Regresamos a la ventana principal, para implementar los comandos generales de navegación y edición. En primer lugar, ¿recuerda que hemos traído dos barras de herramienta para esa ventana? En la primera pondremos las acciones que *siempre* estarán disponibles: crear ventanas de navegación para ciertas entidades, organizar las

ventanas, configurar la impresora, etc. Y en la segunda barra irán los comandos que solamente tienen sentido cuando hay alguna ventana de navegación abierta. Claro, lo que pretendo es esconder la segunda barra de herramientas cuando no la necesitamos. Traiga a la ventana un componente *TApplicationEvents*, de la página *Additional* de la Paleta de Componentes, e intercepte su evento *OnIdle*:

```
procedure TwndPrincipal.ApplicationEvents1Idle(Sender: TObject;
    var Done: Boolean);
var
    W: TForm;
begin
    W := ActiveMDIChild;
    DataBar.Visible := Assigned(W) and (W is TwndExploracion);
end;
```

Se me olvidaba decirle que he llamado *DataBar* a la segunda barra de herramientas. Tendrá además que incluir la unidad *Exploracion* en la cláusula **uses** de la ventana principal, para que ésta sepa qué cuernos es el tipo *TwndExploracion*.

No sé si me hizo caso y creo las cuatro acciones predefinidas de navegación; si no lo hecho, mueva... el ratón y tráigalas de una vez. Puede crear entonces cuatro botones en la *DataBar* para esas cuatro acciones. Ya hemos visto que el mecanismo predefinido de localización de objetivos de las acciones predefinidas lograrán que esos botones actúen automáticamente sobre la rejilla activa de la ventana activa. Esta característica nos ahorrará mucho código.

Pero debemos todavía definir e implementar acciones globales para insertar, modificar y eliminar registros de la ventana de navegación activa. Comencemos por la acción global de modificación:

```
procedure TwndPrincipal.ModificarUpdate(Sender: TObject);
var
    W: TForm;
begin
    W := ActiveMDIChild;
    TAction(Sender).Enabled :=
        Assigned(W) and (W is TwndExploracion) and
        not TwndExploracion(W).dsBase.DataSet.IsEmpty;
end;
```

El método anterior corresponde al evento *OnUpdate* de la acción *Modificar*. Activamos la acción cuando hay una ventana hija MDI activa, que pertenece a una clase derivada de *TwndExploracion*, y cuando el conjunto de datos asociado no está vacío. La respuesta a *OnExecute* es aún más sencilla, porque podemos asumir ya que hay una ventana activa de un tipo derivado de *TwndExploracion*:

```
procedure TwndPrincipal.ModificarExecute(Sender: TObject);
begin
    (ActiveMDIChild as TwndExploracion).Modificar;
end;
```

Ese *Modificar* que se llama ahora es el método público virtual que definimos en la clase *TwndExploracion*.

Muy parecido será el tratamiento de los eventos de *Insertar*:

```

procedure TwndPrincipal.InsertarUpdate(Sender: TObject);
var
    W: TForm;
begin
    W := ActiveMDIChild;
    TAction(Sender).Enabled := Assigned(W) and
        (W is TwndExploracion);
end;

procedure TwndPrincipal.InsertarExecute(Sender: TObject);
begin
    (ActiveMDIChild as TwndExploracion).Insertar;
end;

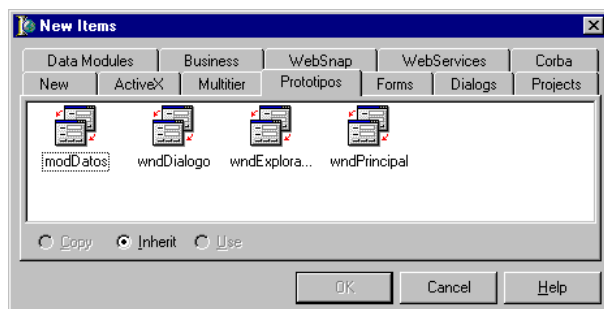
```

La única diferencia consiste en que no se exige que haya un registro en el conjunto de datos activo.

## De lo sublime a lo concreto

Sólo nos queda crear las ventanas concretas de acuerdo a los conjuntos de datos que pongamos en el módulo de datos. Si lo desea, puede guardar una copia del proyecto en su estado actual.

Active el módulo de datos y traiga un *TClientDataSet*. Asigne en su propiedad *File-Name* el fichero *parts.xml* y llame *Productos* al componente. Cuando cree los objetos de acceso a campo, verá que una de las columnas es *VendorNo*, y contiene un código de proveedor. Traiga también un conjunto de datos para el fichero *vendors.xml*, y aprovechélo para definir un campo de búsqueda basado en el código de proveedor.



Para crear la ventana de navegación sobre productos, ejecute el comando de menú *File|New|Other*, para que aparezca el Almacén de Objetos. Vaya a la cuarta página, localice el icono del prototipo de ventana de exploración, y haga doble clic. Entonces Delphi creará un nuevo formulario derivado de *TwndExploracion*. Cambie su nombre a *wndProductos*, añada la unidad *Datos* a su cláusula **uses**, enlace la propiedad *DataSet*



del componente *dsBase* a la tabla de productos del módulo de datos, haga doble clic sobre la rejilla y traiga y configure las columnas que más rabia le den. Todo esto se termina de hacer en menos tiempo de lo que lleva escribirlo. Finalmente, guarde la unidad de la ventana con el nombre de *Productos*.

Debemos crear, a continuación, un diálogo para editar los datos de un producto. Vuelva al Almacén de Objetos, pero haga doble clic esta vez sobre la ventana *Twnd-Dialogo*. Llame *dlgProducto* al nuevo formulario, y guarde la unidad con el nombre de *ProductoDlg*. Enlace la nueva unidad con el módulo de datos y asigne un puntero a la tabla de productos en la propiedad *DataSet* de *dsBase*. Para terminar, traiga controles de acceso a datos al formulario, hasta que tenga más o menos este aspecto:

Ahora hay que vincular el diálogo de productos con la ventana de navegación sobre productos. Regresamos a *wndProductos*, y creamos un método en respuesta al evento *OnCreate* del formulario:

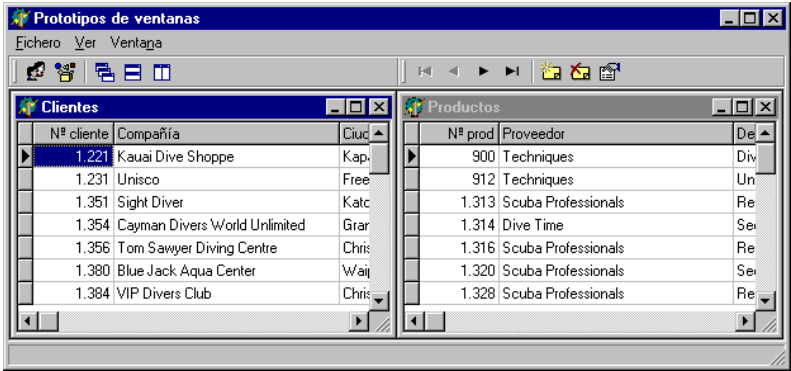
```
procedure TwndProductos.FormCreate(Sender: TObject);
begin
    inherited;
    Editor := TdlgProducto;
end;
```

Finalmente, seleccione la ventana principal, añada la unidad *Productos* a la cláusula **uses** de ésta, cree una acción para lanzar la ventana de productos e implemente la respuesta a su evento *OnExecute* como muestro a continuación:

```
procedure TwndPrincipal.VerProductosExecute(Sender: TObject);
begin
    TwndProductos.Lanzar;
end;
```

¿Se ha dado cuenta que solamente hemos necesitado dos instrucciones para cada tabla que utilizemos en el proyecto. Repita estos pasos con la tabla *customer.xml*, que contiene un listado de clientes. Verá que la mayor parte del tiempo se consume en configurar las propiedades de los campos, especialmente la omnipresente *DisplayLabel*, y en distribuir los controles de acceso a datos en el cuadro de diálogo de edición. La única forma de ahorrar tiempo en estas tareas es disponer de asistentes que hagan por nosotros la parte mecánica del trabajo. Pero esa es otra historia, para ser contada en otro momento.

Y esta es la apariencia final de nuestra pequeña aplicación de muestra:



Puede probar a añadir nuevas ventanas de mantenimiento, por ejemplo, sobre la tabla de empleados. O puede definir nuevas acciones globales, como la posibilidad de deshacer los últimos cambios.

# 4

## **Interfaces de acceso a SQL**

---

- **DB Express: conexiones**
- **DB Express: conjuntos de datos**
- **El Motor de Datos de Borland**
- **Acceso a datos con el BDE**
- **BDE: descenso a los abismos**
- **InterBase Express**
- **ADO y ADO Express**

# Parte



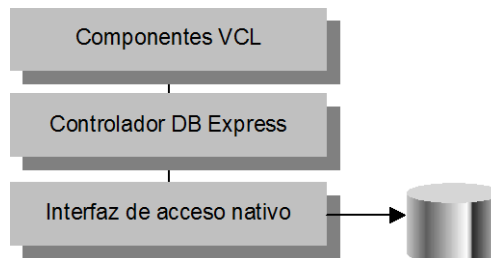
## DB Express: conexiones

**L**A PRIMERA INTERFAZ DE ACCESO A DATOS SQL que estudiaremos será DB Express. Con este nombre nos referiremos tanto a los controladores específicos que acceden a la interfaz nativa SQL, como a los componentes VCL generales que hacen uso de ellos. El gran atractivo de DB Express es su sencillez: su interfaz de programación es la mínima exigible en una interfaz de acceso a datos. No porque se obtenga mayor rapidez en la ejecución de comandos y recuperación de registros, que es también posible, sino porque mientras más sencillo es un sistema, menos oportunidades tienen los humanos que lo diseñan y mantienen de meter la pata. Así de claro.

El precio que se paga por la sencillez es la dificultad de construir una aplicación medianamente grande con acceso a datos utilizando solamente este sistema. Pero la respuesta consiste en combinar los conjuntos de datos unidireccionales de DB Express con los conjuntos de datos clientes que ya conocemos. En este capítulo nos limitaremos a presentar los componentes DB Express, y dejaremos para más adelante la explicación sobre cómo enlazarlos con nuestros *TClientDataSets*.

### La estructura de DB Express

El diagrama que aparece a continuación muestra, a grandes rasgos, la estructura de DB Express. Tenga presente que todos estos módulos se ubican en el lado cliente de la aplicación.

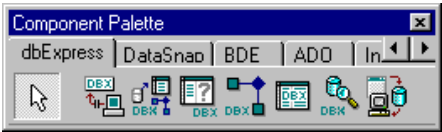


- 1 Tenemos, en primer lugar, una interfaz de acceso nativo, que es la que realmente se comunica con el servidor SQL. Esa capa no es parte de DB Express, pero la

menciono porque existe la tendencia a olvidarse de ella, y en algunos casos incluso a suponer que no es necesaria.

- 2 DB Express comienza en esta capa, que consiste en un módulo DLL que se conecta al servidor SQL a través de la interfaz nativa antes mencionada. Por cada sistema soportado, existe un módulo diferente, pero todos ellos ofrecen una misma interfaz pública.
- 3 En esta capa final, se sitúan varios componentes VCL que se ocupan de realizar conexiones con la base de datos, de la ejecución de instrucciones SQL y de la recuperación de datos de consultas a través de cursores unidireccionales. Por supuesto, esta capa está contenida íntegramente dentro de Delphi.

Los componentes de DB Express se encuentran en la página *dbExpress* de la Paleta de Componentes, como era de esperar:



En este capítulo nos ocuparemos fundamentalmente del primero de ellos, el componente *TSQLConnection*. Los restantes componentes serán presentados en el siguiente.

Los controladores

Los controladores que vienen con la versión 6 de Delphi son los siguientes:

Controlador	Sistema	Versiones de Delphi
<i>dbexpint.dll</i>	InterBase	Professional/Enterprise
<i>dbexpmys.dll</i> <sup>22</sup>	MySQL	Professional/Enterprise
<i>dbexpora.dll</i>	Oracle	Enterprise
<i>dbexpdb2.dll</i>	DB2	Enterprise

Estoy hablando de los controladores desarrollados por Borland. En principio, existe una especificación más o menos formal de la interfaz obligatoria de estos controladores, y es probable que pronto aparezcan controladores de terceros para otros sistemas de acceso a datos.

Naturalmente, no hace falta que todos ellos estén presentes para el funcionamiento de una aplicación de base de datos. A diferencia de otras interfaces de acceso, tampoco existe un controlador adicional con el núcleo del sistema: los componentes DB

<sup>22</sup> Tenga cuidado, porque parte de la documentación se refiere a este controlador con el nombre de *dbexpMy.dll*, sin incluir la 's' final del nombre.

Express se conectan directamente al controlador que corresponde al formato de la base de datos, sin requerir ningún otro fichero como intermediario.

Hay dos posibilidades para distribuir una aplicación DB Express: incorporar los ficheros de los controladores dentro del propio ejecutable de la aplicación, o distribuirlos por separado. Si usted va a distribuir actualizaciones de su aplicación con cierta frecuencia, quizás le interese mantener separada la aplicación de los controladores, para que los parches ocupen menos espacio. Por el contrario, si desconfía de la capacidad intelectual de su cliente, es mejor que distribuya un solo ejecutable para evitar problemas de soporte.

Si distribuye las DLLs por separado, puede instalarlas en cualquier directorio donde pueda encontrarlas su aplicación: puede ser el directorio de la propia aplicación, el directorio de sistema de Windows o cualquier otro directorio que se mencione en la ruta de búsqueda del sistema operativo. Personalmente, prefiero utilizar el directorio de sistema en esos casos. Si tengo que instalar varias aplicaciones, al menos tengo la tranquilidad de que hay una sola copia del controlador en el ordenador del cliente.

La instalación de Delphi, por su parte, copia inicialmente los ficheros de los controladores en el directorio *delphi6\bin*.

## Ficheros de configuración

Si ya ha programado con el BDE, pero es la primera vez que se enfrenta a DB Express, esta sección será probablemente la más importante para usted, porque aclara una confusión frecuente. DB Express se diseñó teniendo bien presente los errores en el diseño del BDE, para no volverlos a cometer. Una de las quejas más escuchadas respecto a este sistema era lo complicado que era instalarlo y configurarlo. Había que lidiar con entradas en el registro, ficheros de configuración con formatos esotéricos, y con multitud de módulos de los que solamente se podía sospechar su función.

Para nuestra tranquilidad, la instalación y configuración de DB Express son mucho más sencillas. Como mencioné en la sección anterior, podemos incluso incorporar el controlador que vayamos a utilizar dentro del ejecutable. Tampoco es necesario realizar algún tipo de configuración externa, ya sea en el registro o en un fichero *ini*. Cuando el programador escucha todo esto, cree que la vida es bella y el cielo es azul... hasta que se pone a trajar en el registro y descubre una clave asociada a DB Express que apunta a unos ficheros *ini* de configuración.

Bien, siguen sin existir motivos de alarma: esa clave y esos ficheros son utilidades que se emplean en tiempo de diseño por los componentes DB Express. En tiempo de ejecución podemos prescindir de todas ellas, pero también podemos dejarlas para añadir flexibilidad a nuestras aplicaciones.

Comencemos por la clave del registro. Se encuentra repetida en las siguientes rutas:

```
HKEY_LOCAL_MACHINE/Software/Borland/DB Express
HKEY_CURRENT_USER/Software/Borland/DB Express
```

Por supuesto, la que se aplica en la que está asociada al usuario activo, y la copia asociada al ordenador local se utiliza como valor por omisión cuando se conecta un nuevo usuario. Dentro de ese nodo encontrará dos parámetros:

Parámetro	Valor
<i>Driver Registry File</i>	<i>dbxdrivers.ini</i>
<i>Connection Registry File</i>	<i>dbxconnections.ini</i>

En ambos casos he omitido el directorio donde se encuentran los ficheros, que por omisión es el siguiente:

```
C:\Archivos de programa\Archivos comunes\Borland Shared\DBExpress
```

Recuerde, sin embargo, que los controladores no se instalan en ese directorio, sino en el directorio *bin* de Delphi.

Busque el fichero *dbxdrivers.ini* y ábralo para examinar su contenido. Dentro de él encontraremos distintas clases de parámetros. Por ejemplo, el primer grupo con el que tropezaremos es el siguiente:

```
[Installed Drivers]
DB2=1
Interbase=1
MYSQL=1
Oracle=1
```

Naturalmente, estoy escribiendo este libro con la versión Enterprise, que incluye los cuatro controladores anteriores. A continuación hay una sección de parámetros para cada uno de los controladores mencionados. Estos son los que corresponden a InterBase:

```
[Interbase]
GetDriverFunc=getSQLDriverINTERBASE
LibraryName=dbexpint.dll
VendorLib=GDS32.DLL
BlobSize=-1
CommitRetain=False
Database=database.gdb
ErrorResourceFile=
LocaleCode=0000
Password=masterkey
RoleName=RoleName
ServerCharSet=
SQLDialect=1
Interbase TransIsolation=ReadCommitted
User_Name=sysdba
WaitOnLocks=True
```

Los tres primeros parámetros son diferentes a los restantes. Cuando creamos una conexión DB Express en Delphi, estos valores van a parar a tres propiedades del



componente de conexión, y son utilizados para cargar en memoria, si fuese necesario, el módulo DLL correspondiente al controlador (*LibraryName*), para encontrar la función que inicializa el controlador (*GetDriverFunc*), y para saber cuál es la DLL de la interfaz de acceso nativo que hay que cargar (*VendorLib*). El resto de los parámetros determinan el modo de funcionamiento particular de ese controlador. Por ejemplo, cuando el parámetro *WaitOnLocks* vale *False* y una transacción detecta un bloqueo sobre un registro que tiene que modificar, se produce el error inmediatamente; en cambio, cuando vale *True*, que es el valor por omisión, el cliente espera pacientemente hasta que termine la transacción que ha causado el bloqueo.

¿Y cómo podemos saber, tanto nosotros como Delphi, que ese parámetro *WaitOnLocks* acepta los valores *False* y *True*? Encontraremos la respuesta un poco más adelante, dentro del mismo fichero:

```
[WaitOnLocks]
False=1
True=0
```

En resumen, dentro del fichero de conexiones encontramos varias clases de parámetros, que paso a detallar:

- 1 Los controladores instalados en ese ordenador.
- 2 Para cada controlador, los tres primeros parámetros indican cómo utilizar el controlador para establecerse una conexión.
- 3 Los restantes parámetros del controlador indican qué parámetros podemos configurar, y cuál es el valor más adecuado inicialmente.
- 4 Finalmente, para los parámetros que aceptan valores de un conjunto finito, hay secciones que asocian valores simbólicos a cada uno de los valores posibles.

¿Se da cuenta de que podemos vivir sin los servicios de este fichero, una vez que sepamos cómo establecer y configurar una conexión?

Abra ahora el otro fichero, *dbxconnections.ini*. Este guarda datos sobre conexiones a bases de datos concretas, y esas conexiones son el equivalente aproximado a los alias del BDE. No hace falta que se salte un par de capítulos, si no ha trabajado con el BDE, para averiguar que son esos “alias”, porque vamos a ver un ejemplo ahora mismo.

Tomemos como punto de partida la conexión identificada como *IBLocal*. Estos son sus parámetros y sus valores:

```
[IBLocal]
BlobSize=-1
CommitRetain=True
Database=c:\archivos de programa\...\data\mastsql.gdb
DriverName=Interbase
ErrorResourceFile=
LocaleCode=0000
Password=masterkey
```

```

RoleName=RoleName
ServerCharSet=
SQLDialect=1
Interbase TransIsolation=RepeatableRead
User_Name=sysdba
WaitOnLocks=True

```

Los nombres de estos parámetros son los mismos que aparecen asociados al controlador en *dbxdrivers.ini*, quitando los tres parámetros especiales de localización y carga, y añadiendo el parámetro *DriverName* para indicar cuál es el controlador apropiado para la conexión. Esta, en particular, hace referencia a un fichero local llamado *mastsql.gdb*<sup>23</sup>, lo que se interpreta como que deseamos utilizar un servidor local de InterBase. Tenemos dos alternativas para definir la conexión:

- 1 Ignorar completamente el fichero de conexión y almacenar internamente todos los parámetros necesarios. Dentro de poco veremos cómo controlar este comportamiento con la propiedad *LoadParamsOnConnect* del componente de conexión de DB Express. Podríamos también inicializar las propiedades que almacenan la configuración con ayuda de *dbxconnections.ini* para ignorarlo a partir de ese momento.
- 2 Dejar el valor *True* en la propiedad *LoadParamsOnConnect* del componente de conexión, para que cada vez que se ejecute la aplicación, se lea el contenido de *dbxconnections.ini*, o del fichero al que apunte la clave del registro que hemos mencionado antes.

Si nos decidimos por la segunda técnica, podríamos cambiar la ubicación de la base de datos una vez que esté en explotación la aplicación, sin necesidad de retocar el código fuente de la misma. Ganaríamos en flexibilidad, que era lo que explicaba al principio del capítulo... pero a costa de tener que preocuparnos por el mantenimiento de un fichero adicional y una clave de registro.

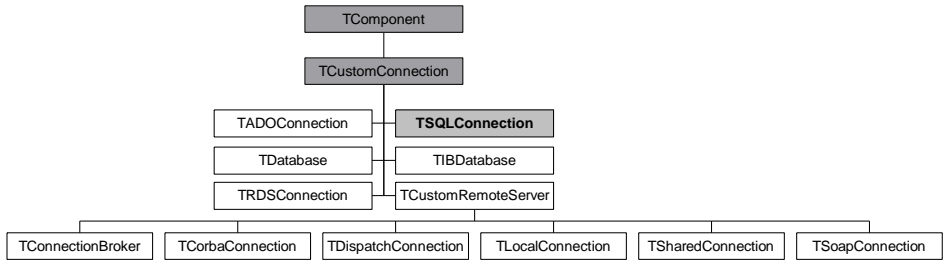
## Conexiones, en general

Ya podemos concentrarnos en los componentes que ofrece DB Express. Comenzaremos por *TSQLConnection*, el que nos permite establecer las conexiones con nuestras bases de datos<sup>24</sup>. El componente de conexión presume de tener un distinguido árbol genealógico, como nuestro a continuación:

---

<sup>23</sup> He abreviado la ruta de la base de datos para no dividir la línea.

<sup>24</sup> Bueno, también con las ajenas...



Como puede comprobar, algunos de estos componentes han tenido una vida sexual muy agitada, que no es muy coherente con eso de la herencia simple. En la foto de familia, nuestro héroe, el *TSQLConnection*, es ese niño cabezón tan serio, con una gorra gris más grande aún que su cabeza, que aparece a la derecha, en la tercera fila contando desde arriba.

El lector imaginará que si he traído toda esta familia a colación será porque deben existir propiedades, métodos y eventos comunes a todos ellos. Tiene razón. Con las conexiones sucede lo mismo que con las honorables ciudadanas de la República Popular China y las pelis pornos: que cuando conoces una, ya has visto todas.

Por ejemplo, todas las conexiones se conectan y desconectan, que para eso se llaman así. En ese mecanismo intervienen los siguientes recursos:

```

property TCustomConnection.Connected: Boolean;
procedure TCustomConnection.Open;
procedure TCustomConnection.Close;

```

Al igual que vimos que sucedía con los conjuntos de datos, *Open* y *Close* son métodos de conveniencia basados en la propiedad *Connected*. Del mismo modo, se utiliza un mecanismo similar al de *TDataSet* para la activación de conexiones durante la lectura del componente desde un recurso *dffm*. Lo resumo en pocas líneas: si dejamos una conexión abierta en tiempo de diseño, al ejecutarse la aplicación y crearse el formulario o módulo de datos que la contiene, se intentará asignar *True* en la propiedad *Connected* de la conexión. Pero el componente ha sido programado para que, en ese único caso, el valor vaya a parar a la propiedad *StreamedConnected*, no a *Connected*. Cuando se termine la carga de todos los demás componentes, se disparará el método *Loaded* de cada uno de ellos. El que corresponde a la conexión ha sido programado para que copie entonces el valor almacenado en *StreamedConnected* en la propiedad *Connected* y establezca realmente la conexión.

También hay eventos relacionados con la apertura y cierre de las conexiones:

```

property TCustomConnection.BeforeConnect: TNotifyEvent;
property TCustomConnection.AfterConnect: TNotifyEvent;
property TCustomConnection.BeforeConnect: TNotifyEvent;
property TCustomConnection.AfterConnect: TNotifyEvent;
property TCustomConnection.OnLogin: TLoginEvent;

```

¿Ve que existe un evento *OnLogin*, definido en la clase base *TCustomConnection*? Al parecer, es el que permite que personalizemos la petición del usuario y la contraseña de la conexión. ¡No le haga caso, que es un señuelo engañabobos! El tipo de “ese” *TLoginEvent* es el siguiente:

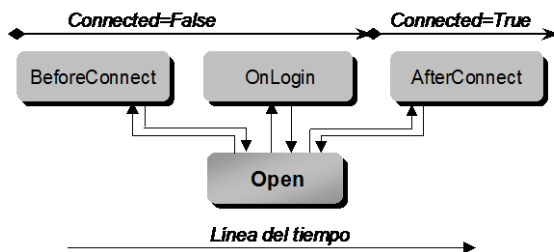
```
type
  TLoginEvent = procedure (Sender:TObject;
    Username, Password: string) of object;
```

Sin embargo, casi todas las interfaces de acceso lo ignoran e implementan un esquema de petición de parámetros propio. Veremos, por ejemplo, que DB Express define su propio *OnLogin* con este otro tipo de datos:

```
type
  TSQLConnectionLoginEvent = procedure (Database: TSQLConnection;
    LoginParams: TStrings) of object;
```

Y el BDE, para no ser menos, hace lo mismo. Se palpan las buenas vibraciones y la excelente comunicación entre los equipos de trabajo de Borland... Veremos, además, que con Delphi 6 ha cambiado el funcionamiento de *OnLogin* y de la propiedad asociada *LoginPrompt*. Debe tener mucho cuidado con este cambio, porque puede que alguna aplicación escrita en Delphi 5 o una versión anterior, deje de funcionar misteriosamente.

De todos modos, cuando los astros son favorables y se dispara *OnLogin*, la secuencia de disparo de eventos es la siguiente:



Es decir, *BeforeConnect* tiene lugar antes de *OnLogin*, y ambos se disparan antes de que se establezca la conexión. Si *OnLogin* es un buen momento para modificar parámetros relacionados con el usuario y la contraseña, *BeforeConnect* es más apropiado para cualquier otro cambio de configuración.

## La contabilidad de las conexiones

Otra área de funcionalidad común a todas las clases de conexiones es la que mantiene las cuentas sobre cuántos conjuntos de datos están asociados a la conexión:

```
property TCustomConnection.DataSetCount: Integer;
property TCustomConnection.DataSets[Index: Integer]: TDataSet;
```

La clase *TSQLConnection* redefine *DataSets* para que sus elementos pertenezcan a la clase *TCustomSQLDataSet*. Pero la nueva propiedad, en definitiva, se limita a llamar a la implementación genérica, y a forzar una conversión de tipos, por comodidad para el programador.

Hay una diferencia importante entre las conexiones del BDE, para las que se utiliza *TDatabase*, y las que efectúa DB Express con *TSQLConnection*. En las primeras, solamente se mantienen las cuentas sobre los conjuntos de datos abiertos, mientras que la conexión de DB Express almacena referencias a todos los conjuntos de datos asociados, estén abiertos o cerrados; bastan con que la mencionen en su propiedad *SQLConnection*. Más bien, deberíamos decir que el BDE actúa como la oveja negra, porque ADO e InterBase Express se comportan igual que DB Express.

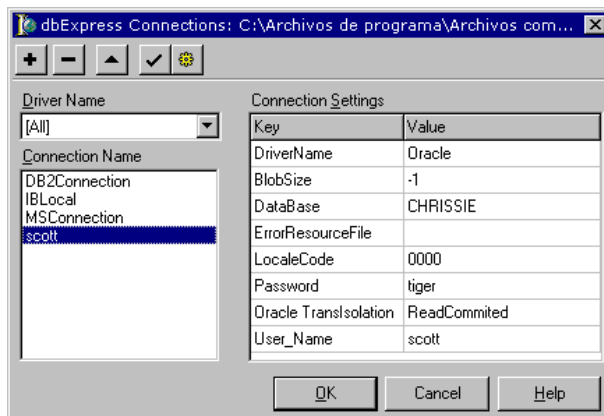
Internamente, los conjuntos de datos son los que piden a su conexión que los tenga en cuenta. Para este propósito, *TCustomConnection* ofrece los siguientes métodos protegidos:

```
procedure TCustomConnection.RegisterClient(Client: TObject;
    Event: TConnectChangeEvent = nil); virtual;
procedure TCustomConnection.UnRegisterClient(Client: TObject);
virtual;
```

Note que el método de registro permite pasar un puntero a un método del conjunto de datos, para que la conexión lo ejecute cuando se active o desactive. En la práctica, solamente ADO, en un caso muy especial, aprovecha esta oportunidad.

## El componente *TSQLConnection*

¡Finalmente!, vamos a conectarnos a una base de datos a través de DB Express. Inicie una nueva aplicación, y deje caer un *TSQLConnection*, de la página *dbExpress*, sobre la ventana principal. Haga doble clic sobre el componente, para que aparezca el editor asociado al mismo:



En la parte izquierda de la ventana, justo debajo de la barra de herramientas, hay un combo con la lista de controladores registrados en *dbxdrivers.ini*, y una entrada adicional, *All*, que es la que se muestra activa por omisión. Ese combo actúa como filtro de la lista de nombres de conexiones que tiene debajo; esta lista, a su vez, se obtiene del fichero *dbxconnections.ini*.

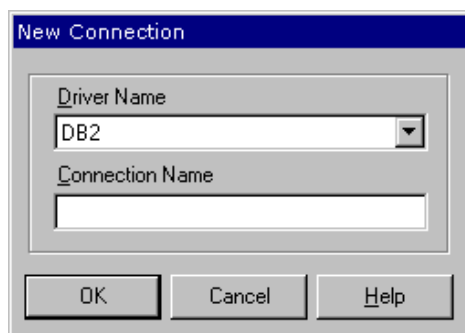
La forma más sencilla de configurar el componente es seleccionar una de las conexiones definidas en la lista y pulsar *OK*. En tal caso, se producen los siguientes cambios en las propiedades del componente de conexión:

- 1 *ConnectionName* recibe el nombre de la conexión seleccionada.
- 2 *DriverName*, *GetDriverFunc*, *LibraryName* y *VendorLib* reciben los valores que corresponden al tipo de controlador al que pertenece la conexión.
- 3 En *Params*, que es una lista de cadenas, se copian los restantes parámetros.

Pero es muy poco probable que quede satisfecho con los valores de prueba de las conexiones que vienen como ejemplo. Puede modificar esos parámetros en dos formas diferentes:

- 1 Si efectúa un doble clic nuevamente, y modifica los valores de los parámetros en el editor del componente, los cambios quedan reflejados tanto en el fichero *dbxconnections.ini* como en la propiedad *Params*. Cuando edite algunos de los parámetros, podrá elegir valores gracias a una lista desplegable que se construye a partir de la información del fichero global de controladores.
- 2 En cambio, si edita directamente sobre *Params*, no se modifica el fichero de configuración global, pero no contará con la ayuda de las listas desplegables con los valores admitidos para cada parámetro.

Como era de esperar, también se nos permite crear una nueva conexión con nombre. Podemos hacerlo desde el editor de la conexión, pulsando el botón que tiene el signo de adición:



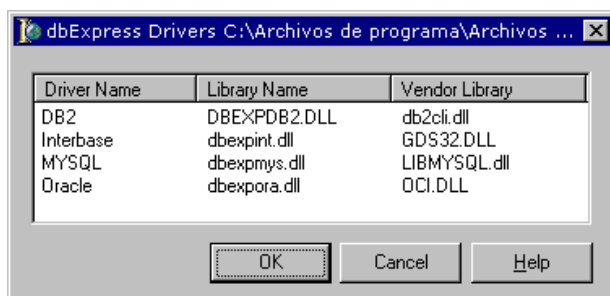
Al crear la conexión, se crea también la lista correspondiente de parámetros y se les asignan los valores por omisión que corresponden al tipo de controlador. Por su-

puesto, cuando cierre el cuadro de diálogo, la nueva conexión habrá sido almacenada en *dbxconnections.ini*.

Si no quiere que la conexión se grabe en el fichero global, puede armarse de valor y cambiar directamente los parámetros en el Inspector de Objetos. Cuando cambie el nombre de la conexión, el componente borrará inmediatamente el valor almacenado en *Params*. Lo mejor en este caso es copiar al portapapeles una lista de parámetros de ejemplo, quizás desde otra conexión o desde el propio fichero *ini*, y modificarlos después *in situ*.

Hay más botones en la barra de herramientas: uno para eliminar una conexión registrada en el fichero global, otro para renombrar una conexión existente. Y mi favorito es el botón que permite comprobar si hemos tecleado bien los parámetros intentando una conexión. Por cierto, sepa que al probar la conexión está simplemente asignando *True* en la propiedad *Connected* del componente. Al cerrar el cuadro de diálogo se mantendrá abierta la conexión.

Por último, ¿ve aquel botón que tiene una imagen parecida a una rueda dentada? Tiene dos funciones. La primera es indicar el terrible mal gusto de Borland para las imágenes de los botones. La segunda es mostrar en un cuadro de diálogo emergente la lista de controladores instalados, el nombre de las DLLs asociadas al controlador y el nombre de los módulos correspondientes a la interfaz nativa de acceso.



Por supuesto, toda esta información tiene al fichero *dbxdrivers.ini* como origen.

## Parámetros de conexiones

Para poder continuar, necesitaremos saber qué significa cada parámetro de cada tipo de controlador. Veamos primero los parámetros que se utilizan en una conexión con InterBase:

```
DriverName=Interbase
BlobSize=-1
CommitRetain=False
Database=c:\archivos de programa\archivos comunes\...\mastsq1.gdb
ErrorResourceFile=
LocaleCode=0000
```

```

Password=masterkey
RoleName=RoleName
ServerCharSet=
SQLDialect=1
Interbase TransIsolation=ReadCommitted
User_Name=sysdba
WaitOnLocks=True

```

El más evidente de todos es *DriverName*, que no requiere explicación. Para el resto no voy a seguir el orden en que aparecen mencionados, sino que los agruparé según su propósito.

Tenemos, en primer lugar, a *Database*: su sintaxis varía de acuerdo al controlador, y en InterBase hace mención simultáneamente al servidor, al protocolo y al fichero de InterBase. Si se trata de una conexión a una instancia local del servidor de InterBase, basta con poner el nombre del fichero, como en el ejemplo. Si vamos a conectarnos mediante TCP/IP debemos mencionar el nombre del servidor al principio:

```
Database=christine:d:/databases/customerData.gdb
```

Tenga en cuenta que para aplicaciones CGI o ISAPI, es preferible que nos conectemos al servidor local mediante TCP/IP, especialmente si el sistema operativo es Windows NT o 2000.

Para completar la identificación necesaria durante el establecimiento de la conexión, debemos proporcionar *User\_name*, para el nombre de usuario, *Password*, con propósito evidente, y *RoleName*, para que opcionalmente asumamos una “función”, “papel” o como demonios quiera traducir *role*. Recuerde que al indicar un *role* durante la conexión estamos sumándonos los derechos de acceso que éste ha recibido.

### ADVERTENCIA

En este ejemplo, he dejado el valor de la contraseña en la propia conexión. No es necesario, por supuesto, ni recomendable.

Seguimos con parámetros relacionados con la conexión. *SQLDialect* nos sirve para indicar el nivel de compatibilidad con la versión 5.6. El dialecto 1 es el más compatible, pero no nos permite utilizar las nuevas características de la versión 6; si está interesado en esas novedades, utilice el dialecto 3. *ServerCharSet* indica al controlador cuál es el conjunto de caracteres que está utilizando el servidor; si es el mismo que utiliza el cliente, podemos dejarlo en blanco. Finalmente *LocaleCode* indica un código numérico asociado a los lenguajes humanos, que podemos utilizar para indicar criterios de ordenación alternativos en la parte cliente.

Los siguientes parámetros están relacionados con el funcionamiento de las transacciones. *Interbase TransIsolation* determina el nivel de aislamiento de las transacciones lanzadas desde esta conexión. Los valores permitidos son los típicos: *ReadCommitted* y *RepeatableRead*. *CommitRetain* es soportado solamente por InterBase, e indica si al terminar una transacción debemos descartar también los cursores abiertos, o no.



Aunque el estándar SQL aconseja descartar los cursores, es más eficiente mantenerlos abiertos, como podrá suponer. *WaitOnLocks* especifica qué debe pasar cuando InterBase va a modificar un registro que está siendo modificado ya por otra transacción: en tal caso podemos esperar a que termine la otra transacción, o abortar inmediatamente la nuestra.

Por último, *BlobSize* es un parámetro presente en todos los controladores, que al parecer se ha introducido pensando en las interfaces nativas que no ofrecen información sobre el tamaño de un campo blob antes de recuperarlo. Por lo visto, ninguno de los cuatro controladores disponibles actualmente, necesita poner valores explícitos en este parámetro.

Ahora nos será más fácil explicar los parámetros utilizados por Oracle, porque la mayoría de ellos coinciden con los de InterBase:

```
DriverName=Oracle
BlobSize=-1
DataBase=CHRISSIE
ErrorResourceFile=
LocaleCode=0000
Password=tiger
Oracle TransIsolation=ReadCommitted
User_Name=scott
```

La diferencia más importante es la interpretación de *Database*, que para este controlador significa el nombre asociado a la base de datos en el fichero local *tnsnames.ora*, que normalmente se crea con herramientas como *Net8 Configuration Assistant*.

Las conexiones para DB2 se configuran de manera parecida:

```
[DB2Connection]
BlobSize=-1
Database=DBNAME
DriverName=DB2
ErrorResourceFile=
LocaleCode=0000
Password=password
DB2 TransIsolation=ReadCommitted
User_Name=user
```

En este caso, *Database* hace referencia al nombre de nodo que se establece en el cliente para identificar una instancia en el servidor.

Por último, en las conexiones de MySQL hay que indicar por separado el *HostName*, para referirnos al servidor, y *Database*, el nombre de la base de datos que queremos abrir:

```
[MSConnection]
BlobSize=-1
Database=DBNAME
DriverName=MYSQL
ErrorResourceFile=
```

```

HostName=ServerName
LocaleCode=0000
Password=password
User_Name=user

```

## El origen de los parámetros

Cuando expliqué la función del fichero *dbxconnections.ini*, mencioné también la existencia de una propiedad en el componente de conexión de DB Express, llamada *LoadParamsOnConnect*. Su valor por omisión es *False*. Cuando está activa, el componente “refresca” los valores de sus parámetros antes de establecer una conexión en tiempo de ejecución. Para ello, busca en el registro de Windows la ubicación el fichero de configuración de conexiones y lo abre. Entonces utiliza el valor de su propiedad *ConnectionName* para identificar una de las secciones del fichero y releer los parámetros.

Esta técnica se puede utilizar para hacer que una aplicación pueda trabajar alternativamente con bases de datos diferentes, sin necesidad de modificarla y recompilarla. Es algo similar a lo que puede lograrse con el Motor de Datos de Borland y los alias “persistentes”.

Pero podemos lograr más control aún si dejamos a *LoadParamsOnConnect* con su valor por omisión y llamamos al siguiente método antes de abrir la conexión:

```

procedure TSQLConnection.LoadParamsFromIniFile(
    Fichero: string = '');

```

Si no especificamos un nombre de fichero, o pasamos explícitamente la cadena vacía, el método vuelve a utilizar el *dbxconnections.ini* situado en la ubicación grabada en el registro de Windows. Si va a utilizar este método, le recomiendo que, siempre que sea posible, sitúe el fichero con los datos de la conexión en el mismo directorio que la aplicación. Si necesita estar seguro acerca de cuál es ese directorio, puede utilizar la siguiente función para averiguarlo:

```

function DirectorioModulo: string;
begin
    SetLength(Result, MAX_PATH);
    SetLength(Result,
        GetModuleFileName(HInstance, PChar(Result), Length(Result)));
    Result := ExtractFilePath(Result);
end;

```

Sí, es cierto que la propiedad *Application.ExeName* y el resultado de *ParamStr(0)* nos ofrecen también el nombre del fichero ejecutable ... pero como comprenderá, no funcionan con una DLL, como sería el caso de una aplicación ISAPI para Internet. ¡Ah!, *HInstance* es una variable global, definida en la unidad *SysInit*, y contiene el *handle*, o identificador, del módulo actual, sea éste un ejecutable o una DLL.

## El diálogo de conexión

Antes he dicho que no es buena idea dejar expuesta la contraseña de conexión dentro de las propiedades de un *TSQLConnection*. Eso no es del todo cierto, porque hay aplicaciones que necesitan extraer información de bases de datos y no se ejecutan con la ayuda de un usuario interactivo, como sucede con las aplicaciones CGI, los servidores de capa intermedia, etc. Pero incluso con estas aplicaciones debemos tener sumo cuidado, porque es muy fácil “olisquear” dentro de un fichero ejecutable con la ayuda de un editor de recursos, y extraer los valores de las propiedades para el componente que elijamos. Muchas veces, esto se puede resolver impidiendo simplemente el acceso en modo de lectura al ejecutable. Pero en los casos que no sea posible restringir el acceso, debemos tomar alguna precaución, como codificar la contraseña.

Por este motivo, veamos qué oportunidades nos ofrece el componente de conexión de DB Express para especificar la contraseña y el nombre del usuario en tiempo de ejecución. Ya sabemos que lo más sencillo es dejar en *False* la propiedad *LoginPrompt* del componente, y asignar directamente la contraseña en *Params*.

Supongamos ahora que asignamos *True* a *LoginPrompt*. Lo que suceda, dependerá de varias circunstancias, pero puedo garantizar que se intentará ejecutar el método asociado al evento *OnLogin*. El siguiente método muestra un posible uso del evento:

```
procedure TForm1.SQLConnection1Login(Database: TSQLConnection;
    LoginParams: TStrings);
begin
    LoginParams.Values['user_name'] := 'SYSDBA';
    LoginParams.Values['password'] :=
        Decodificar(LoginParams.Values['password']);
end;
```

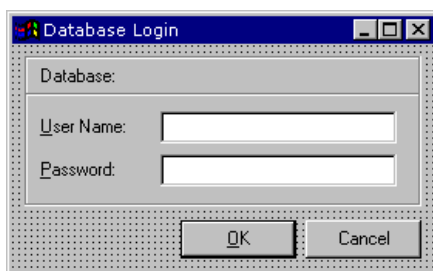
*LoginParams* es una lista de cadenas que contiene pares *parámetro=valor*. ¡Cuidado, porque no contiene todos los parámetros de la conexión! Si el controlador es el de InterBase, por ejemplo, solamente incluye la base de datos, el usuario y la contraseña. En el ejemplo anterior, se está asignando directamente el nombre del usuario. La siguiente instrucción no es coherente con la primera, pero la he puesto para indicar otra posibilidad: estoy llamando a una hipotética función *Decodificar* para efectuar esa operación sobre la contraseña almacenada en el componente. El propósito sería hacer un poco más difícil que un *hacker* pueda averiguar la contraseña si el fichero ejecutable cae en sus manos. En la próxima sección mostraré un algoritmo muy sencillo de codificación.

¿Y qué sucede si *LoginPrompt* vale *True*, pero no hay método alguno asociado a *OnLogin*? Mucha atención, porque el comportamiento ha variado en Delphi 6. Ahora todo depende del contenido de una variable global, declarada en la unidad *DB*:

```
var
    LoginDialogExProc: function (
        const ADatabaseName: string;
```

```
var AUserName, APassword: string;
    NameReadOnly: Boolean): Boolean;
```

La unidad *DB* es común a todos los sistemas de acceso a datos en Delphi, por lo que esta función no se compromete en modo alguno en su implementación. Por lo tanto, si *no hacemos nada especial*, la conexión a la base de datos se intentará realizar con los valores asignados en ese momento como parámetros, y provocará una excepción. En las versiones anteriores de Delphi, se mostraba un cuadro de diálogo predefinido, para esos casos.



Para que aparezca el típico cuadro de diálogo que pide el nombre del usuario y su contraseña, nos bastaría con incluir la unidad *DBLogDlg* en alguna de las cláusulas **uses** del proyecto. Esta unidad define un formulario con el aspecto de la imagen anterior, pero además asigna una referencia a procedimiento en la variable global *LoginDialogExProc*.

Los adeptos a InterBase deberían quejarse amargamente, llegado este momento. Uno de los parámetros de conexión de InterBase es, como hemos visto, el nombre del ... lo diré en inglés, porque me da la gana ... el nombre del *role* que quiere asumir el usuario para esa conexión. El problema no puede resolverse ni siquiera proporcionando una implementación nuestra para *LoginDialogExProc*, porque los parámetros que recibe el diálogo son un subconjunto muy pequeño de todos los parámetros existentes, y no se incluye *RoleName*. Si realmente estamos utilizando *roles*, y el usuario debe poder elegir uno de ellos, tendremos que interceptar el evento *BeforeConnect* o, si la instrucción que abre la conexión está bien localizada, asignar los parámetros necesarios antes de ejecutarla.

## Un sencillo algoritmo de codificación

El error principal que cometemos los programadores (note que me incluyo) al tratar con algoritmos de codificación, es no tener en cuenta el uso que le vamos a dar. Es muy fácil dar con una rutina que destruya una cadena hasta que no la reconozca ni la madre que la parió ... pero cuando la rutina que la recompone se encuentra dentro de un programa que es atacado por un *hacker*, es igual de sencillo ejecutar el programa paso a paso en ensamblador y descubrir el algoritmo que creíamos inexpugnable.

La mayoría de los sistemas de codificación se basan en un algoritmo relativamente sencillo, y que probablemente es conocido por el *hacker*, más una clave secreta que

encierra el verdadero problema para el asaltante. Por eso, no tiene mucho sentido incluir una rutina demasiado compleja para decodificar la contraseña, como nos planteamos en la sección anterior: el pirata tendría a su disposición tanto la pistola (el algoritmo) como la bala (la clave).

Sin embargo, es positivo que pongamos una primera barrera para desalentar a los atacantes menos duchos. Si incluimos la contraseña sin más dentro de las propiedades de un componente, el *hacker* sólo necesitará un editor de recursos para descifrarla. Pero si incluimos un algoritmo de codificación, aunque sea muy sencillo, ya estaremos obligándolo a depurar el programa paso a paso, que es algo más complicado.

Partiendo de estas premisas, ¿cómo podemos disfrazar la contraseña para que no sea tan evidente? El algoritmo más sencillo es perfectamente adecuado para este caso: elija un valor numérico entre 0 y 255. Para cada carácter de la contraseña, obtenga su valor numérico según el código ANSI y aplíquelo la operación conocida como *ó exclusivo*, o complemento binario, tomando como segundo operando el valor numérico que haya elegido. ¡Ah!, eso del complemento binario es nuestro vulgar **xor**:

```
function Codificar1(const Valor: string; Clave: Byte): string;
var
  I: Integer;
begin
  SetLength(Result, Length(Valor));
  for I := 1 to Length(Valor) do
    Result[I] := Chr(Ord(Valor[I]) xor Clave);
  end;
```

Este algoritmo es atractivo porque lo mismo codifica que descodifica. Quiero decir, si aplicásemos nuevamente la función con la misma clave a una cadena codificada, volveríamos a obtener la cadena original.

Podríamos dejarlo ahí ... pero he recordado una idea que encontré en un artículo de Julian Bucknall para la Delphi Magazine (Feb/2000), y me ha parecido interesante mostrar un algoritmo basado en la misma. El hecho es que nuestro primer intento de codificación presenta una característica matemática que lo hace prácticamente inútil cuando el texto a codificar es suficientemente largo: el algoritmo conserva las frecuencias de aparición de caracteres. Si en el texto original la letra 'E' aparece en un 10% de los casos, la letra en la que se transforme aparecerá en el resultado el mismo número de veces. Con esas frecuencias relativas, cualquier persona medianamente hábil puede restablecer el mensaje secreto, como sucede en 'El Escarabajo de Oro', un conocido cuento de Edgar Allan Poe.

Hace ya mucho tiempo, en una de las primeras versiones de dBase, se anunció a bombo y platillo la introducción de tablas cifradas. Pero la alegría duró pocos días: dBase aplicaba el algoritmo **xor** a los datos, y la clave numérica, si no recuerdo mal, era un simple 127. Como comprenderá, los siete bits menos significativos de este número contienen unos, por lo que la operación de complemento se limita a la negación binaria de cada carácter original.

La técnica más usada como paliativo consiste en no utilizar siempre el mismo número para codificar todos los caracteres. Se elige una sucesión de números como clave; si son números de 8 bits, podemos mostrarlos como caracteres ANSI, para simplificar la explicación. Se colocan, uno al lado del otro, el texto original y la cadena que se obtiene repitiendo la clave tantas veces como sea necesario para que tenga la misma longitud que el mensaje a codificar:

H	a	b	í	a	_	u	n	a	_	v	e	z
D	E	L	P	H	I	D	E	L	P	H	I	D

Entonces se ejecuta el algoritmo del **xor**, mezclando la 'H' con la 'D', la 'a' con la 'E' y así sucesivamente. De este modo, se desfigura el histograma de frecuencia de caracteres en el texto transformado. Y sigue cumpliéndose que el algoritmo sirve a la vez como codificador y decodificador.

Mientras más larga sea la clave, mejor; en el caso ideal, la clave debería tener la misma longitud que el texto. Claro, eso tendría un coste en espacio. Pero, ¿qué tal si generamos la clave mediante números aleatorios? Alguien razonará que, al ser una clave generada al azar, perderíamos la posibilidad de recuperar el texto original ... Pues no, se equivoca, amigo mío. Podemos arreglar las cosas para que Delphi genere *siempre* la misma secuencia durante la codificación y la decodificación. Nos basta con asignar en ambas operaciones, un mismo valor a la semilla del generador de números de Delphi, que se encuentra en la variable global *RandSeed*. De esta manera, nuestra clave, mediante un truco ingenioso, deja de ser un simple byte y se convierte en un valor de 32 bits, que es el número que asignamos como semilla. La siguiente función muestra los detalles del algoritmo:

```
function Codificar(const Valor: string;
  Semilla: Integer = 1453): string;
var
  OldSeed: LongInt;
  I: Integer;
begin
  OldSeed := RandSeed;
  try
    RandSeed := Semilla;
    SetLength(Result, Length(Valor));
    for I := 1 to Length(Result) do
      Result[I] := Chr(Ord(Valor[I]) xor Random(256))
    finally
      RandSeed := OldSeed;
    end;
  end;
```

Por ejemplo, si utilizamos la semilla *1453* sobre la contraseña *masterkey*, obtendríamos la siguiente cadena en notación Delphi:

```
#$F5#$EB'h'#$1C#$C4#$A2'2'#$8A'z'
```

Cómo ejemplo de lo que afirmé al principio de esta sección, este algoritmo es relativamente fácil de descifrar, incluso cuando no disponemos de la clave, porque ésta es un número de 32 bits, y es posible abordar el problema con algoritmos de fuerza bruta. Pero ya no basta con cargar el programa en un editor de recursos o en una herramienta de volcado hexadecimal para encontrar la contraseña de la base de datos.

## Ejecución de comandos

Cuando se produjo la primera ola de migraciones desde bases de datos de escritorio a sistemas SQL, escuchaba con mucha frecuencia la siguiente consulta: ¿qué métodos hay que llamar desde Delphi para crear una tabla (o un índice, o un procedimiento almacenado)? Es que incluso el programador que trabajaba en Delphi con Paradox y dBase estaba acostumbrado a utilizar métodos especiales de la clase *TTable* para crear tablas e índices. Esos métodos, a su vez, llamaban a ciertas rutinas muy específicas del Motor de Datos. Cuando la tabla tenía características “avanzadas”, como integridad referencial o valores por omisión en algunas columnas, no quedaba más remedio que saltarse los métodos de “alto” nivel de *TTable* y trabajar directamente con la interfaz del BDE. Por lo tanto, esos mismos programadores esperaban angustiados que la creación de objetos en bases de datos SQL fuera igual de complicada...

Por suerte para todos, esta tarea se lleva a cabo ejecutando instrucciones SQL, del llamado Lenguaje de Definición de Datos, o DDL. En algunos casos, como en la gestión de usuarios y privilegios, las instrucciones necesarias se clasifican dentro un grupo especial llamado Lenguaje de Control de Datos, o DCL; se trata, sin embargo, de una distinción algo bizantina. Lo que ahora nos importa es que es muy sencillo ejecutar una de estas sentencias sobre una base de datos, si utilizamos DB Express. Veremos que también es posible ejecutar así sentencias del Lenguaje de Manipulación de Datos, como **update**, **insert** y **delete**. ¿Ha notado que la única sentencia que he dejado fuera es **select**, la instrucción de recuperación de datos? Como podrá imaginar, la exclusión se debe a que **select** debe devolver un conjunto de filas, una estructura de datos de cierta complejidad, que se maneja mejor utilizando *cursores*. En el próximo capítulo veremos cómo recuperar datos utilizando los conjuntos de datos de DB Express. Pero quiero que sepa que el mecanismo que vamos a presentar ahora mismo puede incluso ejecutar consultas, aunque no sea la forma más adecuada.

Aquí tiene el prototipo del método que necesitamos:

```
procedure TSQLConnection.ExecuteDirect(const SQL: string): LongWord;
```

Más sencillo, imposible. Estando activa la conexión, llamamos a *ExecuteDirect*, pasándole como único parámetro una cadena de caracteres con la instrucción que queremos ejecutar. En cuanto al valor de retorno ... ya sé que a estas alturas debería ser inmune a las mentirijillas de la documentación. Dice la “ayuda” en línea que la función devuelve 0 si todo fue bien, y un código numérico definido por DB Express en caso contrario. Mentira: si se produce un error, no se devuelve nada, porque el método lanza una excepción antes de terminar. Si todo va bien, el valor de retorno de

*ExecuteDirect* representa el número de registros afectados por la instrucción. Si la instrucción ejecutada pertenecía al DDL o al DCL, como **create table**, por ejemplo, no tiene sentido hablar de registros afectados. Pero si se trataba de un **update**, el valor corresponde al número total de registros modificados. Lo mismo se aplica en el caso de las inserciones y borrados. Además...

... puedo imaginarme lo que está rondando por su cabeza. Le advierto que no se entusiasme demasiado: no se puede crear una base de datos de InterBase a través de este método. Piense un momento: para que *ExecuteDirect* funcione, hace falta que la conexión esté activa, y para esto hace falta especificar una base de datos. Un círculo vicioso. Se podrían haber inventado escapatorias, pero puede comprobar que no es posible realizando el experimento, de todos modos.

## Ejecución de un *script*

Lo que sí es sencillo, y potencialmente útil, es una aplicación que lea sentencias SQL desde un fichero y las ejecute sobre la base de datos que determinemos. Con *ExecuteDirect* podemos ejecutar una sola sentencia; sólo tendríamos entonces que dividir el contenido del fichero e ir ejecutando cada una de las instrucciones que se extraigan. Una aplicación, o una DLL, de este tipo sería muy conveniente para instalación de aplicaciones en un servidor, porque podría automatizar parte de la creación de la base de datos. Es cierto que con InterBase, en particular, tendríamos que utilizar otros componentes para crear inicialmente la base de datos vacía, pero a partir de ese momento podríamos crear los objetos dentro de la misma con la ayuda de esta técnica.

Como no me sobra el espacio, sin embargo, he simplificado un poco el formato de *scripts* que vamos a aceptar. En vez de utilizar el innecesariamente complicado sistema de InterBase, utilizaremos el más sencillo de SQL Server: para separar una instrucción de otra utilizaremos una cadena que situaremos en una línea aislada. Así evitaremos el análisis de comentarios y cadenas de caracteres dentro del *script*. He implementado el algoritmo de separación en una clase auxiliar, y he desterrado su código fuente al CD que acompaña al libro. Sólo mostraré aquí la parte pública de la declaración:

```
type
  TScript = class
    // ...
  public
    constructor Create(const AStream: TStream;
                      const ASeparator: string);
    property Eof: Boolean read FEof;
    procedure Next;
    property Statement: string read FStatement;
  end;
```

En realidad, la implementación podría haber sido mucho más sencilla si hubiera estado seguro de que el *script* siempre estará almacenado en un fichero del sistema ope-



rativo. En ese caso, me habría basado en el tipo *TextFile* para ficheros de textos, y en el procedimiento *ReadLn* para leer líneas completas. Pero se me ocurrió que alguien quizás quiera guardar el *script* como un recurso dentro de la propia aplicación, y es por eso que he utilizado la clase abstracta *TStream*, con la consecuencia de tener que implementar un método especial para la lectura de líneas.

En la aplicación, en sí, he puesto un cuadro de edición (*edFichero*) para que el usuario introduzca el nombre del fichero con el *script*, y otro (*edSeparador*) para indicar el separador. He puesto un componente *SQLConnection1*, y un botón *Ejecutar* para que dispare el siguiente método al ser pulsado:

```
procedure TwndPrincipal.EjecutarClick(Sender: TObject);
var
    Stream: TStream;
    Script: TScript;
begin
    Stream := TFileStream.Create(edFichero.Text, fmOpenRead);
    try
        Script := TScript.Create(Stream, edSeparador.Text);
        try
            while not Script.Eof do
                begin
                    SQLConnection1.Execute(Script.Statement);
                    Script.Next;
                end;
            finally
                Script.Free;
            end;
        finally
            Stream.Free;
        end;
    end;
```

Como puede apreciar, la ejecución del *script* es tan sencilla como separar las distintas instrucciones que contiene y ejecutar secuencialmente cada una de ella. Es cierto que si nos hubiéramos restringido a los convenios propios de InterBase tendríamos que haber luchado con problemas no muy bien especificados de la estructura lexical del lenguaje de este sistema. Pero una vez simplificada la sintaxis, es coser y cantar.

Por supuesto, para poder utilizar el código anterior en una instalación tendríamos que hacer algunos pequeños retoques: crear una función que reciba directamente el nombre de fichero y el separador, para no tener que preguntar; quizás crear y configurar dinámicamente el componente de conexión... Se lo dejo para que se divierta.

## Manejo de transacciones

En el capítulo 11 estudiamos las transacciones teóricamente. Si está leyendo el libro secuencialmente, será esta la primera vez que tropezaremos con una implementación concreta de las mismas. Le advierto que no intente generalizar demasiado lo que pueda aprender aquí, porque los métodos transaccionales de ADO y BDE, por un

lado, y de IB Express, por el otro, son ligeramente distintos de los implementados por DB Express.

Para iniciar una transacción, hay que llamar al siguiente método:

```
procedure TSQLConnection.StartTransaction(  
    TransDesc: TTransactionDesc);
```

La declaración del tipo *TTransactionDesc* es, sorprendentemente, la siguiente:

```
type  
    TTransIsolationLevel = (xilDIRTYREAD, xilREADCOMMITTED,  
        xilREPEATABLEREAD, xilCUSTOM);  
  
    TTransactionDesc = packed record  
        TransactionID : LongWord;  
        GlobalID      : LongWord;  
        IsolationLevel : TTransIsolationLevel;  
        CustomIsolation: LongWord;  
    end;
```

No, no se me ha ido la mano con el adverbio, ni es un arrebató lírico: *TTransactionDesc* ocupa 16 bytes. Eso significa, en primer lugar, que no se puede pasar directamente el parámetro en un registro; además, hay que sacar una copia local del parámetro. Si se hubiera utilizado el modificador **const**, el traspaso sería más eficiente. Pero sospecho que todo se debe a la extraña implementación de la interfaz *ISQLConnection* por parte de los controladores de DB Express.

A lo nuestro. Para un viejo programador de Delphi como un servidor (quiero decir, *mois*), resulta extraño tener que pasar un descriptor para poder iniciar una transacción. Lo que sucede es que el BDE no permite activar concurrentemente más de una transacción utilizando una sola conexión; hay servidores SQL que sí lo permiten, pero han sido sacrificados en aras del resto. Pero resulta que el sacrificado servidor es nada más y nada menos que ... ¡InterBase! Es decir, el profeta en su propia tierra. Cuando os torture presentando IB Express, veréis que esta interfaz ha tenido en cuenta esta capacidad de InterBase separando el manejo de transacciones en un componente independiente. Sin llegar a tanto, DB Express ha ideado un descriptor de transacciones para permitir la activación simultánea de varias de ellas, cuando es InterBase quien está al otro extremo de la línea.

A pesar de que *TTransactionDesc* tiene cuatro campos, solamente tendrá que suministrar valores a dos de ellos, al menos cuando trabaje con InterBase. El inicio típico de una transacción DB Express es el siguiente:

```
var  
    TD: TTransactionDesc;  
begin  
    TD.TransactionID := 1;  
    TD.IsolationLevel := xilREPEATABLEREAD;  
    SQLConnection1.StartTransaction(TD);  
    // ...
```

El valor de *TransactionID* debe inventárselo usted mismo. Naturalmente, debe recordar cuál era para poder confirmar o cancelar esa transacción más adelante, porque los métodos correspondientes también requieren de un descriptor de transacción. *IsolationLevel* es el nivel de aislamiento de la transacción, que presenté en el capítulo 11. Cuando trabajo con InterBase siempre utilizo el nivel superior, el de lecturas repetibles, como en el ejemplo. Es cierto que es el más costoso, pero:

- 1 Una de las ventajas de InterBase es que no es *tan* costoso como en otros sistemas de bases de datos.
- 2 Si no se utiliza ese nivel, hay que analizar muy cuidadosamente los posibles conflictos de concurrencia. Y eso suele ser bastante difícil.

¿Y qué pasa con los otros dos campos del descriptor? *CustomIsolation* se debe utilizar solamente cuando se especifica *xi/CUSTOM* como nivel de aislamiento. Es un mecanismo para indicar niveles especiales de aislamiento que no entren en las tres categorías habituales; sin embargo, ninguno de los controladores actuales de DB Express necesita esta ampliación.

El campo *GlobalID*, sin embargo, es más problemático. La documentación se limita a decir que sólo es necesario en Oracle. Y punto. Para colmo, no hay demos específicas para Oracle y DB Express<sup>25</sup>. Por suerte, pude seguir una buena pista: en aplicaciones “reales”, como veremos más adelante, las actualizaciones sobre conjuntos de datos DB Express utilizarán como intermediario el sistema de caché de los conjuntos de datos clientes. Estos últimos componentes, por exigencias de DataSnap, deben saber cómo iniciar y terminar una transacción. Por lo tanto, fui al código fuente, a la implementación de la clase *TCustomSQLDataSet* y eché un vistazo a los métodos correspondientes. Esto fue lo que encontré:

```
procedure TCustomSQLDataSet.PSStartTransaction;
var
    TransDesc: TTransactionDesc;
begin
    FillChar(TransDesc, Sizeof(TransDesc), 0);
    TransDesc.TransactionID := 1;
    FSQLConnection.StartTransaction(TransDesc);
end;
```

No es necesario explicar aquí qué papel desempeña este método; lo haremos al presentar DataSnap. Pero lo importante ahora es que se supone que debe ser capaz de iniciar una transacción independientemente del controlador que se esté utilizando. Por lo tanto, el método funciona en Oracle. Como puede ver, todos los campos, incluyendo el *GlobalID*, se inicializan a cero gracias a la llamada a *FillChar*. He hecho la prueba con Oracle 8.1.7 y no he encontrado problema alguno.

---

<sup>25</sup> Extrañamente, hay una demostración en el directorio *Oracle8*, pero basada en el BDE. Lo “extraño” es que el icono asociado a la aplicación es de antes del diluvio universal.

Y como todo lo que empieza debe acabar en algún momento, incluyendo los dolores de muelas, *TSQLConnection* nos ofrece los siguientes métodos para liquidar una transacción:

```
procedure TSQLConnection.Commit(TD: TTransactionDesc);
procedure TSQLConnection.Rollback(TD: TTransactionDesc);
```

Es decir, hay que volver a pasar un descriptor de transacciones, que en su campo *TransactionID* debe tener el identificador numérico de la transacción que queremos cerrar. Tengo el presentimiento de que ambos métodos ignoran el resto de los campos.

### ADVERTENCIA

Uno se acostumbra tan rápido a la buena vida, que olvida que ciertos sistemas con pretensión de SQL no soportan transacciones: léase MySQL. Si tiene una aplicación que se puede conectar a varios tipos de bases de datos, es recomendable que compruebe el valor de la propiedad *TransactionsSupported* antes de lanzarse a la piscina.

## Agítese antes de usar

Ahora que ya conocemos el final de la película, voy a explicarle cómo vamos a utilizar las transacciones la mayoría de las veces, porque es muy fácil llevarse una idea equivocada de cómo funciona este negocio.

En primer lugar, veremos que en la gran mayoría de las aplicaciones no podremos utilizar DB Express “a secas”, sino que tendremos que complementarlo con conjuntos de datos clientes (CDS). Recuerde que, por una parte, los conjuntos de datos DB Express son unidireccionales, pero lo más importante es que no pueden ser actualizados directamente, y aquí también nos echan una mano los CDS. Lo normal, cuando nos auxiliamos de estos componentes para una actualización de un conjunto de datos, es que sea la propia infraestructura de Delphi la que inicie y finalice las transacciones. Es por este motivo por el que fui al código fuente a buscar una respuesta al problema del *GlobalID* de Oracle. En este tipo de aplicaciones, nos podemos lavar las manos y secárnoslas con las páginas del manual.

Pero es también frecuente realizar cambios en la base de datos a través de procedimientos almacenados y ejecutando directamente sentencias de actualización SQL. Un caso típico son las aplicaciones para Internet: si hay que cambiarle el apellido a un usuario, usted no va a abrir una consulta para traer los valores del registro actual, modificar el registro en memoria y pedirle entonces a alguna interfaz que genere por usted la sentencia SQL necesaria para el cambio. No señor, usted mismo creará la sentencia y la ejecutará. Al menos, si es un programador decente.

Es en este segundo escenario donde será más probable que iniciemos y terminemos transacciones explícitamente. El siguiente ejemplo muestra el patrón de código recomendable:

```

procedure TmodDatos.CorradaDeGrabacion;
var
    TD: TTransactionDesc;
begin
    TD.TransactionID := 1;
    TD.GlobalID := 0;                // Si no es Oracle, sobra
    TD.IsolationLevel := xilREPEATABLE;
    SQLConnection1.StartTransaction(TD);
    try
        // Ejecutar las instrucciones o los procedimientos necesarios
        // ...
        SQLConnection1.Commit(TD);    // Tutto bene
    except
        SQLConnection1.Rollback(TD); // Algo ha salido mal
        raise;
    end;
end;

```

Como puede ver, la transacción se inicia y finaliza dentro del mismo bloque de código.

## Nunca llueve a gusto de todos

Esta vida es muy dura: hay sistemas de bases de datos tacaños y cicateros que solamente te dejan colar una instrucción por conexión. Cuando se utiliza SQL Server a través de la ya obsoleta interfaz *DBLibrary*, solamente se puede ejecutar una instrucción por conexión. En la época en que Delphi sólo ofrecía el BDE como medio de acceso a SQL Server, era obligatorio tener conciencia de esta limitación, si no queríamos que la velocidad de nuestra aplicación sufriese las consecuencias. Sinceramente, no sé si la interfaz de SQL Server de más bajo nivel sigue teniendo esta “característica”; pero no me preocupa en absoluto, porque en caso afirmativo, ADO se encargaría automáticamente de disfrazar esa hipotética limitación. Moraleja: una interfaz de acceso a datos decente debe ser capaz de resolver este tipo de problemas por nosotros.

DB Express ha sido diseñada como una interfaz de acceso a datos decente, y creo que con un par de *service packs* puede alcanzar ese objetivo. Bromas aparte, las conexiones DB Express ofrecen una propiedad llamada *AutoClone*, de tipo lógico. Cuando su valor es *True*, la conexión vigila el valor de su propiedad *ActiveStatements*, para saber cuántas instrucciones activas están pasando a través de ella. Si ese valor sobrepasa el de otra de sus propiedades, *MaxStmtsPerConn*, la conexión crea un duplicado propio de forma invisible para nosotros. Y la fiesta no tiene porqué detenerse.

Esta última propiedad es de sólo lectura, y depende únicamente del controlador DB Express que se esté utilizando. Es muy fácil averiguar su valor: coloque un *TSQL-Connection* sobre un formulario, conéctelo a la base de datos que quiera poner a prueba, y escriba un manejador para el evento *OnCreate* del formulario como el siguiente:

```

procedure TForm1.FormCreate(Sender: TObject);

```

```
begin
    ShowMessage(IntToStr(SQLConnection1.MaxStmtsPerConn));
end;
```

Puede respirar tranquilo, porque InterBase, Oracle y DB2, todos ellos, soportan ... bueno, dice el programa que cero instrucciones por conexión. Por supuesto, eso quiere decir que no hay límites. El único al que le flojean las piernas es a MySQL, que tampoco permite transacciones en su versión actual. Es que a veces lo barato sale caro.

Si prefiere controlar usted mismo la clonación de las conexiones, puede apagar la propiedad *AutoClone*, y llamar al método *CloneConnection* cuando crea que el componente esté a punto de estallar. Yo sigo prefiriendo dejar conectado el piloto automático.

## El monitor de DB Express

Para terminar este capítulo, voy a presentarle un componente casi imprescindible, al menos durante la fase de aprendizaje con DB Express: *TSQLMonitor*. Con su ayuda, podremos observar qué instrucciones envía DB Express al servidor de bases de datos en cualquier momento, e incluso las funciones de la interfaz nativa que se llaman.

Propiedad	Significado
<i>SQLConnection</i>	Un puntero a una conexión de DB Express
<i>Active</i>	Solamente se registran instrucciones cuando <i>Active</i> vale <i>True</i>
<i>TraceList</i>	Por omisión, la traza se acumula en esta lista de cadenas
<i>AutoSave</i>	Para grabar cada mensaje en un fichero de texto
<i>FileName</i>	El nombre del fichero, cuando <i>AutoSave</i> está activo

Cuando el monitor está activo, recibe notificaciones desde la conexión DB Express. Todas estas notificaciones van asociadas con un texto explicativo, y por omisión ese texto se añade dentro de la propiedad *TraceList* del componente. Si confiamos en la estabilidad de la aplicación, podemos dejar que se acumulen las notificaciones y pasarlas a un fichero de texto al terminar la ejecución de la aplicación. En realidad, podemos manipular el contenido de *TraceList* en cualquier momento: podemos guardarlo, borrarlo, modificarlo, e incluso ignorarlo. Ni siquiera hace falta que la aplicación esté ejecutándose para poder fisgonear dentro de *TraceList*: podemos hacerlo incluso en tiempo de diseño.

Pero si quisiéramos realizar un seguimiento serio del funcionamiento de una aplicación, tendríamos que arreglar las cosas para que los mensajes se graben con cierta frecuencia en un fichero o algún otro medio menos volátil. En teoría, el programador de *TSQLMonitor* incluyó las propiedades *AutoSave* y *FileName* con este propósito. Pero hay que tener un poco de cuidado con *AutoSave* porque, cuando está activa, cada vez que el componente recibe un mensaje y lo añade a la lista en memoria, graba además *toda esa lista* de mensajes en el fichero especificado en *FileName*. Si hay

1.000 mensajes y llega uno más (el que anuncia la decapitación de Scheherezade), el monitor internamente ejecuta método *SaveToFile* de la lista de cadenas para grabar los mil y uno mensajes, y así *ad nauseam*.

- *OnTrace*: Es el primero en dispararse. Se diferencia del siguiente evento en que nos pasan un parámetro *LogTrace* por referencia, para que decidamos si se graba el mensaje o no. En la versión actual, *LogTrace* siempre vale *True* al dispararse el evento, porque no se ha implementado todavía un filtro de mensajes.
- *OnLogTrace*: Se dispara solamente si no hemos denegado la grabación del mensaje en el evento anterior. En ese preciso momento, ya se ha añadido el mensaje a *TraceList*. Pero, si la propiedad *Auto.Save* estuviese activa, todavía no se habría guardado la lista de mensajes en el disco.

Por lo tanto, si queremos personalizar el mecanismo de grabación de mensajes debemos utilizar *OnTrace*, antes que *OnLogTrace*. El siguiente manejador de eventos asume que el nombre del fichero donde queremos depositar los mensajes se almacena en la misma propiedad *FileName* del componente. La diferencia de velocidad respecto al algoritmo de fuerza bruta utilizado por omisión, es impresionante, a partir de unos miles de mensajes.

```

procedure TForm1.SQLMonitor1Trace(Sender: TObject;
    CBInfo: pSQLTRACEDesc; var LogTrace: Boolean);
var
    Msg: string;
    F: TextFile;
begin
    // Preparar cadena con el mensaje
    SetLength(Msg, CBInfo.uTotalMsgLen);
    Move(CBInfo.pszTrace[0], Msg[1], CBInfo.uTotalMsgLen);
    // Asociar nombre al descriptor del fichero de texto
    AssignFile(F, SQLMonitor1.FileName);
    try
        Append(F); // Intentar abrir e ir al final
    except
        Rewrite(F); // Si el fichero no existe, crearlo
    end;
    try
        WriteLn(F, Msg); // Grabar el mensaje
    finally
        CloseFile(F); // Eso es todo, amigos
    end;
    LogTrace := False; // Ya lo hemos hecho, ¿no?
end;

```

Naturalmente, si le molesta seguir utilizando en pleno siglo XXI los procedimientos para ficheros de texto del Turbo Pascal, puede reescribir el método utilizando *TFileStream*, o alguna clase aún más “sofisticada”.

El tipo *pSQLTRACEDesc*, al que pertenece el parámetro *CBInfo* del evento anterior, es un puntero a un registro con la siguiente estructura:

```

type
  pSQLTRACEDesc = ^SQLTRACEDesc;
  SQLTRACEDesc = packed record
    pszTrace      : array [0..1023] of Char;
    eTraceCat     : TRACECat;
    ClientData    : Integer;
    uTotalMsgLen  : Word;
  end;

```

Por último, es bueno que sepa que *TSQLMonitor* tiene una propiedad llamada *MaxTraceCount*, disponible solamente en tiempo de ejecución, que pone un límite a la cantidad de mensajes que se almacenan en *TraceList*. Por omisión vale  $-1$ , valor que indica que no hay límites para el crecimiento de *TraceList*.

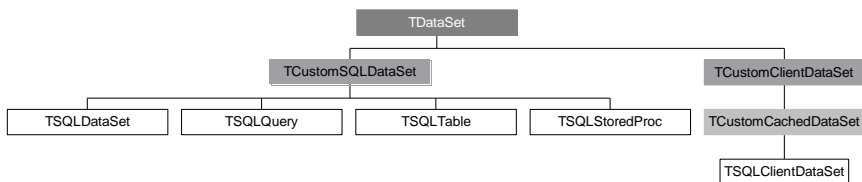


## DB Express: conjuntos de datos

**H**A LLEGADO EL MOMENTO DE PRESENTAR cuatro de los cinco componentes que ofrece DB Express para el manejo de conjuntos de datos. Dejaremos pendiente el estudio de *TSQLClientDataSet* para cuando tratemos con los proveedores de Delphi 6 y las técnicas de caché de datos. Por el mismo motivo, pospondremos el estudio de las actualizaciones a través de DB Express.

### Los conjuntos de datos en DB Express

El diagrama que sigue a este párrafo muestra la rama de la jerarquía de herencia de la VCL que incluye a los conjuntos de datos de DB Express:



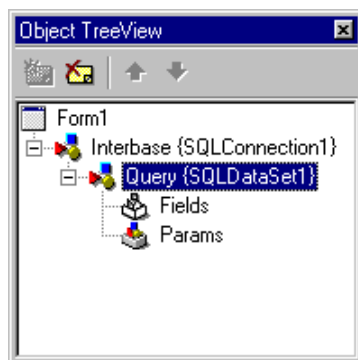
Hay dos ramas independientes: la que cuelga de *TCustomSQLDataSet*, que es la que contiene los componentes que nos interesan ahora, y la que desciende de *TCustomCachedDataSet*, de la que sólo nos va a interesar *TSQLClientDataSet*. Este último componente combina en su interior un conjunto de datos cliente como los que hemos visto en MyBase, con otro conjunto de datos interno, pero de DB Express. Los datos se obtienen a través de DB Express, pero se guardan temporalmente, e incluso se pueden editar, en la caché de memoria del *TSQLClientDataSet*. Es una situación se va a repetir en Delphi con los componentes del BDE y de IB Express, por lo que conviene que se vaya familiarizando con este tipo de ecuaciones:



Aunque parezca más lógico estudiar el uso de *TSQLClientDataSet* en este mismo capítulo, no lo haremos porque el citado componente es apropiado sólo para trabajar con esquemas de datos sencillos. Si queremos aprovechar relaciones más complejas,

como la relación maestro/detalles que sí veremos aquí, hay que recurrir explícitamente a los componentes de DataSnap que el conjunto de datos cliente esconde.

La primera propiedad que hay que configurar para cualquiera de estos componentes es *SQLConnection*, una propiedad común. En tiempo de diseño, podemos utilizar el Inspector de Objetos para seleccionar una de las conexiones existentes a partir de una lista desplegable asociada a la propiedad *SQLConnection*. Pero si somos hábiles moviendo el ratón, quizás sea más sencillo utilizar el árbol de objetos que en Delphi 6 suele encontrarse sobre el Inspector de Objetos. Si al traer un conjunto de datos de DB Express desde la Paleta de Componentes, lo dejamos caer sobre un nodo de conexión de dicho árbol, al nuevo componente se le asigna automáticamente un puntero a la conexión dentro de su *SQLConnection*:



Recuerde que puede utilizar la combinación SHIFT+ALT+F11 si el Arbol de Objetos se encuentra oculto tras alguna ventana.

Pasemos a clasificar los componentes que utilizaremos. Cuando estudiemos el BDE, veremos que existe una gran diferencia en el comportamiento de sus tres tipos de conjuntos de datos. Por ejemplo, el algoritmo de recuperación de datos de un *TQuery* es el más sencillo: la interfaz de recuperación es unidireccional, y el componente almacena automáticamente los registros en una caché interna. En contraste, el *TTable* del BDE utiliza una técnica más compleja, que le permite manejar tablas muy grandes sin necesidad de almacenar una copia de todos los registros en el lado cliente.

Sin embargo, en DB Express no existen diferencias tan abrumadoras entre sus clases de conjuntos de datos. El núcleo del mecanismo de recuperación se implementa ya en la clase abstracta *TSQLCustomDataSet*, que sirve de base a los otros cuatro componentes. De ellos, *TSQLDataSet* es el que más se parece a su padre<sup>26</sup>, pues se limita a publicar algunas de las propiedades definidas por su ancestro en la sección protegida. De todas ellas, las más importantes son estas dos:

---

<sup>26</sup> O a su madre, porque asignarle sexo a un componente es más complicado que elegir color para la canastilla de un pollo.

```

type
    TSQLCommandType = (ctQuery, ctTable, ctStoredProc);

property TCustomSQLDataSet.CommandType: TSQLCommandType;
property TCustomSQLDataSet.CommandText: string;

```

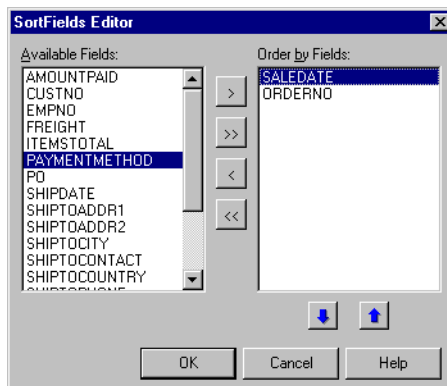
Con *CommandType* indicamos al componente cómo vamos a identificar el origen de los datos. En dependencia del valor que tenga, se interpreta el valor almacenado en *CommandText* de una forma u otra:

Valor	Significado de CommandText
<i>ctQuery</i>	Una instrucción SQL, que puede ocupar una o varias líneas
<i>ctTable</i>	Un nombre de tabla
<i>ctStoredProc</i>	Un nombre de procedimiento almacenado

Pero esta abundancia de opciones no debe llevarle a pensar que existen técnicas de recuperación diferentes para cada una de ellas. Por ejemplo, cuando utilizamos el tipo de comando *ctTable* el componente se limita a generar por nosotros la sentencia SQL correspondiente. Si asignamos *ORDERS* en el nombre de tabla, al servidor se envía la siguiente consulta:

```
select * from ORDERS
```

Cuando usamos *ctTable*, tenemos una propiedad adicional, llamada *SortFieldNames*, que nos permite añadir una cláusula **order by** a la instrucción. La siguiente imagen muestra el editor asociado a la propiedad:



Aquí volvemos a encontrarnos con problemas en la documentación. El primero de ellos es que la ayuda en línea indica con total claridad que se utiliza el punto y coma para separar los campos, si utilizamos un criterio de ordenación compuesto. No es cierto: si queremos ordenar por dos campos, como en la imagen anterior, el propio editor de la propiedad deja el siguiente valor en *SortFieldNames*:

```
SaleDate,OrderNo
```

Esto es, que tenemos que separar los campos por comas; compruebe si lo desea, que los puntos y comas no son aceptados. Acabamos de ver una mentira, pero ahora descubriremos una verdad a medias. SQL nos ofrece dos direcciones para los criterios de ordenación. ¿Cómo podríamos pedirle a DB Express que ordene los pedidos primero por código de cliente, y luego por la fecha de venta, pero en orden descendente? Aunque he estrujado la documentación, no he encontrado pista alguna. Pero fue muy fácil someter al componente a una sesión de “prueba y error”. Resulta que podemos incluir las cláusulas **asc** y **desc** para indicar el sentido de la ordenación. Por ejemplo:

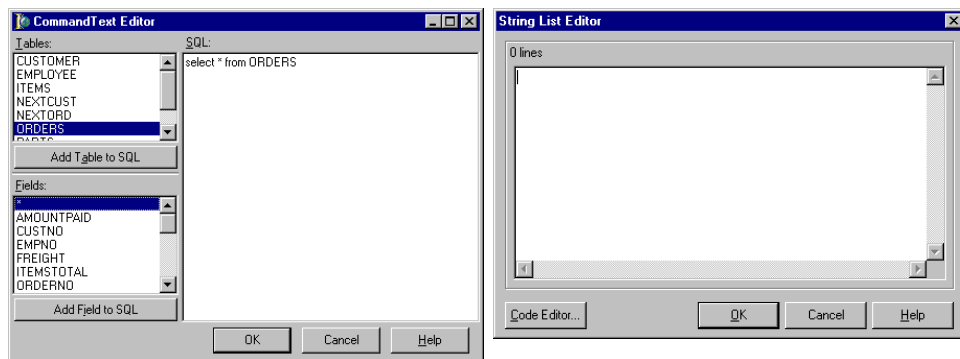
```
SQLDataSet1.SortFieldNames := 'CustNo asc, SaleDate desc';
```

Cuando *CommandType* es *ctStoredProc*, tendremos con casi total seguridad que trabajar también con la propiedad *Params*. Pero dejaremos el estudio de los parámetros para más adelante.

En resumen: el modo fundamental de trabajo de *TSQLDataSet* consiste en enviar al servidor una cadena que contiene una instrucción SQL. Pero, modificando el valor de la propiedad *CommandType*, podemos dejar que sea el propio componente el que genere la instrucción completa en beneficio nuestro.

Y ya estamos en condiciones de comprender por qué existen los componentes adicionales *TSQLQuery*, *TSQLTable* y *TSQLStoredProc*. Se trata simplemente de disfraces que adopta la clase base *TCustomSQLDataSet* para que la configuración de propiedades sea más sencilla, una vez que hemos decidido el tipo de comando que necesitamos.

Consideremos el componente *TSQLQuery*: nos ofrece una sola propiedad de tipo *TStrings*, adecuadamente llamada *SQL*, para que tecleemos la instrucción SQL que más nos apetezca. Si quisiéramos hacer lo mismo con *TSQLDataSet*, tendríamos que modificar dos propiedades diferentes: *CommandType* y *CommandText*. Sin embargo, para mí lo mejor de todo está en los editores de propiedades, en tiempo de diseño, de estos dos componentes. Observe las siguientes imágenes:



La imagen de la izquierda aparece al hacer doble clic sobre el *CommandText* de un componente *TSQLDataSet*; la de la derecha corresponde a la propiedad *SQL* de un

*TSQLQuery*. Aparentemente, a la izquierda tenemos un útil asistente que deja en muy mal lugar el desabrido editor genérico de la derecha, ¿verdad? Sin embargo, el primero de los editores se ejecuta de forma modal, mientras que si pulsamos sobre el botón *Code Editor* en el segundo, podremos editar cómodamente nuestra instrucción SQL en la ventana de edición de Delphi, en paralelo con otros ficheros. Podríamos, por ejemplo, cargar simultáneamente un fichero con un *script* de creación de las tablas de la base de datos, para consultarlo. La intención del que programó el editor de la izquierda era muy buena, pero el resultado nos limita agobiantemente. Es un ejemplo más del típico compromiso entre protección bajo tutela y libertad...

## Nuevos tipos de campos

El funcionamiento de los objetos de acceso a campos en DB Express es idéntico al de los conjuntos de datos clientes. La única particularidad es la adición de dos nuevas clases de campos:

- *TSQLTimeStampField*: Se utiliza para las columnas que almacenan fecha y hora.
- *TFMTBCDField*: ¡Seis siglas juntas! A este paso, sólo los alemanes podrán programar en Delphi. Esta clase se utiliza para campos de tipo numérico en los que el uso de *TBCDField* nos haría perder precisión.

La segunda adición está plenamente justificada. El tipo *TBCDField* almacena sus valores internamente en una variable de tipo *Currency*: 64 bits para un valor entero, que dan aproximadamente para 20 dígitos de precisión (suficiente para InterBase), pero solamente cuatro dígitos después de la coma decimal. No obstante, no desaparece *TBCDField*. En primer lugar, para no fastidiar la compatibilidad con las aplicaciones que ya lo utilizan, pero también porque la aritmética con el tipo *Currency* es más rápida que con el verdadero tipo BCD almacenado por la nueva clase de campo.

Pero no alcanzo a comprender por qué Borland ha añadido una nueva clase de campo para representar fechas y horas. En la ayuda se explica que ha sido necesario por el formato tan particular que tienen los controladores DB Express para representar las columnas de este tipo. Y yo me pregunto: DB Express es nuevo, ¿no podían haberlo diseñado de otra forma? Porque ahora vamos a tener problemas para que una aplicación escrita para SQL Server pueda adaptarse para trabajar con Oracle, incluso si utilizamos un módulo DataSnap de capa intermedia.

No, no puedo estar contento con esta absurda decisión.

## Carga de datos en un control de listas

¿Qué se puede hacer con un conjunto de datos que solamente se mueve en una dirección? Muchas cosas, pero pocas si exigimos además que el ejemplo consuma pocas líneas de código. Vamos a crear una sencilla aplicación que presentará los datos de una tabla que determinemos sobre un control de listas (*TListView*) como el que se

utiliza para las listas de ficheros dentro de un directorio, en el panel de la derecha del Explorador de Windows.

Aunque podríamos particularizar el ejemplo para una tabla o consulta en específico, intentaremos que el código fuente sea lo más genérico posible, y que pueda funcionar con cualquier tabla. En concreto, y por simplicidad, vamos a traer un componente *TSQLTable*, al que renombraremos como *Datos*; las tablas que utilizaremos en nuestros ejemplos no son excesivamente grandes. Pero tenga presente que DB Express no tiene ningún secreto mágico para que podamos, sin más, abrir una tabla de 5 millones de registros y traerlos todos a la aplicación.

Traiga también un *TListView* al formulario, y asigne el valor *vsReport* en su propiedad *ViewStyle*. Este modo de visualización es el que da al control un mayor parecido con una rejilla de datos. Podríamos ahora definir columnas en tiempo de diseño pero, como he dicho antes, vamos a crearlas en tiempo de ejecución a partir de las propiedades de los campos que contenga la tabla.

Comencemos por el método de más alto nivel, que se encargará de abrir la tabla, recorrer los registros (¡en una sola dirección!) y copiarlos en el control:

```

procedure TwndPrincipal.CargarTabla;
begin
    ListView1.Items.BeginUpdate;
    try
        CrearColumnas;
        Datos.Open;
        try
            while not Datos.Eof do           // ;No se llama a First!
                begin
                    LeerFila;
                    Datos.Next;
                end;
            finally
                Datos.Close;
            end;
        finally
            ListView1.Items.EndUpdate;
        end;
    end;

```

Las llamadas a los métodos *BeginUpdate* y *EndUpdate* son indispensables. Cuando añadimos un elemento a una lista o colección, es muy común que se dispare un evento interno del control para redibujar la zona que controla en pantalla. Imagine qué sucedería si para añadir 50 líneas tuviéramos que borrar y repintar la superficie de la lista el mismo número de veces. Los métodos mencionados anulan temporalmente este mecanismo, utilizando un contador de referencias, y posteriormente lo vuelven a habilitar.

El siguiente método, *CrearColumnas*, elimina las columnas que podían ya existir y crea un nuevo conjunto de ellas, deduciendo sus propiedades a partir de las propiedades de cada campo de la tabla.

```

procedure TwndPrincipal.CrearColumnas;
var
  I: Integer;
  F: TField;
begin
  ListView1. Clear;
  ListView1.Columns.Clear;
  ListView1.ViewStyle := vsReport;
  for I := 0 to Datos.FieldCount - 1 do
    begin
      F := Datos.Fields[I];
      with ListView1.Columns.Add do
        begin
          Caption := F.DisplayLabel;
          Width := Ancho(F);
          Alignment := F.Alignment;
        end;
    end;
  end;

```

Para determinar el ancho que debe tener inicialmente la columna, he utilizado un método auxiliar, *Ancho*, que incluyo a continuación:

```

function TwndPrincipal.Ancho(F: TField): Integer;
var
  AvgText: string;
begin
  SetLength(AvgText, F.DisplayWidth + 2);
  FillChar(AvgText[1], Length(AvgText), 'n');
  Result := Max(
    ListView1.StringWidth(AvgText),
    ListView1.StringWidth(F.DisplayLabel + 'nn'));
end;

```

*StringWidth* es un método muy oportuno de los controles de listas que calcula el ancho en píxeles necesario para dibujar una cadena con el tipo de letra en uso. Como referencia para estimar el ancho necesario, comenzamos por calcular el espacio necesario para la etiqueta de visualización del campo, *DisplayLabel*, que utilizaremos en la cabecera de la columna. Para dejar espacios antes y después de la etiqueta, he incluido dos letras *n* en el cálculo; es una práctica común en tipografía. Se supone que la letra más “ancha” es la *m*, y que la *n* representa la anchura media del juego de caracteres. Este cálculo nos da una idea del ancho de la cabecera. Para estimar el ancho de los datos en la columna, utilizamos la propiedad *DisplayWidth*, añadiendo las dos consabidas letras *n* como margen. La función *Max* se define en la unidad *Math*, que tendrá que incluir en la cláusula **uses**.

Por último, he aquí la implementación del método *LeerFila*:

```

procedure TwndPrincipal.LeerFila;
var
  I: Integer;
  F: TField;
  Item: TListItem;
begin
  Item := nil;

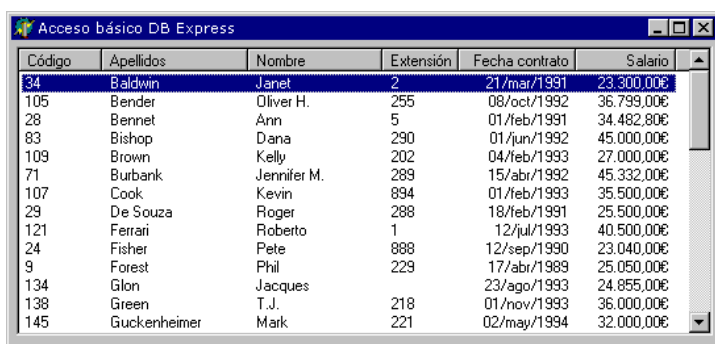
```

```

for I := 0 to Datos.FieldCount - 1 do
begin
    F := Datos.Fields[I];
    if not Assigned(Item) then
    begin
        Item := ListView1.Items.Add;
        Item.Caption := F.DisplayText;
    end
    else
        Item.SubItems.Add(F.DisplayText)
    end;
end;
end;

```

Quiero resaltar el uso de la propiedad *DisplayText* para obtener la representación del valor almacenado en el campo. La imagen que aparece a continuación corresponde a nuestra aplicación en funcionamiento:



Código	Apellidos	Nombre	Extensión	Fecha contrato	Salario
34	Baldwin	Janet	2	21/mar/1991	23.300,00€
105	Bender	Oliver H.	255	08/oct/1992	36.799,00€
28	Bennet	Ann	5	01/feb/1991	34.482,80€
83	Bishop	Dana	290	01/jun/1992	45.000,00€
109	Brown	Kelly	202	04/feb/1993	27.000,00€
71	Burbank	Jennifer M.	289	15/abr/1992	45.332,00€
107	Cook	Kevin	894	01/feb/1993	35.500,00€
29	De Souza	Roger	288	18/feb/1991	25.500,00€
121	Ferrari	Roberto	1	12/jul/1993	40.500,00€
24	Fisher	Pete	888	12/sep/1990	23.040,00€
9	Forest	Phil	229	17/abr/1989	25.050,00€
134	Glon	Jacques		23/ago/1993	24.855,00€
138	Green	T.J.	218	01/nov/1993	36.000,00€
145	Guckenheimer	Mark	221	02/may/1994	32.000,00€

### ADVERTENCIA

Como se observa en la imagen, la primera columna de un *TListView* ignora la alineación que le asignemos. Si quisiéramos corregir esta “imperfección”, tendríamos que interceptar el evento *OnDrawItem* para dibujar el texto nosotros mismos.

## Consultas paramétricas

Casi nunca, en la programación cliente/servidor, se necesita realmente que una consulta devuelva todos los registros de una tabla. Casi siempre habrá alguna condición que limite el conjunto de registros, y en la mayoría de los casos, esa condición dependerá de uno o más parámetros.

Haga una copia, en un nuevo directorio, de la aplicación que acabamos de programar. Vaya al formulario principal y borre el componente *TSQLTable*, llamado *Datos*. Añada entonces un *TSQLQuery* y asícielo a una conexión con la base de datos *mastsql.gdb*. Cambie también su nombre a *Datos*, para que el procedimiento *CargarTabla* siga funcionando igual que antes. Asigne la siguiente instrucción en su propiedad *SQL*:

```

select *
from   CUSTOMER

```



**where** Company **starting with** :prefijo

¡Mucho cuidado! El parámetro es *:prefijo*, y se identifica por los dos puntos iniciales. No pueden existir espacios entre los dos puntos y el resto del identificador. Hay quienes, por inercia, asocian los dos puntos en expresiones como la que aparece a continuación con el signo de igualdad, erróneamente, por supuesto:

```
Company = :prefijo /* El error sería Company =: prefijo */
```

Ahora haga doble clic sobre la propiedad *Params* del componente, y compruebe que hay un elemento dentro de la colección. Este elemento pertenece a la clase *TParam*, y es uno de los componentes más caóticos de Delphi. Cuando aparecieron los primeros componentes de acceso que no dependían del BDE, cada equipo o desarrollador utilizó el sistema de parámetros que le vino en ganas. Es cierto que el tratamiento de los parámetros varía mucho de acuerdo a la interfaz de acceso que se utilice, pero no hubo ni la más mínima coordinación al respecto. En Delphi 6 se ha intentado paliar esta dificultad, pero sólo parcialmente. Quedan aún dos sistemas diferentes de parámetros:

- 1 El utilizado por ADO Express (perdón, por dbGo™). Cada parámetro es de tipo *TParameter*, y se almacenan en una colección de tipo *TParameters*.
- 2 El utilizado por el resto de los componentes. Cada parámetro es de tipo *TParam*, y la colección que los contiene pertenece a la clase *TParams*.

Nos limitaremos por ahora al estudio de *TParam*. Sus propiedades son:

Propiedad	Significado
<i>Name</i>	El nombre del parámetro, que coincide con el identificador utilizado en la consulta.
<i>ParamType</i>	Tiene un significado especial para los procedimientos almacenados. En una consulta, siempre es de tipo <i>ptInput</i> .
<i>DataType</i>	El tipo de dato del parámetro.
<i>Size</i>	Cuando es de tipo <b>string</b> , el número de caracteres.
<i>Precision</i>	Cuando es de tipo BCD o FMTBCD, el número total de dígitos, y la
<i>NumericScale</i>	cantidad de dígitos a la derecha de la coma.
<i>Value</i>	Almacena en un <i>Variant</i> el valor del parámetro.

Es muy importante que memorice que DB Express no puede adivinar el tipo de los parámetros que tecleamos en una consulta. Menciono esta “falta” porque ADO sí puede echarnos una mano en esto. Por lo tanto, cuando tecleamos una instrucción con parámetros en una consulta, debemos ir inmediatamente al parámetro y asignarle el tipo de datos al que pertenece en su propiedad *DataType*.

En la tabla anterior menciono la existencia del parámetro *Value*, de tipo variante, accesible en tiempo de diseño. Podemos aprovecharlo para asignar un valor inicial en un parámetro. Pero lo más frecuente es que hagamos las asignaciones de valores en

tiempo de ejecución, y entonces tendremos otras propiedades más útiles para este propósito.

Para probar el cambio de valores de un parámetro, añada a la nueva aplicación un componente *TToolBar*, de la página *Win32*. Deje caer un *TEdit* sobre este componente; luego, pulse el botón derecho del ratón sobre la barra y ejecute los comandos *New separator* y *New button*, para añadir un separador y un botón (¡qué otra cosa podría ser!). Bautice al botón con el nombre de *bnBuscar*. Y arregle un poco el resultado para que tenga la siguiente apariencia:



La secuencia típica de cambio de parámetros se parece a la siguiente:

```
procedure TwndPrincipal.bnBuscarClick(Sender: TObject);
begin
    Datos.Close;
    Datos.Params[0].AsString := Edit1.Text;
    Datos.Open;
    CargarTabla;
end;
```

Para asignar valores en parámetros en tiempo de ejecución, la consulta debe estar inactiva. Podemos entonces utilizar cualquiera de las propiedades *AsString*, *AsInteger*, *AsFloat*, etcétera, del parámetro para modificar su valor. Para terminar, debemos activar nuevamente la consulta.

Hay algo que no me gusta en el ejemplo anterior: estoy haciendo referencia a los parámetros por medio de su posición. Al igual que sucede con los campos, existe una función *ParamByName* que permite recuperar un parámetro por su nombre, sin distinguir entre mayúsculas y minúsculas:

```
procedure TwndPrincipal.bnBuscarClick(Sender: TObject);
begin
    Datos.Close;
    Datos.ParamByName('prefijo').AsString := Edit1.Text;
    Datos.Open;
    CargarTabla;
end;
```

## Parámetros repetidos

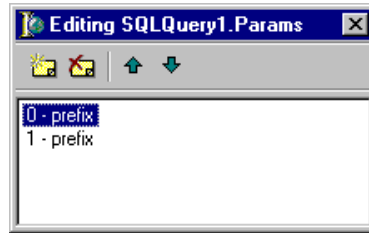
A primera vista, puede parecer peligrosa la existencia de una función como *ParamByName*. Analice la siguiente instrucción con parámetros:

```

select *
from   EMPLOYEE
where  LastName starting with :prefijo or
       FirstName starting with :prefijo

```

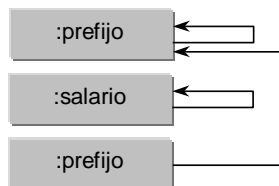
Queremos conocer qué clientes tienen nombres o apellidos que comienzan por un prefijo tecleado por el usuario. Si tecleamos esa instrucción dentro de un componente *TSQLQuery* y examinamos el contenido de *Params*, encontraremos que *prefijo* aparece dos veces en la lista de parámetros:



Supongamos ahora que vamos a asignar valores a los parámetros. Si accedemos a los parámetros a través de su posición, poca duda cabe que si realizamos las dos asignaciones no tendremos nada que temer. Pero, ¿y si utilizamos *ParamByName*? Es natural pensar entonces que *ParamByName* va a devolvernos arbitrariamente uno de los dos parámetros, y que si nos limitamos a una asignación tendremos problema con el segundo valor. ¿Certo?

... he dicho que el razonamiento anterior es “natural”, pero no que sea correcto. Haga un experimento: cree un componente de consulta con las características explicadas, y asigne solamente el tipo de dato para el primer parámetro, en su propiedad *DataType*. Seleccione el segundo parámetro: ¡el valor de *DataType* ha cambiado simultáneamente! Lo mismo sucederá cuando modifique *Value*. Cuando me di cuenta de este fenómeno por primera vez, pensé que Delphi lo resolvía recorriendo la colección y repitiendo la asignación en todos los parámetros de igual nombre. Pero examinando el código fuente, me di cuenta que Borland había elegido una solución más interesante:

Propiedad *ParamRef*



Como muestra el diagrama, cada parámetro tiene una propiedad privada llamada *ParamRef*, que es una referencia a un parámetro. Internamente, la clase *TParam* la implementa con la ayuda de una función y un atributo. La primera vez que se llama a

*ParamRef*, el parámetro busca en la colección a la cual pertenece el primer parámetro que tiene su mismo nombre, y almacena esa referencia en el atributo interno. De ahí en adelante, no hace falta recalculiar el valor de la propiedad.

Cuando el parámetro ocurre una sola vez, *ParamRef* apunta al propio objeto, como en el caso del parámetro *salario* del diagrama anterior. Sucede lo mismo con la primera ocurrencia de los parámetros repetidos. En cambio, la segunda ocurrencia de *prefijo* apunta al primer parámetro. A partir de aquí, es fácil de entender que todas las propiedades significativas de *TParam* se implementan a través de *ParamRef*. Si asignamos un valor en el segundo *prefijo*, el valor irá a parar internamente a su primera ocurrencia. Recíprocamente, cuando preguntemos cuál es el valor del segundo *prefijo*, Delphi buscará nuevamente la respuesta en el primero de ellos. Ingenioso, ¿verdad?

## Generación explícita de sentencias SQL

Usted podrá preguntarse: ¿para qué complicarnos esta hermosa mañana (en este momento, hay una mañana hermosa en algún lugar del mundo) con todo este rollo de los parámetros? En definitiva, la selección de clientes de acuerdo al nombre de la empresa podría haber sido implementada de la siguiente manera:

```
procedure TwndPrincipal.bnBuscarClick(Sender: TObject);
begin
  Datos.Close;
  Datos.SQL.Text := Format(
    'select * from CUSTOMER where Company starting with %s',
    [QuotedStr(Edit1.Text)]);
  Datos.Open;
  CargarTabla;
end;
```

En este ejemplo estamos generando explícitamente *toda* la sentencia SQL, y la asignamos dentro de la propiedad *SQL* de la consulta. Debemos recordar que *SQL* es una propiedad de tipo *TStrings*, una lista de cadenas, y que por lo tanto no permite que le asignemos directamente una cadena de caracteres. El mismo ejemplo es presentado en muchos libros en la siguiente forma alternativa:

```
// ...
Datos.Close;
Datos.SQL.Clear;
Datos.SQL.Add(Format(
  'select * from CUSTOMER where Company starting with %s',
  [QuotedStr(Edit1.Text)]));
Datos.Open;
// ...
```

Reconozca, no obstante, que es más fácil asignar la cadena en la subpropiedad *Text*, sin más rodeos. Reconozcamos también que, aunque este método de modificar la consulta es correcto, ya vamos encontrando problemas. Por ejemplo, ¿qué pinta esa función *QuotedStr* aplicada sobre la cadena tecleada por el usuario? Su presencia es necesaria porque el segundo operando de **starting with** debe ser una constante de

cadena, en este caso... y las constantes de cadenas deben ir encerradas entre comillas. ¿Por qué no me he limitado entonces a incluir las comillas en esta forma, directamente en la cadena de formato?

```
'select * from CUSTOMER where Company starting with '%"s'''
```

Para encontrar la respuesta, piense en lo que pasaría si el usuario teclease un prefijo que tuviese comillas, como el siguiente:

```
Diver's
```

En ese caso, la comilla del prefijo fastidiaría el formato de la cadena enviada al servidor SQL, y se produciría un error de sintaxis. En cambio, *QuotedStr* maneja bien estos casos, y duplica las comillas interiores.

En definitiva, lo que quería explicar es que si generamos las instrucciones SQL a mano, nos encontraremos con toda una larga serie de problemas relacionados con el tipo de las constantes: tendremos que saber cuándo hay que encerrar o no el valor entre comillas, y tendremos que hacerlo correctamente cuando sea necesario.

## Compilación de consultas

Pero la principal justificación para el uso de parámetros viene de otro frente. Aunque las sentencias SQL suelen ser de pequeña longitud, necesitan ser compiladas por el servidor para que puedan ser ejecutadas. A primera vista, parece sencillo compilar un **select** como los anteriores; en la práctica, encontraremos que es una operación costosa en términos de tiempo:

- 1 Hay que buscar la información sobre tablas, columnas y sus tipos en el catálogo del sistema, es decir, en otras tablas.
- 2 Normalmente, el servidor intentará optimizar la consulta. No solamente necesitará evaluar muchas alternativas de implementación, sino que también es usual que se consulten datos estadísticos asociados al uso de índices y tablas.

Cada vez que enviamos una consulta al servidor por primera vez, tiene lugar el penoso proceso de compilación. Cuando se reenvía la misma consulta, lo que sucede dependerá de lo listo o lo barato que sea el servidor. Pero, en cualquier caso, si primero pedimos los clientes cuyos nombres comienzan con *B* y luego los que comienzan con *A*, con toda seguridad tendrán lugar dos compilaciones.

Los parámetros abren una vía de escape, porque para compilar y optimizar una consulta con parámetros no es necesario que estos tengan valores concretos. Es posible generar un algoritmo eficiente para evaluar “el conjunto de clientes cuyos nombres comienzan con determinado prefijo”. Se pueden ejecutar secuencias de acciones como la siguiente:



Dentro de poco veremos cómo enlazar dos tablas o consultas en una relación de dependencia, la denominada relación maestro/detalles, y comprenderemos la importancia de la compilación previa de la consulta.

## Preparación en DB Express

Prepárese usted, no la consulta, pero para otro disgusto con la documentación. Toda esta cantilena sobre lo importante de compilar las consultas paramétricas que se van a abrir y cerrar mucho es también aplicable a cualquier otra interfaz de acceso a datos, como ADO, InterBase Express ... y nuestro querido aunque decadente BDE.

Supongamos que tenemos una consulta con parámetros en una aplicación que utiliza el BDE, y que esa consulta se abre y se cierra varias veces, en forma consecutiva. El siguiente esquema representa la apertura y cierre de la consulta ejecutada dos veces. En los bloques más pequeños, muestro lo que sucede cada vez:



Cada vez que se ejecuta la secuencia de apertura y cierre, la consulta llama internamente a la rutina de compilación, que he representado con su nombre en inglés: *Prepare*. Acto seguido, se procede a la ejecución real, por decirlo de algún modo, de la instrucción: *Execute*. Y cuando se cierra la consulta, internamente se dispara *Unprepare*, que surte el efecto contrario a *Prepare*; supuestamente, libera las estructuras de datos creadas en el servidor durante la compilación. Este último paso puede ser muy importante en algunos servidores que no controlan muy bien el tiempo de vida de esas estructuras.



Este otro diagrama, en cambio, muestra lo que sucede cuando se llama explícitamente a *Prepare* antes de iniciar la secuencia: ya no es necesario repetir innecesariamente la preparación/finalización para cada paso. El tiempo que se llega a ganar puede ser muy significativo.

Pues bien, la documentación de DB Express sugiere que algo similar sucede cuando asignamos *True* en la propiedad *Prepared* de un conjunto de datos de DB Express;

puede ser también en una tabla o en un procedimiento almacenado. Sin embargo, la práctica muestra algo muy diferente. Voy a describir un experimento que podrá repetir con mucha facilidad: configure una consulta con parámetros y programe un bucle con un número suficientemente alto de repeticiones. En el interior del bucle, asigne parámetros de forma más o menos aleatoria, abra y cierre la consulta. Mida entonces el tiempo total, con la consulta preparada o sin preparar. Los resultados que obtendrá serán bastante confusos, por regla general, porque influirá más el estado de la caché de InterBase que la preparación de la consulta.

Ese resultado me desconcertó y repetí el experimento controlando la interacción con el servidor a través del componente *TSQLMonitor*. Para no alargar la historia, le adelantaré que no encontré *ni rastros* de la compilación repetida que sí realiza el BDE. Es una buena señal: aunque solamente un análisis minucioso del código fuente me daría una seguridad completa, al parecer DB Express realiza la factorización de las compilaciones sin nuestra ayuda, al menos en InterBase.

Entonces, ¿qué hace la llamada a *Prepare*? Obsérvelo usted mismo. Este listado, obtenido con la ayuda de *TSQLMonitor*, muestra lo que sucede cuando se ejecuta una consulta *sin preparación* previa:

```

INTERBASE - isc_attach_database
INTERBASE - isc_dsql_allocate_statement
INTERBASE - isc_start_transaction
               select count(*) from orders
               where saledate between ? and ?
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_allocate_statement
SELECT 0, '', '', A.RDB$RELATION_NAME, A.RDB$INDEX_NAME,
        B.RDB$FIELD_NAME, B.RDB$FIELD_POSITION, '', 0,
        A.RDB$INDEX_TYPE, '', A.RDB$UNIQUE_FLAG,
        C.RDB$CONSTRAINT_NAME, C.RDB$CONSTRAINT_TYPE
FROM    RDB$INDICES A, RDB$INDEX_SEGMENTS B FULL OUTER JOIN
        RDB$RELATION_CONSTRAINTS C
ON A.RDB$RELATION_NAME = C.RDB$RELATION_NAME AND
   C.RDB$CONSTRAINT_TYPE = 'PRIMARY KEY'
WHERE   (A.RDB$SYSTEM_FLAG <> 1 OR A.RDB$SYSTEM_FLAG IS NULL) AND
        (A.RDB$INDEX_NAME = B.RDB$INDEX_NAME) AND
        (A.RDB$RELATION_NAME = UPPER('orders'))
ORDER BY RDB$INDEX_NAME
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_fetch
INTERBASE - isc_commit_retaining
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_fetch
INTERBASE - isc_commit_retaining
INTERBASE - isc_dsql_free_statement

```

Si a continuación se ejecuta la misma consulta, pero asignando antes *True* en su propiedad *Prepared*, he aquí la traza obtenida:

```
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_fetch
INTERBASE - isc_commit_retaining
INTERBASE - isc_dsql_free_statement
```

Todo apunta a que *Prepared* evita la ejecución repetida de esa grande y casi ilegible sentencia SQL del primer listado. Estupendo, pero ¿qué demonios hace ese **select** aparte de devorar el ancho de banda disponible en la red?

## El misterio de los metadatos

Para saber qué utilidad tiene la extraña consulta que se lanza a nuestras espaldas, podemos copiar su texto y ejecutarlo desde la Consola de InterBase. Basta, sin embargo, una simple ojeada y algo de sentido común para comprender que la instrucción recupera entre otras cosas todos los índices que están definidos sobre la tabla base. ¿La tabla base? ¡Estábamos trabajando con una consulta! Al parecer, DB Express elige siempre la primera tabla que aparece en la cláusula **from** más externa. Haga la prueba con una consulta como la siguiente:

```
select *
from   CUSTOMER c join ORDERS o
      on c.custno = o.custno
```

Si la teclea exactamente como muestro, la consulta sobre índices mencionará la tabla *CUSTOMER*; si invierte el orden de las tablas, utilizará *ORDERS*. Aparentemente, se trata de una decisión de diseño del tipo “hacerlo lo mejor posible”. La información sobre índices y columnas de la clave primaria es de suma utilidad para DataSnap, cuando hay que determinar cómo se actualiza el resultado de una consulta; eso lo veremos al estudiar el algoritmo de resolución en DataSnap. No siempre es posible actualizar una consulta, y no se puede determinar mecánicamente todos los casos en que se puede o no. Pero cuando se puede, lo común es que las actualizaciones afecten a una sola de las tablas del *join*, si es que la consulta es un *join*. Por lo tanto, DB Express ha decidido adelantar alguna información en beneficio de DataSnap... por si acaso es útil.

A este tipo de información, que no tiene que ver directamente con el propósito de la consulta, que es leer registros y nada más, se le denomina genéricamente como *metadatos*, o datos sobre el esquema relacional. La propiedad *NoMetadata* es común a los cuatro tipos de conjuntos de datos de DB Express, y determina si se deben leer esos registros adicionales o no.

### MAL DISEÑO

El nombre de la propiedad anterior, *NoMetadata*, es un caso flagrante de mal diseño. ¿Puede usted decirme, sin pensar, rápidamente, si los metadatos se leen cuando *NoMetadata* vale *False*? El programador que creó el componente quiso aprovechar



que las variables booleanas se inicializan a falso en Delphi por omisión para ahorrarse una asignación en el constructor de la clase, y una cláusula **default** en la declaración de la propiedad. Estupendo ahorro.

Es tiempo ya de resumir. ¿Quiere decir que tenemos que tenemos que hacer que *Prepared* valga *True* para evitar que la lectura de los metadatos se repita innecesariamente? ¡Tampoco! Y ya sé que me va a odiar, y que esta información parece contradecir el “resultado” del experimento que expliqué en la anterior sección; sólo le pediré un poco más de paciencia...

Abra y cierre varias veces una consulta paramétrica sin preparar. Esto puede hacerlo incluso en tiempo de diseño, acoplando además un *TSQLMonitor* a la conexión, y cerciorándose de que esté activo. Aunque cambie el valor de los parámetros, comprobará que no se repite la consulta de los metadatos. ¿Por qué, entonces, en el experimento anterior, se repetía la consulta de metadatos al utilizar una consulta sin preparar? Pues por un pequeño detalle que distorsionó el experimento: si se asigna *False* a *Prepared*, la información sobre los metadatos desaparece y hay que volver a leerla.

## RESUMEN

Utilizando InterBase con DB Express, no he encontrado ningún tipo de mejora “explicable” en el funcionamiento de las consultas con parámetros. Observe que estoy hablando de InterBase. Con Oracle 8.1.7 no hay diferencia alguna en la traza, aunque se prepare o no la consulta; es posible que con DB2 o MySQL sí se note alguna mejoría. Esto no quiere decir que DB Express esté desaprovechando una oportunidad de mejorar su rendimiento; por el contrario, significa que esa optimización se encuentra incorporada en el propio código del componente. Mi consejo: si no es mucha molestia, asigne *True* a la propiedad *Prepared* al principio de la aplicación, solamente una vez. Hágalo aunque sea por fe. Probablemente no surta ningún efecto positivo, pero quién sabe...

## Parámetros creados al vuelo

No obstante, seguirán existiendo situaciones en las que sea más recomendable generar el texto completo de la consulta SQL. Suponga que tiene que implementar una ventana de búsqueda de pedidos. Los criterios de búsqueda que solicita su cliente son muy dispares: hay que permitir la búsqueda por fecha de venta, por fecha de envío, por cliente, por la fase de la Luna... En vez de crear innumerables consultas especializadas, es mejor generar dinámicamente la instrucción **select** correspondiente.

El problema está en que la generación de código SQL puede ser más difícil de lo que aparenta, por culpa principalmente de los valores constantes. Por ejemplo, ¿cómo se representa una constante de fecha en InterBase? ¿Cómo se representa en SQL Server? ¡Mucho cuidado con responder sin pensar! Considere la siguiente consulta, que hace referencia a una de las bases de datos que vienen con SQL Server:

```
select * from sales
where ord_date < '1/14/1994'
```

¿Es correcta o incorrecta? En el ordenador donde escribo ahora funciona sin problemas... porque el SQL Server instalado está en inglés americano, que utiliza el orden mes/día/año para las fechas. Si estuviese configurado en castellano, produciría un error. Claro, si la aplicación va a utilizar *siempre* el mismo servidor, se puede zanjar el asunto con el universal algoritmo de “prueba y error”, pero si es una aplicación que deba ejecutarse en varias instalaciones, podemos meternos en un berenjenal.

En dos palabras: generar constantes dentro de una sentencia SQL es un mal rollo. Pero podemos evitarlo sin renunciar a la generación dinámica: basta que utilicemos parámetros en la sentencia que creemos. He ampliado el ejemplo de búsqueda anterior para que, en otra ventana, nos permita mostrar las ventas en un año determinado:



ORDERNO	CUSTNO	SALEDATE
1102	1231	06/06/92
1103	3042	13/07/92
1104	1234	10/07/92

Esta es la respuesta al evento *OnClick* del botón de búsqueda:

```
procedure TwndVentas.bnBuscarClick(Sender: TObject);
begin
    Datos.Close;
    Datos.ParamCheck := True;
    Datos.SQL.Clear;
    Datos.SQL.Add('select * from orders');
    Datos.SQL.Add('where saledate between :f1 and :f2');
    Datos.ParamByName('f1').AsDate :=
        EncodeDate(StrToInt(cbYears.Text), 1, 1);
    Datos.ParamByName('f2').AsDate :=
        EncodeDate(StrToInt(cbYears.Text), 12, 31);
    Datos.Open;
    CargarTabla;
end;
```

Dentro de la sentencia utilizamos parámetros del mismo modo en que lo haríamos en tiempo de diseño. Antes, nos hemos asegurado que la propiedad *ParamCheck* valga *True*, para que DB Express extraiga automáticamente la lista de parámetros de la instrucción; es cierto que *True* es el valor por omisión de la propiedad, pero no está de más ser cautos. He utilizado el método “largo” de modificación de la propiedad *SQL*, sólo por variar un poco.

Una vez que la sentencia se encuentra dentro del componente, ya podemos acceder a los parámetros como si los hubiéramos configurado en tiempo de diseño. Hay que tener un poco de cuidado llegados a este punto: los parámetros todavía no tienen un tipo asignado. Por eso, al escoger la propiedad *AsDate* para la asignación del valor, estamos a la vez asignando *f1Date* en la propiedad *DataType* del parámetro. Observe que la fecha que se asigna está en el formato binario interno de Delphi. Será DB

Express, en colaboración con InterBase, el encargado de traducir este valor binario al valor esperado por el servidor.

### ADVERTENCIA

Cuando estaba preparando el ejemplo inicialmente, utilicé *AsDateTime* para la asignación de valores al parámetro. El nuevo InterBase 6 (o quizás sea DB Express, no estoy seguro) es muy quisquilloso con sus nuevos tipos de datos, y la sentencia provocaba un error muy extraño al intentar activar la consulta: “*unassigned code*”, todo en minúsculas. Me costó trabajo adivinar de qué se trataba.

## Parámetros para la ejecución directa

En el capítulo anterior expliqué que el componente *TSQLConnection* nos ofrecía un método, llamado *ExecuteDirect*, para ejecutar directamente instrucciones SQL representadas en una cadena de caracteres. Si este método tiene un “apellido” tan largo es porque existe otro método similar, llamado *Execute* a secas. En aquel momento no mencioné su existencia porque su principal diferencia respecto a *ExecuteDirect* es que nos permite utilizar parámetros:

```
function TSQLConnection.Execute (
    const SQL: string;
    Params: TParams;
    ResultSet: Pointer = nil): Integer;
```

Hay otra novedad importante: se admite ahora la posibilidad de que la instrucción SQL devuelva un conjunto de datos como resultado. Si nos interesa manejar ese conjunto de datos, debemos utilizar el tercer parámetro del método. Pero nos concentraremos primero en el uso de parámetros en la instrucción.

Supongamos que quiero retocar la calificación de los clientes que han realizado una compra en determinado día:

```
update CLIENTES
set      BuenCliente = 1
where    IDCliente in (
    select IDCliente
    from   PEDIDOS
    where   FechaVenta = :fecha)
```

Si quisiéramos ejecutar esta instrucción, que contiene un parámetro *:fecha*, utilizando *Execute*, necesitaríamos algo similar a esto:

```
procedure TmodDatos.RecalificarClientes(Fecha: TDateTime);
const
    SInstruccion =
        'update CLIENTES set BuenCliente = 1 ' +
        'where IDCliente in (select IDCliente from PEDIDOS ' +
        'where FechaVenta = :fecha)';
var
    Parametros: TParams;
begin
    Parametros := TParams.Create;
```

```

    try
        Parametros.CreateParam(ftSQLTimeStamp, 'fecha', ptInput);
        Parametros[0].AsSQLTimeStamp := DateTimeToSQLTimeStamp(Fecha);
        SQLConnection1.Execute(SInstruccion, Parametros);
    finally
        Parametros.Free;
    end;
end;

```

Tenemos que crear, como puede ver, una colección de parámetros temporal y añadirle a la misma las definiciones de parámetros que existan en la instrucción. Esta vez, DB Express no nos echa una mano para identificar los parámetros y construir la lista. Por último, hay que asignarle valores a cada parámetro antes de pasar la colección al método *Execute*.

### MUCHO CUIDADO

Dice la documentación que la correspondencia entre los parámetros de la instrucción (quiero decir, dentro de la cadena de caracteres) y los elementos de la colección *TParams* se establece por su orden, no por el nombre que les hayamos dado. Es decir, que si la instrucción menciona dos parámetros, primero *:fecha* y luego *:valor*, tenemos que crear los parámetros en la colección utilizando exactamente ese mismo orden.

## Relaciones maestro/detalles en DB Express

En el capítulo 21 vimos por vez primera qué es una relación maestro/detalles, y aprendimos a configurarlas utilizando conjuntos de datos clientes. Ahora veremos cómo se realiza esta configuración cuando los conjuntos de datos involucrados pertenecen a DB Express. Comenzaremos asumiendo que el conjunto de datos de detalles pertenece a la clase *TSQLTable*, y utilizaré la siguiente base de datos de InterBase:

*c:\Archivos de programa\Archivos comunes\Borland Shared\Data\MastSql.gdb*

En ella se encuentran dos tablas, *CUSTOMER* con datos de clientes, y *ORDERS* con registros de pedidos. En la segunda encontramos la siguiente relación de integridad referencial:

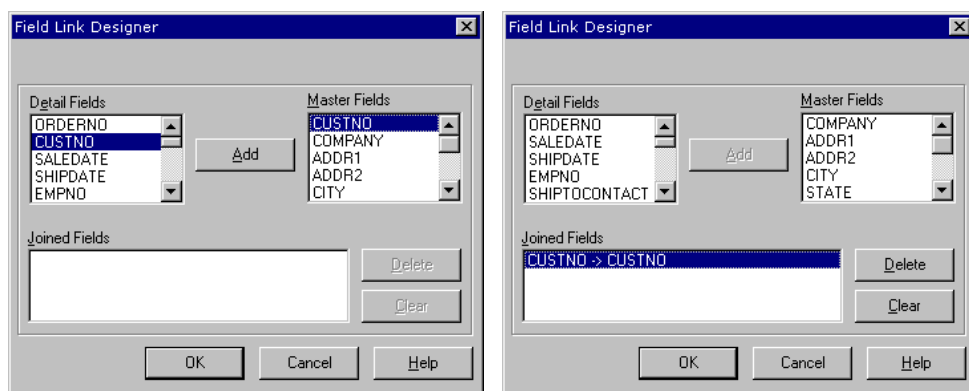
```

create table ORDERS (
    /* ... */
    foreign key (CustNo) references CUSTOMER(CustNo)
);

```

Primero traeremos, sobre un formulario vacío, un *TSQLConnection* y lo configuraremos para que se conecte a la base de datos citada. Como se trata de tablas pequeñas, traeremos un componente de tipo *TSQLTable* y lo asociaremos a la conexión anterior. Debemos cambiar su nombre a *tbClientes*, y asignar *CUSTOMER* en su propiedad *TableName*. Para terminar con esta parte, traigamos un *TDataSource*, llamémoslo *dsClientes*, y asignemos en su propiedad *DataSet* la referencia al conjunto de datos anterior.

Ahora es cuando aparecen las novedades: traiga otra *TSQLTable* y asígnele la misma conexión SQL. Cambie su nombre a *tbPedidos* y asíciela a la tabla *ORDERS*. Si en este preciso momento activásemos *tbPedidos* obtendríamos todos los registros existentes de pedidos. Pero vamos a seguir modificando propiedades. Localice la propiedad *MasterSource* de la tabla de pedidos y asigne en ella *dsClientes*, la fuente de datos asociada a la tabla de clientes. ¿Para qué esta asociación? Ya he mencionado, en capítulos anteriores, que una fuente de datos emite notificaciones subterráneas a los componentes que se enganchan a ella. Cada vez que cambie la posición de la fila activa de *tbClientes*, *dsClientes* se encargará de hacerlo saber a *tbPedidos*, para que reevalúe el conjunto de filas que debe mostrar.



Pero con *MasterSource* solamente hemos indicado que el contenido de la tabla de pedidos depende de la fila activa en la tabla de clientes. A continuación debemos modificar la propiedad *MasterFields*, también en la tabla de pedidos. Las imágenes anteriores muestran el cuadro de diálogo que aparece al hacer doble clic sobre el valor de dicha propiedad.

La lista de campos de la izquierda corresponde al esquema relacional de los pedidos. La otra lista, claro está, contiene los campos de un cliente. Primero seleccionamos en ambas listas los campos que las enlazan, de acuerdo a la relación de integridad referencial que hemos visto. Podemos entonces pulsar sobre el botón *Add* para añadir el par de campos a la lista inferior. En nuestro ejemplo, la relación de integridad referencial era simple, porque incluía una sola columna en ambas partes; en caso de relaciones con más columnas, repetiríamos el paso anterior tantas veces como fuese necesario.

## Ordenación de la tabla dependiente

Al cerrar el cuadro de diálogo, notará que se han modificado dos propiedades al mismo tiempo en *tbPedidos*:

Propiedad	Valor
<i>MasterFields</i>	<i>CUSTNO</i>

Propiedad	Valor
<i>IndexFieldNames</i>	<i>CUSTNO</i>

Es difícil verlo en nuestro ejemplo, porque las columnas elegidas en el maestro y en el detalle se llaman igual, pero en *IndexFieldNames* se han asignado los nombres de campos elegidos en la lista *Detail Fields* del editor de la propiedad *MasterFields*.

*IndexFieldNames* es una especie de reflejo de la propiedad *SortFieldNames* de la clase *TCustomSQLDataSet*, aunque no un reflejo inmediato, como veremos enseguida. Al igual que sucede con *SortFieldNames*, se utiliza para indicar los campos por los que queremos ordenar la consulta que el componente envía a la base de datos. Pero esta vez sí se admiten los puntos y comas como separadores de campos, además de las comas, y seguimos teniendo la posibilidad de utilizar los modificadores **asc** y **desc**.

Ahora usted me pedirá que justifique la asignación en *IndexFieldNames*... y me veré obligado a reconocer que se trata de una antigua, noble e inútil tradición heredada de la época en que nadie sentía vergüenza de decir que programaba en Paradox.

Las tablas de Paradox y dBase se implementan mediante un eficiente sistema de acceso directo al fichero. En compensación, existen algunas restricciones importantes; una de ellas es que, si queremos utilizar una tabla como conjunto de datos dependiente en una relación maestro/detalles, debemos ordenarla antes por los campos que participan en la relación. Esto no es necesario en absoluto para una base de datos cliente/servidor, como veremos un poco más adelante, pero se ha mantenido en Delphi por compatibilidad con las bases de datos de escritorio.

¿Qué efecto tiene entonces esa ordenación superflua? En general, ninguno visible. En el ejemplo de clientes y pedidos, las filas de pedidos se ordenan por la columna *CustNo*... pero a la vez, solamente se muestran las filas que tienen el mismo valor en esa columna. Podemos entonces retocar manualmente el criterio de ordenación a nuestro antojo, añadiendo otras columnas o quitando incluso las columnas originales. Por ejemplo:

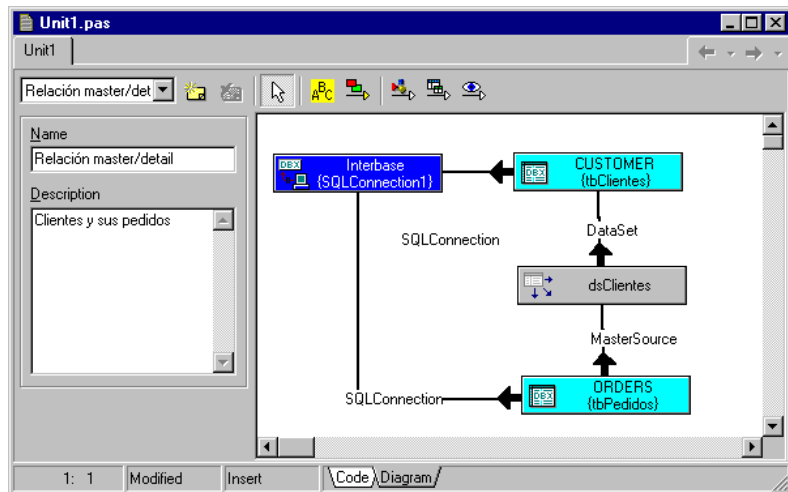
```
// Para hacer retoques, las tablas deben estar cerradas
tbPedidos.Close;
tbClientes.Close;
// Asignación explícita de propiedades
dsClientes.DataSet := tbClientes;
tbPedidos.MasterSource := dsClientes;
tbPedidos.MasterFields := 'CustNo';
tbPedidos.IndexFieldNames := 'CustNo;OrderNo';
// Apertura de las tablas
tbClientes.Open;
tbPedidos.Open;
```

Hay que tener precaución solamente de que la consulta resultante generada por la tabla dependiente se pueda evaluar eficientemente en el servidor.

## Diagramas

Antes de seguir con nuestro ejemplo, mostraré un interesante recurso de programación introducido en Delphi 5. Vaya al Editor de Código para ver el código fuente de la unidad principal del proyecto. Observe la parte inferior de la ventana: hay dos pestañas más bien a la derecha. La pestaña activa se llama *Code*; active la otra pestaña, *Diagram*.

La vista que tiene ante usted le permitirá crear algunos sencillos diagramas para describir las conexiones entre los componentes que sitúe en ese formulario o módulo de datos. No se trata de diagramas con algún tipo de formalismo asociado, sino una representación informal ideada por Borland. Para cada módulo o formulario podemos definir varios diagramas; en Delphi 5 solamente se permitía uno de ellos.

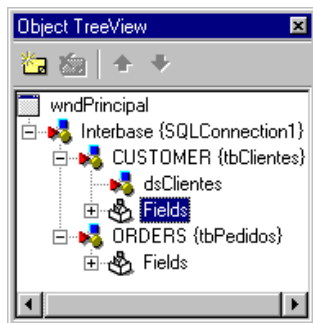


Para añadir elementos al diagrama, debe arrastrarlos desde el árbol de objetos que se encuentra sobre el Inspector de Objetos. Arrastre, por ejemplo, la tabla de clientes, la tabla de pedidos y el componente *dsClientes*. Obtendrá algo parecido a la imagen de la página anterior (he retocado un poco los colores). Como comprobará, las relaciones entre los componentes se añaden de forma automática una vez que están presentes los extremos necesarios.

En circunstancias reales, no habría incluido el componente de conexión en el diagrama; a no ser que estemos trabajando simultáneamente con varias conexiones, se sobreentiende que todos los conjuntos de datos deben estar asociados a la única conexión existente.

Pero podíamos haber procedido a la inversa, al menos en teoría. Comenzaríamos trayendo al formulario el componente de conexión y las dos tablas. Tendríamos, además, que asignar *TableName* en ambas tablas. Entonces pulsaríamos sobre el penúltimo botón de la barra sobre el diagrama, *Master/detail connector*, y trazáramos un

segmento partiendo de la tabla de clientes y terminando en la de pedidos. No intente hacerlo, porque no va a funcionar. Si en vez de DB Express estuviéramos trabajando con componentes del BDE, al añadir el conector se añadiría también el *TDataSource* requerido por la relación, y se nos mostraría el cuadro de diálogo asociado a *Master-Fields* para que configurásemos la relación.



## Recorrido maestro/detalles

Todavía tenemos que terminar el ejemplo, haciendo algo “útil” con los conjuntos de datos que acabamos de configurar, pero lo único que se me ocurre es crear un fichero de texto para “exportar” los datos de todos los clientes, seguidos de todos los registros de pedidos que ha realizado.

¿Es una tontería? Sí, casi siempre lo es. Ya hemos visto cómo crear un fichero XML con la estructura que nos venga en gana utilizando conjuntos de datos clientes. En el capítulo sobre proveedores estudiaremos cómo alimentar de forma automática un conjunto de datos cliente con las filas de un componente DB Express. No obstante, no todas las aplicaciones aceptan todavía ficheros XML para importar datos. Además, aprovecharé para repasar unas cuantas técnicas de manejo de cadenas de caracteres.

Las reglas que impondré a nuestro procedimiento de exportación son:

- 1 Los datos de cada registro se ubicarán en una fila independiente. La secuencia de cambio de línea será la de MS-DOS: el carácter 13 seguido del carácter 10. Muchas “especificaciones” que he sufrido son ambiguas en este punto.
- 2 Cada columna ocupará un ancho fijo, determinado por el valor de la propiedad *DisplayWidth* del campo correspondiente. La sustitución por un formato con delimitadores sería igual de sencilla.
- 3 Se incluirán todos los campos de cada tabla. Se trata, también, de una inocente simplificación. En realidad, nos sobraría el código de cliente en la fila de pedidos, y deberíamos tener cuidado con los campos blob.
- 4 Tras un registro de cliente, seguirán todos los registros de pedidos asociados. No voy a exigir cabeceras, ni pies de grupos con totales.



- 5 Los registros de pedidos y clientes se diferenciarán por el primer carácter de la línea; *C* para clientes, *P* para pedidos. Es un sistema muy frecuente.

Order ID	Customer Name	Address	Date 1	Date 2
C1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy	Suite 1	
P1023	1221	01/07/88	02/07/88	
P1076	1221	16/12/94	26/04/89	
P1123	1221	24/08/93	24/08/93	
P1169	1221	06/07/94	06/07/94	
P1176	1221	26/07/94	26/07/94	
P1269	1221	16/12/94	16/12/94	
C1231	Unisco	PO Box Z-547		
P1060	1231	28/02/89	01/03/89	
P1073	1231	15/04/89	16/04/89	
P1102	1231	06/06/92	06/06/92	
P1160	1231	01/06/94	01/06/94	
P1173	1231	16/07/94	16/07/94	
P1178	1231	02/08/94	02/08/94	
P1202	1231	06/10/94	06/10/94	
P1278	1231	23/12/94	23/12/94	
P1302	1231	16/01/95	16/01/95	

Los prototipos de los métodos básicos que definiremos se muestran a continuación. Observe el uso de la cláusula **overload**:

```
type
  TwndPrincipal = class(TForm)
    // ...
  private
    procedure Exportar(const AFileName: string); overload;
    procedure Exportar(S: TStream); overload;
    procedure ExportarCliente(S: TStream);
    procedure ExportarPedido(S: TStream);
    // ...
  end;
```

La versión de *Exportar* que utilizaremos con toda seguridad es ésta:

```
procedure TwndPrincipal.Exportar(const AFileName: string);
var
  F: TFileStream;
begin
  F := TFileStream.Create(AFileName, fmCreate);
  try
    Exportar(F);
  finally
    F.Free;
  end;
end;
```

Es decir, creamos un objeto *TFileStream*, que es un caso particular de *TStream*, y nos encargamos de destruirlo al final del algoritmo. En el intermedio, nos lavamos las extremidades superiores y le pasamos la patata caliente a la versión de *Exportar* que trabaja precisamente con parámetros *TStream*:

```
procedure TwndPrincipal.Exportar(S: TStream);
begin
  tbClientes.Open;
```

```

try
  tbPedidos.Open;
  try
    while not tbClientes.Eof do
    begin
      ExportarCliente(S);
      while not tbPedidos.Eof do
      begin
        ExportarPedido(S);
        tbPedidos.Next;
      end;
      tbClientes.Next;
    end;
  finally
    tbPedidos.Close;
  end;
  finally
    tbClientes.Close;
  end;
end;

```

### SUGERENCIAS

Si quisiéramos que el procedimiento anterior fuese a prueba de tontos, añadiríamos llamadas al método *Close* de ambos conjuntos de datos, al comienzo del algoritmo. Así garantizaríamos que siempre partimos del primer registro del cliente y de sus pedidos.

Esta decisión tiene mucho que ver con el hecho de que los conjuntos de datos en DB Express son unidireccionales. En otro caso, quizás sería mejor abrir y cerrar sólo si las tablas estaban cerradas, y llamar siempre a *First*, antes de lanzarnos al bucle.

Ahora puedo explicar la conveniencia de haber sobrecargado el método *Exportar*:

- 1 Tenemos una versión muy general de *Exportar* que acepta cualquier descendiente de la clase *TStream* como destino. Podría tratarse de un fichero de texto con un *buffer* de escritura (algo que no implementa nuestro sencillo *TFileStream*), pero podría ser algo más esotérico, como una clase que permitiese escribir en un *named pipe* de Windows, o en un *zócalo* (o *socket*) de TCP/IP.
- 2 Como efecto secundario, pero útil, evitamos que la “escalera” de bloques anidados se haga muy profunda, como sucedería si todo el código se hubiera incluido en un único método.

## Exportar una fila

Llegados a este punto, la implementación de *ExportarCliente* y *ExportarPedido* no tiene mayor importancia, que no sea la de terminar el ejercicio. Voy a resolver el asunto con un método genérico, cuyo prototipo no he mostrado antes:

```

procedure TwndPrincipal.ExportarFila(S: TStream; D: TDataSet;
  const Prefijo: string);
var
  I, Ancho, Posicion: Integer;
  Linea, Campo: string;

```

```

begin
    // Calcular ancho total, incluyendo prefijo y cambio de línea
    Ancho := Length(Prefijo) + 2;
    for I := 0 to D.FieldCount - 1 do
        Inc(Ancho, D.Fields[I].DisplayWidth);
    // Asignar espacio en memoria
    SetLength(Linea, Ancho);
    // Inicializar con espacios
    FillChar(Linea[1], Ancho, ' ');
    // Cambio de línea MSDOS al final
    Linea[Ancho - 1] := #13;
    Linea[Ancho] := #10;
    // Mover el prefijo, si no es vacío
    Posicion := 1;
    if Prefijo <> '' then
        begin
            Move(Prefijo[1], Linea[1], Length(Prefijo));
            Inc(Posicion, Length(Prefijo));
        end;
    for I := 0 to D.FieldCount - 1 do
        with D.Fields[I] do
            begin
                Campo := Copy(Text, 1, DisplayWidth);
                if Campo <> '' then
                    Move(Campo[1], Linea[Posicion], Length(Campo));
                Inc(Posicion, DisplayWidth);
            end;
        // Copiar la línea en el stream
        S.Write(Linea[1], Ancho)
    end;
end;

```

Reconozco que no es habitual que en un libro se incluya código tan sucio como el anterior; es que todo lo real es más o menos sucio. El fantasma que he exorcizado no es la lentitud, que también se combate, sino la fragmentación de memoria. La tentación de muchos sería inicializar una cadena vacía, e ir concatenando en su final la representación en modo texto de cada columna. Pero esa técnica provocaría innumerables llamadas al gestor de memoria dinámica. En este algoritmo, con las pocas filas que existen en la base de datos de ejemplo, casi no se nota la diferencia, pero con un número mayor de filas, la memoria quedaría hecha un asco al terminar el procedimiento.

## Consultas enlazadas

A pesar de que he presentado primero las relaciones maestro/detalles sobre tablas, la mejor técnica para sistemas cliente/servidor requiere que el conjunto de datos dependiente sea de tipo consulta. Por tres razones:

- 1 El mecanismo de enlace se hace muy evidente, porque somos nosotros los que especificamos la instrucción **select** del conjunto de datos de detalles. Eso evita algunos errores que pueden afectar al correcto funcionamiento de la relación y a su eficiencia.
- 2 Hay más libertad para configurar la relación.
- 3 Las consultas se adaptan mejor a la metodología de trabajo cliente/servidor.

Veamos cómo se configuran dos consultas para obtener los datos de *un* cliente, más todos sus pedidos. Voy a asumir que ya tenemos configurada la consulta maestra, en un componente al que llamaremos *qrCliente*; utilizo el singular para el nombre para indicar que la consulta debe devolver una sola fila a lo sumo.

A continuación, igual que en la relación clásica, necesitamos asociar un *TDataSource* a esa consulta; al nuevo componente lo llamaremos *dsCliente*, para que haga juego con el conjunto de datos. Ya podemos concentrarnos en la nueva consulta, a la que le daremos el nombre de *qrPedidos*. Después de asociar la consulta al componente de conexión, debe teclear la siguiente instrucción dentro de la propiedad *SQL*:

```
select *
from   ORDERS o inner join EMPLOYEE e
      on (o.EmpNo = e.EmpNo)
where  o.CustNo = :CustNo
order by OrderNo asc
```

La sentencia mezcla datos de dos tablas: la de pedidos, que es la que principalmente nos interesa, pero además se incluye la tabla de empleados, para recuperar los datos del empleado que ha recibido el pedido. Lo más importante: hay un parámetro, *:CustNo*, pero esta vez no necesitaremos asignarle un tipo explícitamente. Lo que haremos es buscar la propiedad *MasterSource* de *qrPedidos*, y asignarle una referencia al componente *dsCliente*.

Ahora veamos cómo funciona esta configuración. Cada vez que se selecciona una fila en la consulta maestra, la fuente de datos (*data source*) detecta la incidencia y dispara toda una serie de eventos internos, además del evento publicado *OnDataChange*. Digamos de paso que este aviso se produce también cuando se abre la consulta por primera vez; recuerde que en este ejemplo la consulta maestra contiene una o ninguna fila.

La consulta de detalles, al estar conectada a la fuente de datos, está muy pendiente de los eventos internos mencionados. Cuando detecta un cambio de fila, en primer lugar, se desactiva. Y a continuación tiene lugar una asignación automática de valores sobre parámetros. ¿Recuerda que *qrPedidos* tenía un único parámetro llamado *:CustNo*? La consulta busca entonces una columna en el conjunto de datos indirectamente asociado a su propiedad *MasterSource* que tenga precisamente el mismo nombre que el parámetro. Por supuesto, hay un campo *CustNo* en la consulta de clientes, y el valor actual de esa columna se copia en el parámetro de la consulta de detalles. Finalmente, se vuelve a activar la consulta dependiente, que como es lógico esperar, mostrará solamente los pedidos correspondientes al cliente activo.

Cuando estudiemos el proceso de grabación de datos con la ayuda de los conjuntos de datos clientes, veremos que es muy fácil lograr que la consulta de detalles que hemos mostrado sea actualizable, a pesar de haber utilizado un encuentro natural en su formulación.





## El Motor de Datos de Borland

**D**ELPHI ES UN LENGUAJE DE PROPÓSITO GENERAL; nunca me cansaré de repetirlo. Hubo un tiempo en que se puso de moda clasificar los lenguajes por “generaciones”, y mientras más alto el número asignado, supuestamente mejor era el lenguaje. Pascal y C desfilaban en el pelotón de la tercera generación: los lenguajes de “propósito general”, que venían a ser algo así como el lenguaje ensamblador de los de “cuarta generación”. Y a esta categoría “superior” pertenecían los lenguajes de programación para bases de datos que comenzaban a proliferar en aquel entonces.

El punto débil de esos lenguajes con esteroides era, precisamente, su orientación a un sistema de base de datos muy concreto. Cuando la base de datos pasaba a mejor vida, el lenguaje correspondiente, como una viuda hindú de leyenda, debía arrojarse también a la pira. Delphi, en cambio, ha sobrevivido a la decadencia del BDE, que en un primer momento fue el eje sobre el que se construyó su sistema de acceso a datos.

### Qué es, y cómo funciona

La principal diferencia entre BDE y DB Express es que uno de los objetivos de diseño del primero es hacernos creer, cuando trabajamos con bases de datos SQL, que estamos accediendo a una tabla de Paradox o dBase. Si utilizamos un servidor que no soporta cursores bidireccionales, el Motor de Datos se encargará de implementarlos en el lado cliente. Y a la inversa: para acercar las bases de datos de escritorio a las orientadas a conjuntos, BDE ofrece un intérprete local para el lenguaje SQL, que aprovechan Paradox, dBase y FoxPro.

Físicamente, el Motor de Datos de Borland consiste en una serie de DLLs. Parte de ellas forman lo que se denomina el núcleo del motor. Alrededor de ese núcleo, se implementan los controladores para los siguientes formatos de bases de datos de escritorio: dBase, Paradox, FoxPro y Access; es necesario tener el motor Jet instalado para trabajar con Access, por lo que no es una buena opción para trabajar con este sistema. El BDE puede también trabajar, aunque con las lógicas limitaciones, con tablas almacenadas en ficheros de texto.

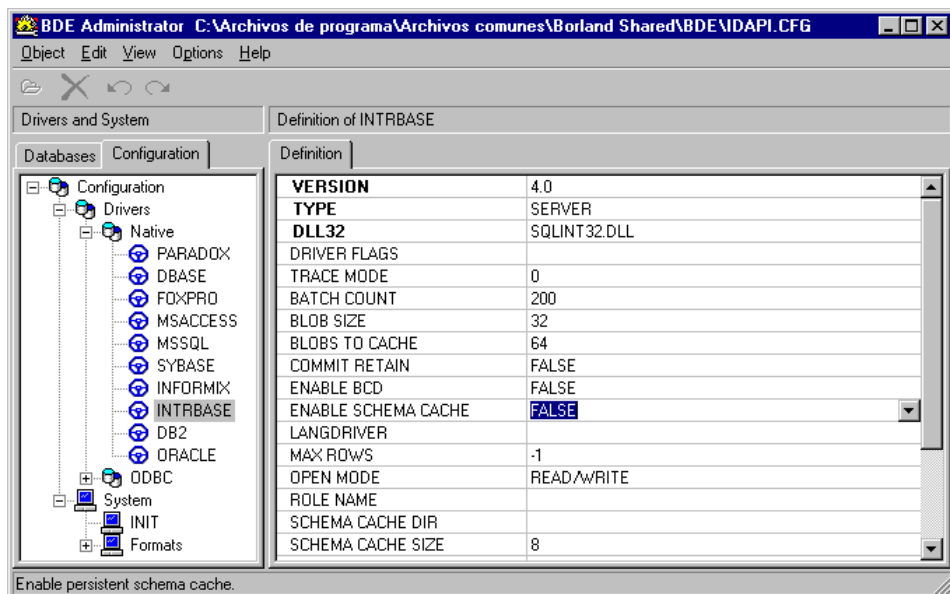
El acceso a bases de datos SQL se logra mediante DLLs adicionales, conocidas como *SQL Links (enlaces SQL)*. Actualmente existen SQL Links para los siguientes formatos: InterBase, Oracle, DB2, Sybase, Informix y SQL Server. Es mejor no utilizar el acceso a SQL Server, porque está basado en DB Library, una interfaz obsoleta. Tampoco es recomendable acceder a InterBase mediante el BDE, pero esta vez se debe a la existencia de una alternativa mejor: DB Express.

Finalmente, BDE permite también conexiones a controladores ODBC, cuando algún formato no es soportado directamente en el Motor. No obstante, debido a las numerosas capas de software que se interponen entre la aplicación y la base de datos, solamente debemos recurrir a esta opción en casos desesperados.

Para distribuir una aplicación de bases de datos escrita con los componentes del BDE, necesitamos distribuir también el Motor de Datos. La instalación y configuración de todos los ficheros necesarios es bastante complicada. Afortunadamente, la versión reducida de Install Shield Express que se distribuye con Delphi tiene opciones para integrar la instalación del BDE con la de nuestra aplicación. Y muchas otros generadores de instalaciones, como Wise, soportan esta posibilidad.

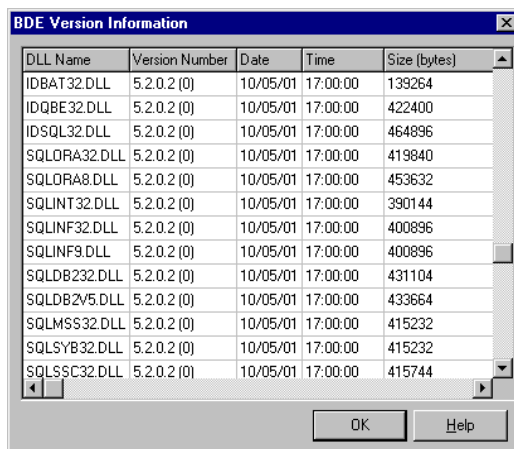
## El Administrador del Motor de Datos

La configuración del BDE se realiza mediante el programa *BDE Administrator*, que se puede ejecutar por medio del acceso directo existente en el grupo de programas de Delphi. Incluso cuando se instala el *runtime* del BDE en una máquina “limpia”, se copia este programa para poder ajustar los parámetros de funcionamiento del Motor.





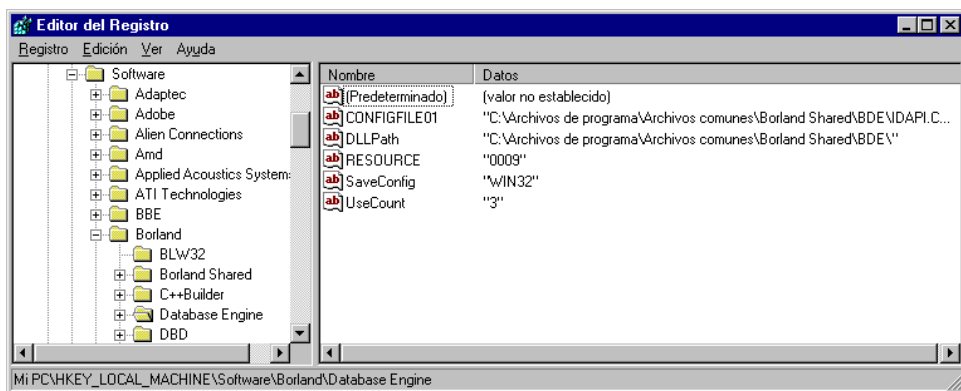
La ventana principal del Administrador está dividida en dos páginas, *Databases* y *Configuration*; las opciones disponibles en el menú dependen de la página activa. Seleccione, por ejemplo, la página *Configuration*, y busque dentro del menú *Object* el comando *Version Information*. La ventana que aparece contiene los números de versión de cada una de las DLLs del BDE. Estos son los números que importan: según se aprecia en la imagen, tengo instalada la versión 5.2.0.2, pero si el diálogo *About* me miente diciendo que es la 5.01. Claro, el nota de *copyright* hace referencia al 1998...



DLL Name	Version Number	Date	Time	Size (bytes)
IDBAT32.DLL	5.2.0.2 (0)	10/05/01	17:00:00	139264
IDQBE32.DLL	5.2.0.2 (0)	10/05/01	17:00:00	422400
IDSQL32.DLL	5.2.0.2 (0)	10/05/01	17:00:00	464896
SQLORA32.DLL	5.2.0.2 (0)	10/05/01	17:00:00	419840
SQLORA8.DLL	5.2.0.2 (0)	10/05/01	17:00:00	453632
SQLENT32.DLL	5.2.0.2 (0)	10/05/01	17:00:00	390144
SQLENT32.DLL	5.2.0.2 (0)	10/05/01	17:00:00	400896
SQLENT32.DLL	5.2.0.2 (0)	10/05/01	17:00:00	400896
SQLDB232.DLL	5.2.0.2 (0)	10/05/01	17:00:00	431104
SQLDB2V5.DLL	5.2.0.2 (0)	10/05/01	17:00:00	433664
SQLMSS32.DLL	5.2.0.2 (0)	10/05/01	17:00:00	415232
SQLSYB32.DLL	5.2.0.2 (0)	10/05/01	17:00:00	415232
SQLSSC32.DLL	5.2.0.2 (0)	10/05/01	17:00:00	415744

¿Dónde se almacena la información de configuración del BDE? Resulta que se usan dos sitios diferentes: la parte principal se guarda en el registro de Windows, mientras que la información de los alias y un par de parámetros especiales se guardan en el fichero *idapi32.cfg*, cuya ubicación se define en el propio registro. La clave a partir de la cual se encuentran los datos de configuración del BDE es la siguiente:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Borland\Database Engine]
```



Por ejemplo, la cadena *UseCount* indica cuántos productos instalados están utilizando el BDE, mientras que *ConfigFile01* especifica el fichero de configuración a utilizar, y *DLLPath*, el directorio donde están los demás ficheros del BDE.

## El concepto de alias

Para “aplanar” las diferencias entre tantos formatos diferentes de bases de datos y métodos de acceso, BDE introduce los *alias*. Un alias es, sencillamente, un nombre simbólico para referirnos a un base de datos. Cuando un programa que utiliza el BDE quiere, por ejemplo, abrir una tabla, sólo tiene que especificar un alias y la tabla que quiere abrir. Entonces el Motor de Datos examina su lista de alias y puede saber en qué formato se encuentra la base de datos, y cuál es su ubicación.

Existen dos tipos diferentes de alias: los alias *persistentes* y los alias *locales*, o de *sesión*. Los alias persistentes se crean por lo general con el Administrador del BDE, y pueden utilizarse por cualquier aplicación que se ejecute en la misma máquina. Los alias locales son creados mediante llamadas al BDE realizadas desde una aplicación, y son visibles solamente dentro de la misma. La VCL facilita esta tarea mediante componentes de alto nivel, en concreto mediante la clase *TDatabase*.

Un alias ofrece, a la aplicación que lo utiliza, independencia con respecto al formato de los datos y su ubicación. Esto vale sobre todo para los alias persistentes, creados con el BDE. Puedo estar desarrollando una aplicación en casa, que trabaje con tablas del alias *datos*. En mi ordenador, este alias está basado en el controlador de Paradox, y las tablas encontrarse en un determinado directorio de mi disco local. El destino final de la aplicación, en cambio, puede ser un ordenador en que el alias *datos* haya sido definido como una base de datos de Oracle, situada en tal servidor y con tal protocolo de conexión. A nivel de aplicación no necesitaremos cambio alguno para que se ejecute en las nuevas condiciones; al menos, en teoría.

Por último, existen los llamados alias *virtuales*, que corresponden a los nombres de fuentes de datos de ODBC (*data source names*). Gracias a ellos, una aplicación puede utilizar directamente el nombre de un DSN como si fuera un alias nativo del BDE.

## Parámetros del sistema

Los parámetros globales del Motor de Datos se cambian en la página *Configuration*, en el nodo *System*. Este nodo tiene a su vez dos nodos hijos, *INIT* y *Formats*, para los parámetros de funcionamiento y los formatos de visualización por omisión. Puede olvidarse de *Formats*, que es una reliquia de eones pasados.

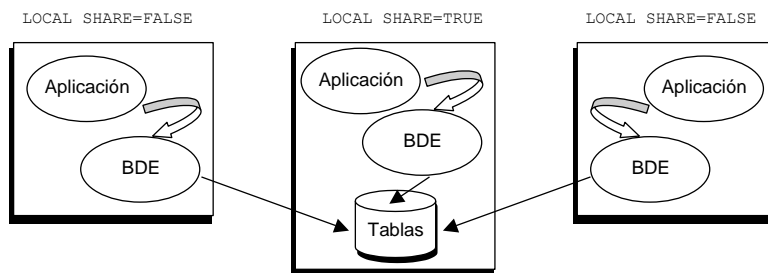
La mayor parte de los parámetros de sistema tienen que ver con el uso de memoria y otros recursos del ordenador. Estos parámetros son:

Parámetro	Explicación
<i>MAXFILEHANDLES</i>	Máximo de <i>handles</i> de ficheros admitidos.
<i>MINBUFSIZE</i>	Tamaño mínimo del <i>buffer</i> de datos, en KB.
<i>MAXBUFSIZE</i>	Tamaño máximo del <i>buffer</i> de datos, en KB (múltiplo de 128).

Parámetro	Explicación
<i>MEMSIZE</i>	Tamaño máximo, en MB, de la memoria consumida por BDE.
<i>LOW MEMORY USAGE LIMIT</i>	Memoria por debajo del primer MB consumida por BDE.
<i>SHAREDMEMSIZE</i>	Tamaño máximo de la memoria para recursos compartidos.
<i>SHAREDMEMLOCATION</i>	Posición en memoria de la zona de recursos compartidos

El área de recursos compartidos se utiliza para que distintas instancias del BDE, dentro de un mismo ordenador, pueden compartir datos entre sí. Las bibliotecas de enlace dinámico de 32 bits tienen un segmento de datos propio para cada instancia, por lo cual tienen que recurrir a ficheros asignados en memoria (*memory mapped files*), un recurso de Windows 95 y NT, para lograr la comunicación entre procesos.

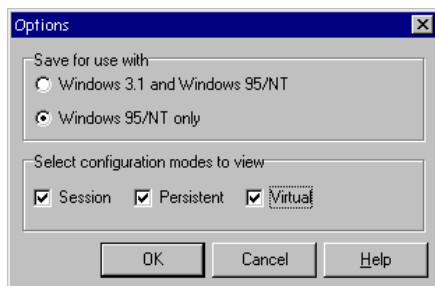
Cuando se va a trabajar con dBase y Paradox es importante configurar correctamente el parámetro *LOCAL SHARE*. Para acelerar las operaciones de bloqueo, BDE normalmente utiliza una estructura en memoria para mantener la lista de bloqueos impuestos en las tablas pertenecientes a los discos locales; esta técnica es más eficiente que depender de los servicios de bloqueo del sistema operativo. Pero solamente vale para aplicaciones que se ejecutan desde un solo puesto. La suposición necesaria para que este esquema funcione es que las tablas sean modificadas siempre por la misma copia del BDE. Si tenemos cinco aplicaciones ejecutándose en un mismo ordenador, todas utilizan la misma imagen en memoria del BDE, y todas utilizan la misma tabla de bloqueos.



Si, por el contrario, las tablas se encuentran en un disco remoto, BDE utiliza los servicios de bloqueo del sistema operativo para garantizar la integridad de los datos. En este caso, el Motor de Datos no tiene acceso a las estructuras internas de otras copias del Motor que se ejecutan en otros nodos de la red. El problema surge cuando dos máquinas, llamémoslas *A* y *B*, acceden a las mismas tablas, que supondremos situadas en *B*. El BDE que se ejecuta en *A* utilizará el sistema operativo para los bloqueos, pero *B* utilizará su propia tabla interna de bloqueos, pues las tablas se encuentran en su disco local. En consecuencia, las actualizaciones concurrentes sobre las tablas destruirán la integridad de las mismas. La solución es cambiar *LOCAL*

*SHARE* a *TRUE* en el ordenador *B*. De cualquier manera, es preferible que en una red punto a punto que ejecute aplicaciones para Paradox y dBase concurrentemente, el servidor de ficheros esté dedicado exclusivamente a este servicio, y no ejecute aplicaciones en su espacio de memoria; la única razón de peso contra esta política es un presupuesto bajo, que nos obligue a aprovechar también este ordenador.

El otro parámetro importante de la sección *INIT* es *AUTO ODBC*. En versiones anteriores del BDE, este parámetro se utilizaba para crear automáticamente alias en el Motor de Datos que correspondieran a las fuentes de datos ODBC registradas en el ordenador; la operación anterior se efectuaba cada vez que se inicializaba el Motor de Datos. En la versión actual, no se recomienda utilizar esta opción, pues el nuevo modo de configuración virtual de fuentes ODBC la hace innecesaria. Para activar la visualización de alias virtuales, seleccione el comando de menú *Object|Options*, y marque la opción correspondiente en el cuadro de diálogo que aparece a continuación:



## Parámetros de los controladores para BD locales

Para configurar controladores de bases de datos locales y SQL, debemos buscar estos controladores en la página *Configuration*, bajo el nodo *Configuration/Drivers/ Native*. Los parámetros para los formatos de bases de datos locales son muy sencillos; comencemos por Paradox.

Quizás el parámetro más importante de Paradox es el directorio del fichero de red: *NET DIR*. Esta variable es indispensable cuando se van a ejecutar aplicaciones con tablas Paradox en una red punto a punto, y debe contener el nombre de un directorio de la red compartido para acceso total. El nombre de este directorio debe escribirse en formato UNC, y debe ser *idéntico* en todos los ordenadores de la red; por ejemplo:

```
\\SERVIDOR\DirRed
```

He recalcado el adjetivo “idéntico” porque si el servidor de ficheros contiene una copia del BDE, podemos vernos tentados a configurar *NET DIR* en esa máquina utilizando como raíz de la ruta el nombre del disco local: *c:\DirRed*. Incluso en este ordenador debemos utilizar el nombre UNC. En las versiones de 16 bits del BDE, que no pueden hacer uso de esta notación, se admiten diferencias en la letra de unidad asociada a la conexión de red.

En el directorio de red de Paradox se crea dinámicamente el fichero *pdoxusr.net*, que contiene los datos de los usuarios conectados a las tablas. Como hay que realizar escrituras en este fichero, se explica la necesidad de dar acceso total al directorio compartido. Algunos administradores inexpertos, en redes Windows 95, utilizan el directorio raíz del servidor de ficheros para este propósito; es un error, porque así estamos permitiendo acceso total al resto del disco.

Cuando se cambia la ubicación del fichero de red de Paradox en una red en la que ya se ha estado ejecutando aplicaciones de bases de datos, pueden producirse problemas por referencias a este fichero almacenadas en ficheros de bloqueos, de extensión *lck*. Mi consejo es borrar todos los ficheros *net* y *lck* de la red antes de modificar el parámetro *NET DIR*.

Los otros dos parámetros importantes de Paradox son *FILL FACTOR* y *BLOCK SIZE*. El primero indica qué porcentaje de un bloque debe estar lleno antes de proceder a utilizar uno nuevo. *BLOCK SIZE* especifica el tamaño de cada bloque en bytes. Paradox permite hasta 65.536 bloques por tabla, por lo que con este parámetro estamos indicando también el tamaño máximo de las tablas. Si utilizamos el valor por omisión, 2.048 bytes, podremos tener tablas de hasta 128MB. Hay que notar que estos parámetros se aplican durante la creación de nuevas tablas; si una tabla existente tiene un tamaño de bloque diferente, el controlador puede utilizarla sin problema alguno.

La configuración de dBase es aún más sencilla. Los únicos parámetros dignos de mención son *MDX BLOCK SIZE* (tamaño de los bloques del índice) y *MEMO FILE BLOCK SIZE* (tamaño de los bloques de los memos). Recuerde que mientras mayor sea el tamaño de bloque de un índice, menor será su profundidad y mejores los tiempos de acceso. Hay que tener cuidado, sin embargo, pues también es más fácil desbordar el *buffer* en memoria, produciéndose más intercambios de páginas.

## Bloqueos oportunistas

Windows NT permite mejorar la concurrencia en los accesos directos a ficheros de red introduciendo los bloqueos oportunistas. Cuando un cliente de red intenta abrir un fichero situado en el servidor, Windows NT le asigna un bloqueo exclusivo sobre el fichero completo. De esta manera, el cliente puede trabajar eficientemente con copias locales en su caché, realizar escrituras diferidas, etc. La única dificultad consiste en que cuando otro usuario intenta acceder a este mismo fichero, hay que bajarle los humos al oportunista primer usuario, forzándolo a vaciar sus *buffers*.

Bien, resulta que esta maravilla de sistema funciona mal con casi todos los sistemas de bases de datos de escritorio, incluyendo a Paradox. El error se manifiesta cuando las tablas se encuentran en un Windows NT Server, y un usuario encuentra que una tabla completa está bloqueada, cuando en realidad solamente hay un registro bloqueado por otro usuario. Este error es bastante aleatorio: el servidor NT con el que

trabajo habitualmente contiene tablas de Paradox para pruebas con miles de registros, y hasta el momento no he tenido problemas. Pero ciertas personas que conozco han pillado este resfriado a la primera.

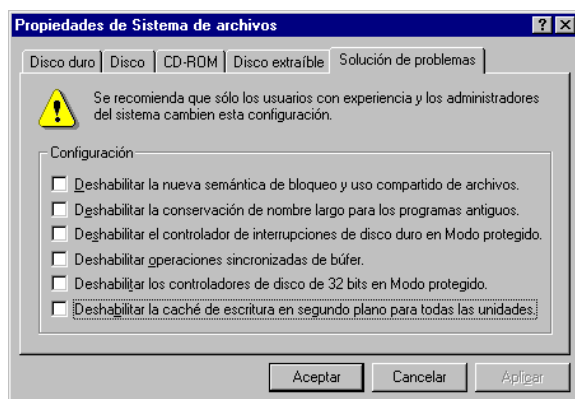
Para curarnos en salud, lo más sensato es desactivar los bloqueos oportunistas añadiendo una clave al registro de Windows NT Server. El camino a la clave es el siguiente:

```
[HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\
LanManServer\Parameters]
```

La clave a añadir es la siguiente:

```
EnableOplocks: (DWORD) 00000000
```

También es conveniente modificar las opciones por omisión de la caché de disco en los clientes de Paradox. Por ejemplo, Windows activa por omisión una caché de escritura en segundo plano. Por supuesto, un fallo de la alimentación o un programa colgado pueden afectar a la correcta grabación de las actualizaciones sobre una tabla. Esta opción puede desactivarse directamente desde la interfaz gráfica de Windows:



Pero si desea que su programa compruebe el estado de la opción e incluso lo modifique, puede mirar la siguiente clave del registro:

```
[HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\FileSystem]
DriveWriteBehind = 00 (DWORD)
```

Por último, muchos programadores recomiendan añadir la siguiente entrada en el registro de Windows 95/98:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\VXD\VREDIR]
DiscardCacheOnOpen = 01
```

Así, cada vez que se abre una tabla de Paradox se elimina cualquier versión obsoleta de la misma que haya podido quedar en la memoria caché del cliente de red.

## Parámetros comunes a los controladores SQL

A pesar de las diferencias entre los servidores SQL disponibles, en la configuración de los controladores correspondientes existen muchos parámetros comunes. La mayor parte de estos parámetros se repiten en la configuración del controlador y en la de los alias correspondientes. El objetivo de esta aparente redundancia es permitir especificar valores por omisión para los alias que se creen posteriormente. Existen, no obstante, parámetros que solamente aparecen en la configuración del controlador, y otros que valen sólo para los alias.

Todos los controladores SQL tienen los siguientes parámetros no modificables:

Parámetro	Significado
<i>VERSION</i>	Número interno de versión del controlador
<i>TYPE</i>	Siempre debe ser <i>SERVER</i>
<i>DLL</i>	DLL de acceso para 16 bits
<i>DLL32</i>	DLL de acceso para 32 bits

En realidad, *DLL* y *DLL32* se suministran para los casos especiales de Informix, Sybase y Oracle, que admiten dos interfaces diferentes con el servidor. Solamente en estos casos deben modificarse los parámetros mencionados.

La siguiente lista muestra los parámetros relacionados con la apertura de bases de datos:

Parámetro	Significado
<i>SERVER NAME</i>	Nombre del servidor
<i>DATABASE NAME</i>	Nombre de la base de datos
<i>USER NAME</i>	Nombre del usuario inicial
<i>OPEN MODE</i>	Modo de apertura: <i>READ/WRITE</i> ó <i>READ ONLY</i>
<i>DRIVER FLAGS</i>	Modifíquelo sólo según instrucciones de Borland
<i>TRACE MODE</i>	Indica a qué operaciones se les sigue la pista

No es necesario, ni frecuente, modificar los cuatro primeros parámetros a nivel del controlador, porque también están disponibles a nivel de conexión a una base de datos específica. En particular, la interpretación del nombre del servidor y del nombre de la base de datos varía de acuerdo al formato de bases de datos al que estemos accediendo. Por ejemplo, el controlador de InterBase utiliza *SERVER NAME*, pero no *DATABASE NAME*.

En los sistemas SQL, la información sobre las columnas de una tabla, sus tipos y los índices definidos sobre la misma se almacena también en tablas de catálogo, dentro de la propia base de datos. Si no hacemos nada para evitarlo, cada vez que nuestra aplicación abra una tabla por primera vez durante su ejecución, estos datos deberán viajar desde el servidor al cliente, haciéndole perder tiempo al usuario y ocupando el

ancho de banda de la red. Por fortuna, es posible mantener una caché en el cliente con estos datos, activando el parámetro lógico *ENABLE SCHEMA CACHE*. Si está activo, se utilizan los valores de los siguientes parámetros relacionados:

Parámetro	Significado
<i>SCHEMA CACHE DIR</i>	El directorio donde se copian los esquemas
<i>SCHEMA CACHE SIZE</i>	Cantidad de tablas cuyos esquemas se almacenan
<i>SCHEMA CACHE TIME</i>	Tiempo en segundos que se mantiene la caché

En el directorio especificado mediante *SCHEMA CACHE DIR* se crea un fichero de nombre *scache.ini*, que apunta a varios ficheros de extensión *scf*, que son los que contienen la información de esquema. Si no se ha indicado un directorio en el parámetro del BDE, se utiliza el directorio de la aplicación. Tenga en cuenta que si la opción está activa, y realizamos cambios en el tipo de datos de una columna de una tabla, tendremos que borrar estos ficheros para que el BDE “note” la diferencia.

Un par de parámetros comunes está relacionado con la gestión de los campos BLOB, es decir, los campos que pueden contener información binaria, como textos grandes e imágenes:

Parámetro	Significado
<i>BLOB SIZE</i>	Tamaño máximo de un blob a recibir
<i>BLOBS TO CACHE</i>	Número máximo de blobs en caché

Ambos parámetros son aplicables solamente a los blobs obtenidos de consultas no actualizables. Incluso en ese caso, parece ser que InterBase es inmune a estas restricciones, por utilizar un mecanismo de traspaso de blobs diferente al del resto de las bases de datos.

Por último, tenemos los siguientes parámetros, que rara vez se modifican:

Parámetro	Significado
<i>BATCH COUNT</i>	Registros que se transfieren de una vez con <i>BatchMove</i>
<i>ENABLE BCD</i>	Activa el uso de <i>TBCDField</i> por la VCL
<i>MAX ROWS</i>	Número máximo de filas por consulta
<i>SQLPASSTHRU MODE</i>	Interacción entre SQL explícito e implícito
<i>SQLQRYMODE</i>	Dónde se evalúa una consulta

El BDE, como veremos más adelante, genera implícitamente sentencias SQL cuando se realizan determinadas operaciones sobre tablas. Pero el programador también puede lanzar instrucciones SQL de forma explícita. ¿Pertenecen estas operaciones a la misma transacción, o no? El valor por omisión de *SQLPASSTHRU MODE*, que es *SHARED AUTOCOMMIT*, indica que sí, y que cada operación individual de actualización se sitúa automáticamente dentro de su propia transacción, a no ser que el programador inicie explícitamente una.



*MAX ROWS*, por su parte, se puede utilizar para limitar el número de filas que devuelve una tabla o consulta. Sin embargo, los resultados que he obtenido cuando se alcanza la última fila del cursor no han sido del todo coherentes, por lo que prefiero siempre utilizar mecanismos semánticos para limitar los conjuntos de datos.

## Configuración de InterBase

InterBase es el sistema que se configura con mayor facilidad. En primer lugar, la instalación del software propio de InterBase en el cliente es elemental: sólo necesitamos una DLL, *gds32.dll*, para tener acceso a un servidor. En las versiones actuales, todavía es necesario un pequeño cambio adicional para utilizar TCP/IP como protocolo de comunicación: hay que añadir una entrada en el fichero *services*, situado en el directorio de Windows, para asociar un número de puerto al nombre del servicio de InterBase:

```
gds_db      3050/tcp
```

Ya dentro del Administrador del BDE, el único parámetro de configuración obligatoria para un alias de InterBase es *SERVER NAME*. A pesar de que el nombre del parámetro se presta a equívocos, lo que realmente debemos indicar en el mismo es la base de datos con la que vamos a trabajar, junto con el nombre del servidor en que se encuentra. InterBase utiliza una sintaxis especial mediante la cual se indica incluso el protocolo de conexión. Por ejemplo, si el protocolo que deseamos utilizar es NetBEUI, un posible nombre de servidor sería:

```
//WILMA/C:/MasterDir/VipInfo.gdb
```

En InterBase, generalmente las bases de datos se almacenan en un único fichero, aunque existe la posibilidad de designar ficheros secundarios para el almacenamiento; es éste fichero el que estamos indicando en el parámetro *SERVER NAME*; observe que, curiosamente, las barras que se emplean para separar las diversas partes de la ruta son las de UNIX. Sin embargo, si en el mismo sistema instalamos el protocolo TCP/IP y lo utilizamos para la comunicación con el servidor, la cadena de conexión se transforma en la siguiente:

```
WILMA:/MasterDir/VipInfo.gdb
```

Como puede comprender el lector, es mejor dejar vacía esta propiedad para el controlador de InterBase, y configurarla solamente a nivel de alias.

Hay algo importante para comprender: los clientes no deben (aunque pueden) tener acceso al directorio donde se encuentre el fichero de la base de datos. Esto es, en el ejemplo anterior *MasterDir* no representa un nombre de recurso compartido, sino un directorio, y preferiblemente un directorio no compartido, por razones de seguridad. La cadena tecleada en *SERVER NAME* es pasada por el software cliente al servicio

instalado en el servidor, y es este programa el que debe tener derecho a trabajar con ese fichero.

Podemos, y debemos, jugar un poco con *DRIVER FLAGS*. Si colocamos el valor 4096 en este parámetro, las grabaciones de registros individuales se producirán más rápidamente, porque el BDE utilizará la función *isc\_commit\_retaining* para confirmar las transacciones implícitas. Este modo de confirmación graba definitivamente los cambios, pero no crea un nuevo contexto de transacción, sino que vuelve a aprovechar el contexto existente. Esto acelera las operaciones de actualización. Pero lo más importante es que evita que el BDE tenga que releer los cursores activos sobre las tablas afectadas por la transacción. También se puede probar con el valor 512 en *DRIVER FLAGS*, que induce el nivel de aislamiento superior para las transacciones implícitas, el nivel de lecturas repetibles. Si quiere combinar esta constante con la que hemos explicado antes, puede sumarlas:

$$512 + 4096 = 4608$$

El nivel de lecturas repetibles no es el apropiado para todo tipo de aplicaciones. En concreto, las aplicaciones que tienen que estar pendientes de las actualizaciones realizadas en otros puestos no son buenas candidatas a este nivel.

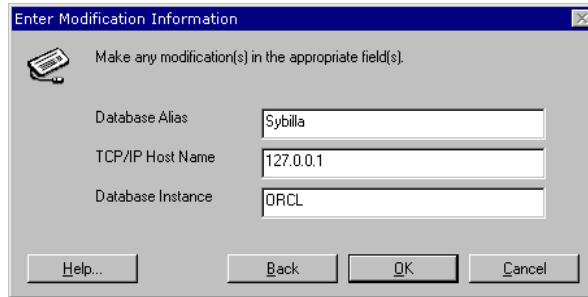
A partir de la versión 5.0.1.24 del SQL Link de InterBase, también disponemos de los siguientes parámetros:

Parámetro	Significado
<i>COMMIT RETAIN</i>	Evita releer los cursores en transacciones explícitas
<i>WAIT ON LOCKS</i>	Activa el modo de espera por bloqueos en transacciones
<i>ROLE NAME</i>	Permite al usuario asumir un rol inicialmente

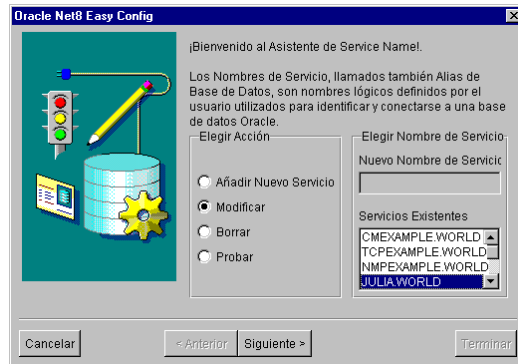
El primero de ellos, *COMMIT RETAIN*, surte el mismo que asignar 4096 en el parámetro *DRIVER FLAGS*, pero su acción se centra en las transacciones explícitas. Recuerde que activar esta opción mejorará la eficiencia del BDE al trabajar con transacciones. *WAIT ON LOCKS* fuerza a InterBase a esperar a que un registro esté disponible, en vez de fallar inmediatamente, que era lo que pasaba en versiones anteriores. Por último, *ROLE NAME* permite que el usuario especifique el perfil que desea asumir durante la conexión que va a iniciar. Los derechos del perfil se suman a los derechos que ya tenía como usuario.

## Configuración de Oracle

Para configurar un cliente de Oracle, necesitamos instalar obligatoriamente y configurar el software SQL Net que viene con este producto. La siguiente imagen corresponde a uno de los cuadros de diálogo de SQL Net Easy Configuration, la aplicación de configuración que acompaña a Personal Oracle. Los datos corresponden a una conexión que se establece a un servidor situado en la propia máquina. Observe que la dirección IP suministrada es la 127.0.0.1.



La siguiente imagen corresponde al SQL Net Easy Configuration que acompaña a la versión Enterprise 8.0.5. A pesar de las diferencias en formato, el procedimiento de conexión sigue siendo básicamente el mismo, y no ha cambiado demasiado en las versiones posteriores de Oracle:



Una vez que hemos configurado un alias con SQL Net, podemos acceder a la base de datos correspondiente. El parámetro *SERVER NAME* del controlador de Oracle se refiere precisamente al nombre de alias que acabamos de crear.

Parámetro	Significado
<i>VENDOR INIT</i>	Nombre de DLL suministrada por Oracle
<i>NET PROTOCOL</i>	Protocolo de red; casi siempre TNS
<i>ROWSET SIZE</i>	Número de registros que trae cada petición
<i>ENABLE INTEGERS</i>	Traduce columnas de tipo <i>NUMERIC</i> sin escala a campos enteros de la VCL
<i>LIST SYNONYMS</i>	Incluir nombres alternativos para objetos

*ROWSET SIZE* permite controlar una buena característica de Oracle. Este servidor, al responder a los pedidos de registros por parte de un cliente, se adelanta a nuestras intenciones y envía por omisión los próximos 20 registros del cursor. Así se aprovecha mejor el tamaño de los paquetes de red. Debe experimentar, de acuerdo a su red y a sus aplicaciones, hasta obtener el valor óptimo de este parámetro.

A partir de la versión 5 del BDE, tenemos dos nuevos parámetros. El primero de ellos es *DLL32*, en el cual podemos asignar uno de los valores *SQLORA32.DLL* ó *SQLORA8.DLL*. Y es que existen dos implementaciones diferentes del SQL Link de Oracle; *SQLORA32* sólo sirve para las versiones de Oracle anteriores a la 8. El segundo nuevo parámetro es *OBJECT MODE*, que permite activar las extensiones de objetos de Oracle para que puedan ser utilizadas desde Delphi.

### PROBLEMAS FRECUENTES

El problema más frecuente al configurar el SQL Link de Oracle es el mensaje “*Vendor initialization failure*”. Las dos causas más probables: hemos indicado en el parámetro *VENDOR INIT* una DLL que no corresponde a la versión de Oracle instalada, o que dicha DLL no se encuentre en el PATH del sistema operativo. La última causa puede parecer una enfermedad infantil fácilmente evitable, pero no lo crea: muchos programas de instalación modifican el PATH en el fichero *autoexec.bat* utilizando nombres largos que contienen espacios. Al arrancar la máquina, el comando da el error “*Demasiados parámetros*”, pues interpreta el espacio como separador de parámetros. Y como casi nadie mira lo que hace su ordenador al arrancar...

## Configuración de MS SQL Server y Sybase

Voy a detenerme en los parámetros de SQL Server porque son muy similares a los de Sybase. Pero recuerde que no es buena idea utilizar el BDE para trabajar con el servidor de Microsoft.

En esos sistemas, *SERVER NAME* representa el nombre del servidor en el cual se encuentran las bases de datos. Este nombre suele coincidir con el nombre del ordenador dentro del dominio, pero puede ser diferente si el nombre del ordenador contiene acentos o caracteres no válidos para un identificador SQL. Una vez que hemos especificado con qué servidor lidiaremos, hay que especificar el nombre de la base de datos situada en ese servidor, en el parámetro *DATABASE NAME*.

En contraste con InterBase, estos controladores tienen muchos más parámetros que pueden ser configurados. La siguiente tabla ofrece un resumen:

Parámetro	Significado
<i>CONNECT TIMEOUT</i>	Tiempo máximo de espera para una conexión, en segundos
<i>TIMEOUT</i>	Tiempo máximo de espera para un bloqueo
<i>MAX QUERY TIME</i>	Tiempo máximo de espera para la ejecución de una consulta
<i>BLOB EDIT LOGGING</i>	Desactiva las modificaciones transaccionales en campos BLOB
<i>APPLICATION NAME</i>	Identificación de la aplicación en el servidor
<i>HOST NAME</i>	Identificación del cliente en el servidor
<i>DATE MODE</i>	Formato de fecha: 0= <i>mdy</i> , 1= <i>dmy</i> , 2= <i>ymd</i>
<i>TDS PACKET SIZE</i>	Tamaño de los paquetes de intercambio
<i>MAX DBPROCESSES</i>	Número máximo de procesos en el cliente

Quizás el parámetro *MAX DBPROCESSES* sea el más importante de todos. La biblioteca de acceso utilizada por el SQL Link de Borland para MS SQL Server es la DB-Library. Esta biblioteca, en aras de aumentar la velocidad, sacrifica el número de cursores que pueden establecerse por conexión de usuario, permitiendo solamente uno. Así que cada tabla abierta en una estación de trabajo consume una conexión de usuario, que en la versión 6.5 de SQL Server, por ejemplo, necesita 55KB de memoria en el servidor. Es lógico que este recurso se limite a un máximo por cliente, y es ese el valor que se indica en *MAX DBPROCESSES*. Este parámetro solamente puede configurarse en el controlador, no en el alias.

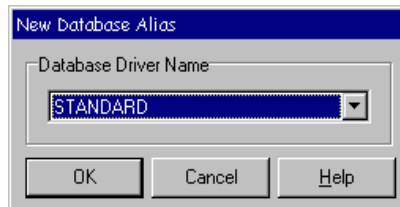
Por su parte, el parámetro *TDS PACKET SIZE* está relacionado con la opción *network packet size* del procedimiento *sp\_configure* de la configuración del servidor. Se puede intentar aumentar el valor del parámetro para una mayor velocidad de transmisión de datos, por ejemplo, a 8192. TDS quiere decir *Tabular Data Stream*, y es el formato de transmisión utilizado por MS SQL Server.

#### CONFIGURACION DE OTROS SISTEMAS

Con un poco de sentido común y la ayuda en línea, se puede descifrar el significado de los restantes parámetros de otros controladores. DB2 utiliza el parámetro *DB2 DSN* para indicar la base de datos con la que se quiere trabajar; este nombre se crea con la herramienta correspondiente de catalogación en clientes. Informix tiene un mecanismo similar a SQL Server: es necesario suministrar *SERVER NAME* y *DATABASE NAME*. El parámetro *LOCK MODE* indica el número de segundos que un proceso espera por la liberación de un bloqueo; por omisión, se espera 5 segundos. Y el formato de fechas se especifica mediante los parámetros *DATE MODE* y *DATE SEPARATOR*.

## Creación de alias para bases de datos locales y SQL

Una vez que sabemos cómo manejar los parámetros de configuración de un controlador, es cosa de niños crear alias para ese controlador. La razón es que muchos de los parámetros de los alias coinciden con los de los controladores. El comando de menú *Object|New*, cuando está activa la página *Databases* del Administrador del Motor de Datos, permite crear nuevos alias. Este comando ejecuta un cuadro de diálogo en el que se nos pide el nombre del controlador, y una vez que hemos decidido cuál utilizar, se incluye un nuevo nodo en el árbol, con valores por omisión para el nombre y los parámetros. A continuación, debemos modificar estos valores.



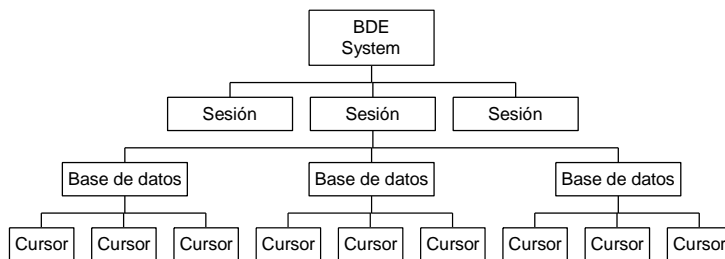
Para crear un alias de Paradox ó dBase debemos utilizar el controlador *STANDARD*. El principal parámetro de este tipo de alias es *PATH*, que debe indicar el directorio (sin la barra final) donde se encuentran las tablas. El parámetro *DEFAULT DRIVER* especifica qué formato debe asumirse si se abre una tabla y no se suministra su extensión, *db* ó *dbf*. Note que este parámetro también existe en la página de configuración del sistema.

## Acceso a datos con el BDE

**H**OY ES EL PRIMER DÍA DE ENERO del 2002. Este es el último capítulo que me falta por terminar. Los libros se escriben de forma parecida a como se filman las películas; la diferencia principal es que no hay actrices hermosas dando vueltas alrededor. Aquí en España le dan un sentido adiós a la peseta. La bella Marianne hace lo mismo con el franco, Kunigunde deja caer una lágrima por el marco... y Millicent se abraza a una libra esterlina y le ruega que nunca la abandone. Mientras, en mi habitación, abro una selecta botella del mejor whisky escocés y brindo por el paso del BDE a una mejor vida: que su agonía sea breve. Sic transit gloria mundi.

### La arquitectura de objetos del Motor de Datos

Cuando utilizamos el BDE para acceder a bases de datos, nuestras peticiones pasan por toda una jerarquía de objetos. El siguiente esquema muestra los tipos de objetos con los que trabaja el BDE, y la relación que existe entre ellos:



El nivel superior se ocupa de la configuración global, inicialización y finalización del *sistema*: el conjunto de instancias del BDE que se ejecutan en una misma máquina. Las *sesiones* representan las diferentes aplicaciones y usuarios que acceden concurrentemente al sistema; en una aplicación de 32 bits pueden existir varias sesiones por aplicación, especialmente si la aplicación soporta concurrencia mediante hilos múltiples.

Cada sesión puede trabajar con varias bases de datos. Estas bases de datos pueden estar en distintos formatos físicos y en diferentes ubicaciones en una red. Su función

es controlar la conexión a bases de datos protegidas por contraseñas, la gestión de transacciones y, en general, las operaciones que afectan a varias tablas simultáneamente.

Por último, una vez que hemos accedido a una base de datos, estamos preparados para trabajar con los cursores. Un *cursor* es una colección de registros, de los cuales tenemos acceso a uno solo a la vez, por lo que puede representarse mediante los conjuntos de datos de la VCL. Existen funciones y procedimientos para cambiar la posición del registro activo del cursor, y obtener y modificar los valores asociados a este registro. El concepto de cursor nos permite trabajar con tablas, consultas SQL y con el resultado de ciertos procedimientos almacenados de manera uniforme, ignorando las diferencias entre estas técnicas de obtención de datos.

Cada uno de los objetos internos del BDE tiene un equivalente directo en la VCL:

Objeto del BDE	Clase de la VCL
Sesiones	<i>TSession</i>
Bases de datos	<i>TDatabase</i>
Cursores	<i>TBDEDataSet</i> y sus descendientes

Sin embargo, un programa escrito en Delphi no necesita utilizar explícitamente los objetos superiores en la jerarquía a los cursores para acceder a bases de datos. Los componentes de sesión y las bases de datos, por ejemplo, pueden ser creados internamente por la VCL, aunque el programador puede acceder a los mismos en tiempo de ejecución.

## Sesiones

Las sesiones del BDE nos ayudan a lograr los siguientes objetivos:

- 1 Cada sesión define un usuario diferente que accede al BDE. Si nuestra aplicación utiliza una misma base de datos desde diferentes hilos, es necesario que cada uno de ellos realice todas sus peticiones al BDE a través de una sesión diferente.
- 2 Las sesiones nos permiten administrar desde un mismo sitio las conexiones a bases de datos de la aplicación. Esto incluye la posibilidad de asignar valores por omisión a las propiedades de las bases de datos.
- 3 Las sesiones nos dan acceso a la configuración del BDE. De este modo podemos administrar los alias del BDE y extraer información de esquemas de las bases de datos.
- 4 Mediante las sesiones, podemos controlar el proceso de conexión a tablas Paradox protegidas por contraseñas.

Por supuesto, el uso que más nos interesa es el primero de todos: el acceso concurrente. ¿Qué posibilidad hay de que una aplicación necesite varios hilos de ejecución? Si se trata de una aplicación interactiva, es poco probable que lleguemos a tal ex-



tremo. En todo caso, puede que tengamos un hilo adicional, para alguna tarea que deba realizarse en segundo plano. Por ejemplo, puede que queramos lanzar un procedimiento almacenado que tarde mucho en ejecutarse, y no nos interesa esperar a que termine.

Sin embargo, otros tipos de aplicaciones sí que necesitan el acceso concurrente desde un mismo proceso. Las aplicaciones para Internet basadas en ISAPI/NSAPI son un buen ejemplo; se trata de DLLs que se “suman” a un proceso gestionado por el servidor HTTP. Otro ejemplo son los servidores de capa intermedia de DataSnap, o Midas. Aunque existen varias configuraciones físicas, la más común es mantener varios objetos remotos conviviendo dentro de un mismo proceso ejecutable.

En el primer caso, no necesitamos utilizar componentes de sesión explícitamente. La propia VCL crea una sesión calladamente, y si tenemos necesidad de trabajar con ella podemos buscarla en la variable global *Session*, definida en la unidad *DBTables*.

#### ADVERTENCIA

La inicialización del BDE tiene lugar en el código de inicio de la unidad *DBTables*. Si por algún motivo su proyecto incluye *DBTables* en la cláusula **uses** de alguna de sus unidades, sepa que la aplicación buscará el BDE al ejecutarse. Para asegurarse de que no le está pasando esto, compile el proyecto haciendo uso de los *packages*, y ejecute el comando de menú *Project|Information for...* Hay un panel *Packages Used* en el diálogo que aparece; si ve que se ha incluido *bdertl60.bpl*, tenemos problemas.

Traiga un componente *TTable* o *TDatabase* sobre un formulario. Observe que ambos tienen una propiedad *SessionName*, y que por omisión está vacía. El enlace entre un componente del BDE y su sesión no se realiza a través de punteros, como sucede entre un *TSQLDataSet* y su *TSQLConnection*, por poner un ejemplo, sino que cada componente de sesión dentro de un proceso registra su nombre en una estructura global. Si selecciona el *TTable* y despliega la lista asociada a *SessionName*, verá que ya existe un nombre de sesión llamado *Default*. Se trata, claro está, del “nombre” de la sesión implícita; da lo mismo usar una cadena vacía, porque la VCL sabe que se trata de la sesión implícita.

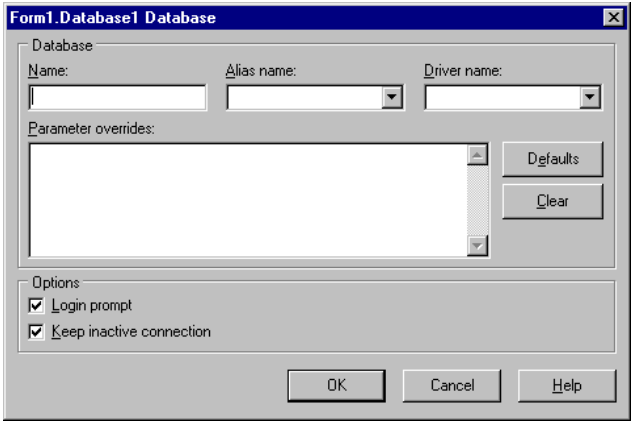
Si necesitamos una sesión adicional, de todos modos, podemos traer un componente *TSession* a algún módulo o formulario del proyecto, y asignar un nombre único globalmente en su propiedad *SessionName* de la sesión. Pero también podríamos activar su propiedad *AutoSessionName*, de tipo lógico. Esta forma de trabajo es muy útil cuando estamos desarrollando aplicaciones para Internet o servidores de capa intermedia, porque no podemos determinar de antemano cuántas sesiones vamos a necesitar: depende del número de clientes conectados.

## El componente *TDatabase*

Al igual que sucede con las sesiones, es posible escribir aplicaciones basadas en el BDE, pero que no utilicen explícitamente componentes *TDatabase*. Sin embargo, no

es algo aconsejable. Hay muchas características de las conexiones que deben configurarse a este nivel, mientras que hay solamente un par de parámetros que puede interesar configurar al nivel de la sesión.

Las propiedades de un *TDatabase* pueden editarse también mediante un cuadro de diálogo que aparece al realizar una doble pulsación sobre el componente:



Siempre que traemos un *TDatabase* a una aplicación, estamos creando un alias local a la sesión a la cual pertenece. El nombre de dicho alias viene determinado por la propiedad *DatabaseName* del componente, que en el cuadro de diálogo anterior corresponde al control etiquetado como *Name*. Básicamente, existen dos formas de configurar esa conexión:

- 1 Crear un alias a partir de cero, siguiendo casi los mismos pasos que en la configuración del BDE, especificando el nombre del alias, el controlador y sus parámetros.
- 2 Tomar como punto de partida un alias ya existente. En este caso, también se pueden alterar los parámetros de la conexión.

En cualquiera de los dos casos, la propiedad *IsSQLBased* nos dice si la base de datos está conectada a un servidor SQL o un controlador ODBC, o a una base de datos local.

Estas son las propiedad relevantes para configurar un *TDatabase*:

Propiedad	Propósito
<i>DatabaseName</i>	El nombre del alias temporal que vamos a definir
<i>SessionName</i>	La sesión dentro de la cual se creará el alias
<i>AliasName</i>	Un alias externo en el que nos vamos a basar ...
<i>DriverName</i>	... o un controlador, para comenzar desde cero
<i>Params</i>	La lista de parámetros de configuración

La propiedad *Params* es de tipo *TStrings*, y en ella se almacenan, uno por cada línea, los mismos pares parámetros/valores que se utilizan para definir alias con el Administrador del BDE. Por ejemplo:

```
SERVER NAME=localhost:C:/SqlData/Pedidos.GDB
WAIT ON LOCKS=TRUE
COMMIT RETAIN=TRUE
ENABLE SCHEMA CACHE=TRUE
SCHEMA CACHE SIZE=32
USER NAME=SYSDBA
PASSWORD=masterkey
```

### ATENCIÓN

Es sumamente importante no dejar espacios en blanco antes o después del signo de igualdad. Este es el error que se comete con mayor frecuencia al configurar un componente *TDatabase*.

Estas otras propiedades controlan algunos otros parámetros de la conexión:

Propiedad	Propósito
<i>Exclusive</i>	Abre la base de datos en exclusiva, si el controlador lo soporta
<i>ReadOnly</i>	Apertura en modo de sólo lectura
<i>TransIsolation</i>	El nivel de aislamiento de las transacciones explícitas

El BDE reconoce tres niveles de aislamiento simplificados: *tiDirtyRead*, *tiReadCommitted* y *tiRepeatableRead*. El primero de ellos solamente puede utilizarse con Paradox y dBase; además, es obligatorio establecerlo si queremos realizar transacciones con este formato de tablas. Los sistemas SQL parten del nivel de lecturas confirmadas, y hay algunos que “incluso” permiten la activación del nivel de lecturas repetibles; con algunos detalles “entrañables”, como que las bases de datos de Oracle sólo permiten lecturas en ese nivel...

## Tablas y consultas

Hay dos tipos básicos de componentes que encapsulan cursores: las tablas, *TTable*, y las consultas, *TQuery*; es cierto que existen otros componentes como los procedimientos almacenados y las tablas anidadas, pero pueden asimilarse a una de las dos categorías mencionadas.

Veamos primero las tablas. Para configurar una tabla se necesitan pocas propiedades:

- Si estamos utilizando varias sesiones, el nombre de la sesión en *SessionName*.
- Un nombre de alias, en *DatabaseName*. Puede ser un alias persistente, definido con el Administrador del BDE, o un alias de sesión, creado con un *TDatabase*.
- Finalmente, el nombre de la tabla, en *TableName*. ¿Ve qué sencillo ha sido?

A partir de ese momento, ya puede crear los componentes de acceso a campo, abrir y cerrar el cursor, navegar, poner el componente en modo de edición, modificar sus campos, grabar, borrar registros... exactamente igual que si estuviera trabajando con un conjunto de datos clientes. Incluso hay un par de propiedades excluyentes, *IndexName* e *IndexFieldNames*, para seleccionar un criterio de ordenación. No obstante, existen diferencias importantes en la forma en que se implementan todas esas operaciones, diferencias que veremos más adelante.

También es muy fácil configurar una consulta. Las dos primeras propiedades a tener en cuenta son *SessionName* y *DatabaseName*, al igual que sucede con las tablas. Sólo cambia el tercer paso, y se añade uno nuevo:

- Hay que asignar una instrucción SQL en la propiedad del mismo nombre.
- La consulta puede especificar parámetros, con la misma sintaxis que en DB Express. En tal caso, hay que asignar el tipo a cada parámetro, dentro de la propiedad *Params*, y posteriormente debemos asignarles valores para poder abrir la consulta.

Es decir, un *TQuery* es muy parecido a un *TSQLQuery*, excepto que podemos navegar en todas las direcciones, si usamos su configuración por omisión. Aparte, tenemos un par de propiedades para cambiar el funcionamiento de la consulta:

- *Unidirectional*: vale *False* por omisión. Como imaginará, al activarla hacemos que la navegación sea posible solamente en una dirección, pero se tarda menos tiempo en leer los registros desde el servidor.
- *RequestLive*: también vale *False* por omisión. Puede pedir al BDE que la consulta sea “modificable” asignando *True* en esta propiedad. Eso no significa que siempre se pueda complacer al programador. Si estamos trabajando con una base de datos de escritorio, se solicita *RequestLive* y no se puede satisfacer la solicitud, la consulta se abre, pero si comprobamos la propiedad de sólo lectura *CanModify*, veremos que sigue siendo *False*. En cambio, si el fallo ocurre con un controlador SQL, se produce una excepción.

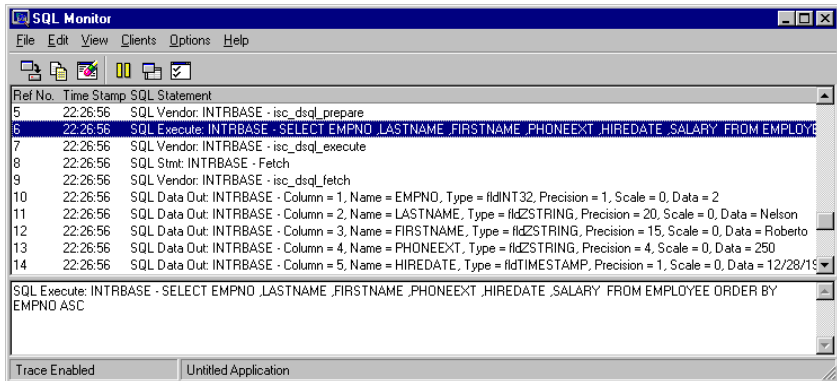
A partir de este momento, amigo mío, puede hacer con estos componentes casi todo lo que ya sabe hacer con los componentes de MyBase y DB Express: mover el cursor, buscar registros, poner filtros y todo lo que le dicte su imaginación. El único problema es ver cómo se implementan estas operaciones internamente, porque las diferencias en velocidad pueden ser decisivas.

## Navegación sobre tablas SQL

Y ahora viene la parte más importante de cualquier explicación sobre los componentes del BDE. ¿Qué tipo de componentes debemos utilizar en nuestras aplicaciones? Eliminemos antes un caso en el que es fácil decidir. Si debe trabajar con bases de datos de escritorio, como Paradox o los diversos dialectos de dBase, utilice *TTable*

todo lo que pueda. Habrá momentos en los que preferirá usar un *TQuery*: si necesita un informe con subtotales, por ejemplo, no tiene mucho sentido hacer los cálculos manualmente. Pero recuerde que hay muchos recursos en Delphi que fueron introducidos simplemente para facilitar la programación con Paradox y dBase. Por ejemplo, los campos de búsqueda (*lookup fields*) evitan el uso de encuentros entre tablas, y los filtros son una forma primitiva de sustituir una cláusula **where**.

Sin embargo, cuando se trata de bases de datos SQL, la respuesta no es sencilla, aunque pueda parecerlo. En esta sección expondré, a grandes rasgos, lo que sucede cuando se utiliza una tabla para acceder a una base de datos SQL. Para averiguarlo, he utilizado una herramienta muy importante del BDE: SQL Monitor. Puede ejecutarlo desde el Entorno de Desarrollo, mediante el comando *Database | SQL Monitor*. O si lo prefiere, desde el grupo de programas de Delphi, en el menú de inicio del sistema. Este es su aspecto:



SQL Monitor funciona de forma parecida al monitor de DB Express: intercepta y muestra todos los comandos que realiza el BDE a la interfaz SQL nativa con la que nos conectamos. La única, e importante, diferencia es que SQL Monitor puede funcionar como una aplicación independiente.

Para explicar el funcionamiento del BDE voy a utilizar la base de datos *mastsqldb* de InterBase. Vamos a suponer que tenemos un componente *TTable* asociado a la tabla *Employee*, y que es la primera vez que la abrimos. Verá aparecer un montón de instrucciones en el SQL Monitor: antes de empezar a recuperar los datos verdaderos, el BDE necesita recoger información sobre qué campos tiene la tabla, cuáles son sus tipos, con qué índices contamos, cuál es la clave primaria... En una base de datos SQL la información necesaria tiene que obtenerse interrogando las tablas del sistema. Esta es una muestra de las consultas que envía el BDE al servidor antes de comenzar a traer los datos de empleados:

```
select rdb$owner_name, rdb$relation_name, rdb$system_flag,
       rdb$view_blr, rdb$relation_id
from   rdb$relations
where  rdb$relation_name = 'employee'
```

```

select r.rdb$field_name, f.rdb$field_type, f.rdb$field_sub_type,
        f.rdb$dimensions, f.rdb$field_length, f.rdb$field_scale,
        f.rdb$validation_blr, f.rdb$computed_blr,
        r.rdb$default_value, f.rdb$default_value, r.rdb$null_flag
from   rdb$relation_fields r, rdb$fields f
where  r.rdb$field_source = f.rdb$field_name and
        r.rdb$relation_name = 'employee'
order by r.rdb$field_position asc

select i.rdb$index_name, i.rdb$unique_flag, i.rdb$index_type,
        f.rdb$field_name
from   rdb$indices i, rdb$index_segments f
where  i.rdb$relation_name = 'employee' and
        i.rdb$index_name = f.rdb$index_name
order by i.rdb$index_id, f.rdb$field_position asc

select r.rdb$field_name, f.rdb$validation_blr, f.rdb$computed_blr,
        r.rdb$default_value, f.rdb$default_value, r.rdb$null_flag
from   rdb$relation_fields r, rdb$fields f
where  r.rdb$field_source = f.rdb$field_name and
        r.rdb$relation_name = 'employee'
order by r.rdb$field_position asc

```

¿Cuánto tiempo tarda recuperar todos estos datos? El número total de registros, como podrá comprobar, puede obtenerse mediante la siguiente fórmula:

$$Total = 1 + 2 * Campos + \sum CamposIndices_i$$

En otras palabras, hay que traer siempre un registro de la primera consulta. La segunda y la cuarta consulta devuelven un registro por cada columna que tenga la tabla; por eso hay que multiplicar por dos el número de campos. Por último, la tercera consulta devuelve una fila por cada columna que participa en un índice, por lo que hay que sumar, para cada índice, el número de columnas que hay en su definición.

Para la tabla *Employee*, el BDE necesita leer 16 registros, pero se trata de una tabla atípica, con sólo seis columnas. Algunas tablas características de aplicaciones reales pueden tener cincuenta, ochenta, cien columnas (aunque después se utilicen menos de la mitad). Para una tabla de cien columnas, el BDE tendría que leer al menos doscientos registros, ¡antes de devolver una sola fila de datos! Claro, eso puede llevar algún tiempo, y fue la causa de que muchos programadores difundieran el mito de que *TTable* necesita cargar todos los registros en memoria antes de echar a andar.

Por supuesto, hay una solución para este problema: activa la opción *ENABLE SCHEMA CACHE* de la conexión, y no olvide aumentar el valor de *SCHEMA CACHE SIZE* hasta su valor máximo, 32. Al hacerlo, el BDE guardará esta información en una caché local, en disco. La próxima vez que ejecute la aplicación y abra la tabla, la información sobre esquemas se leerá de la caché.

Superado este prólogo, el BDE crea una instrucción SQL de selección a partir de la información obtenida, y abre un cursor. Esta es la consulta que se crea para la tabla de empleados:

```
select empno, firstname, lastname, phoneext, hiredate, salary
from   employee
order  by empno asc
```

La cláusula **order by** ha aparecido por iniciativa del BDE, porque no hemos indicado criterio alguno de ordenación. La explicación que sigue asume este detalle, para mayor sencillez, pero es posible extenderla al caso en que se modifican *IndexName* o *IndexFieldNames*.

A partir del momento en que el cursor está abierto, podemos ir navegando hacia delante, que el BDE traerá registros mediante la instrucción **fetch**. Si navegamos hacia atrás, no se pedirá nada al servidor, porque los registros van almacenándose en una caché en memoria. Compliquemos el cuadro, y ejecutemos el método *Last*, para ir al último registro. Veremos entonces que el BDE cierra el cursor activo, y ejecuta esta otra instrucción:

```
select empno, firstname, lastname, phoneext, hiredate, salary
from   employee
order  by empno desc
```

La primera fila que devuelve, al estar ordenada descendentemente, ¡es la última fila del cursor original! Si a partir del último registro navegamos hacia atrás, el BDE lo resuelve con **fetchs** adicionales, que se van almacenando en una nueva caché; los registros de la zona inicial de la tabla se eliminan de memoria.

Conociendo estos detalles, ya podemos plantear algunos requisitos necesarios para que un *TTable* funcione correctamente:

- 1 Es obligatorio que la tabla base tenga una clave primaria. Además, mientras más sencilla y compacta sea, mucho mejor. Por varios motivos adicionales, relacionados con las técnicas de diseño, prefiero las que consisten en una sola columna, de tipo entero binario.
- 2 Es fundamental que las consultas del tipo “dame registros ordenados ascendentemente” y descendentemente se compilen eficientemente en el servidor, al menos para los criterios de ordenación que establezca la ordenación.

He incluido el segundo punto pensando en InterBase. Como recordará, sus índices ascendentes no pueden utilizarse para optimizar consultas con ordenación descendente. Si quiere poder abrir un componente *TTable* en InterBase, y no perder los pelos en el intento, deberá añadir índices descendentes, aparentemente redundantes. El fallo en cumplir con este requerimiento ayudó a crear la leyenda negra sobre los componentes de tablas.

## Búsquedas y filtros sobre componentes de tablas

Según lo explicado, un *TTable* utiliza trucos inteligentes para navegar con las cuatro operaciones básicas: *First*, *Prior*, *Next* y *Last*. ¿Qué tal se desempeña con operaciones más complicadas? Supongamos que se ejecuta la siguiente instrucción:

```
Table1.Locate('EMPNO', 36, []);
```

Se trata de una búsqueda sobre la clave primaria. Según SQL Monitor, esta es la instrucción que se envía al servidor:

```
select empno, firstname, lastname, phoneext, hiredate, salary
from   employee
where  empno = ?
```

La sentencia tiene un parámetro, representado por un signo de interrogación. Para ejecutarla, el BDE utiliza antes una instrucción *Data in*, en la que proporciona el valor pasado en el segundo parámetro de *Locate* a la instrucción SQL. Lo interesante es lo que sucede cuando nos movemos con *Prior* y *Next* a partir del registro seleccionado. Si buscamos el registro anterior después de haber localizado uno, se genera la siguiente instrucción:

```
select empno, firstname, lastname, phoneext, hiredate, salary
from   employee
where  empno < ?
order by empno desc
```

Es decir, el BDE abre un cursor descendente con los registros cuyo código es menor que el actual. El primer registro que devuelve es el registro al que tenemos que navegar. Y se vuelve a inicializar una caché con los registros del nuevo cursor. Muy listo. ¿Qué tal si pedimos una búsqueda aproximada, por una columna que no pertenezca a la clave primaria?

```
Table1.Locate('LASTNAME', 'Guck', [loPartialKey]);
```

Observe esta vez la presencia de la opción *loPartialKey*. Si vamos a SQL Monitor, encontraremos las siguientes instrucciones:

```
select empno, firstname, lastname, phoneext, hiredate, salary
from   employee
where  lastname = ?

select empno, firstname, lastname, phoneext, hiredate, salary
from   employee
where  lastname > ?
order by lastname asc
```

La primera consulta intenta averiguar si hay alguien cuyo apellido sea exactamente “Guck”. Si no lo hubiera, razona así: si un apellido comienza con “Guck”, es posterior alfabéticamente a este prefijo. Es eso lo que busca la segunda consulta: el primer



registro cuyo apellido es mayor que el patrón de búsqueda suministrado. Después de localizar el registro, las operaciones de navegación que se ejecuten se comportan igual que antes.

### ADVERTENCIA

Cuando sí se produce es desastre es al añadir la opción *loCaseInsensitive* a *Locate*, porque se resuelve entonces mediante una búsqueda secuencial. Quiero aclarar que el API del BDE solamente soporta las búsquedas sobre las columnas del criterio de ordenación activo. La VCLDB se encarga de implementar los restantes tipos de búsquedas. De modo que si tuviéramos que culpar a alguien, deberíamos mirar a Delphi.

¿Y los filtros, qué tal van? Normalmente, *TTable* añade la condición a la cláusula **where** del cursor:

```
select empno, firstname, lastname, phoneext, hiredate, salary
from employee
where salary >= ?
order by emp_no asc
```

Tenga en cuenta que las operaciones de navegación generan a veces condiciones para la cláusula **where**. Pero no hay problemas: el BDE mezcla correctamente estas condiciones con cualquier filtro que aplicamos. Eso sí, debemos evitar dos cosas. En primer lugar, jamás activar la opción *foCaseInsensitive* en la propiedad *FilterOptions*. Y no debemos utilizar comparaciones de prefijos que utilicen la siguiente sintaxis, propia del BDE:

```
LastName = 'A*'
```

También se implementan de forma eficiente los filtros latentes, que ya hemos visto antes, en el capítulo 20.

## La implementación de las consultas

En contraste, la implementación de *TQuery* no tiene tantos secretos. El BDE siempre abre un cursor unidireccional en el servidor, pero va acumulando los registros recuperados en una caché, en el lado cliente. Si ejecuta el método *Last*, para ir al último registro de la consulta, forzará al BDE a que traiga todos los registros a la caché. Si llama a *Locate*, se leerán todos los registros consecutivamente hasta dar con el que satisface la condición de búsqueda. Este comportamiento obliga a que, si quiere usar *TQuery*, deba añadir siempre alguna condición en la consulta que limite el resultado a un conjunto de filas de tamaño razonable.

¿Y qué pasa con las actualizaciones? En primer lugar, un *TQuery* tiene por omisión su propiedad *RequestLive* a *False*, es decir, no es actualizable. Cuando se abre una consulta en ese estado, no hay necesidad de leer la información de las tablas de sistema para averiguar la clave primaria y esas cosas. No obstante, si activamos *RequestLive*, la consulta debe traer al cliente la misma información que hemos visto que maneja

*TTable*. Se pierden las ventajas iniciales de la consulta, y tenemos que seguir sufriendo la navegación “tonta” registro a registro.

Para rematar, el comportamiento de una consulta actualizable durante las inserciones deja mucho que desear. Pongamos que hay diez registros en la tabla de países, que abrimos una consulta actualizable basada en esa tabla y que insertamos un nuevo país. Pues bien, después de la inserción verá que desaparece uno de los países del cursor. Si había 10 filas al principio, seguirá habiendo 10 filas en la caché, no importa cuántos nuevos registros añadamos. Será necesario cerrar la consulta y volver a abrirla para ver todas nuestras inserciones.

Hay soluciones, por supuesto. El extraño comportamiento de las inserciones se resuelve activando las actualizaciones en caché, que estudiaremos más adelante. Y si utilizamos un objeto *TUpdateSQL* estando la caché activa, tampoco será necesario consultar las tablas de sistema para saber cómo generar las actualizaciones sobre la base de datos. Esta última técnica también la estudiaremos al final del capítulo.

## Extensiones para los tipos de objetos de Oracle 8

Un motivo para decidir usar el BDE, a pesar de todo, es tener entre manos una base de datos de Oracle diseñada con extensiones orientadas a objetos, y tener que desarrollar programas para ella. La versión actual de DB Express sólo permite trabajar con los tipos ADT, pero todavía no reconoce las tablas anidadas. Sin embargo, no hay problema alguno en utilizar el BDE con estas tablas.

Para poder manejar los nuevos tipos de datos de Oracle 8, Delphi cuenta con cuatro tipos de campos, un tipo especial de conjunto de datos y un par de propiedades no muy conocidas en el componente *TDBGrid*, para poder visualizar datos de objetos. Los cambios en el BDE son los siguientes:

- 1 El parámetro *DLL32* del controlador de Oracle debe ser *sqlora8.dll*.
- 2 *VENDOR INIT* debe ser *oci.dll*.
- 3 *OBJECT MODE* debe valer *TRUE*.

Los tipos de campos mencionados son:

Tipo	Significado
<i>TADTField</i>	Para los objetos anidados
<i>TArrayField</i>	Representa un vector
<i>TDataSetField</i>	Tablas anidadas
<i>TReferenceField</i>	Contiene una referencia a un objeto compartido

Y el nuevo conjunto de datos es *TNestedTable*, que representa al conjunto de filas contenidas en un campo de tipo *TDataSetField*.

Comencemos por el tipo de campo más sencillo: el tipo *TADTField*. Pongamos por caso que hayamos creado la siguiente tabla:

```
create type TFraction as object (
    Numerador    number(9),
    Denominador  number(9)
);
/

create table Probabilidades (
    Suceso        varchar2(30) not null primary key,
    Probabilidad  TFraction not null
);
```

Ya en Delphi, traemos un *TTable*, lo conectamos a esta tabla y traemos todos los campos, mediante el comando *Add all fields*. Estos son los punteros a campos que son creados:

```
tbProb: TTable;
tbProbSUCEO: TStringField;
tbProbPROBABILIDAD: TADTField;
tbProbPROBABILIDADNUMERADOR: TFloatField;
tbProbPROBABILIDADDENOMINADOR: TFloatField;
```

Es decir, podemos trabajar directamente con los campos básicos de tipos simples, o acceder a la columna que contiene el objeto mediante el tipo *TADTField*. Por ejemplo, todas estas asignaciones son equivalentes:

```
tbProbNUMERADOR.Value := 1;
tbProbPROBABILIDAD.FieldValues[0] := 1;
tbProbPROBABILIDAD.Fields.Fields[0].AsInteger := 1;
tbProb['PROBABILIDAD.NUMERADOR'] := 1;
```

O sea, que hay más de un camino para llegar a Roma. Ahora mostraremos la tabla en una rejilla. Por omisión, este será el aspecto de la rejilla:

SUCEO	PROBABILIDAD
Acertar Lotería	(1; 14000000)
Pagar impuestos	(1; 1)
Pillar un resfriado	(3; 16)

El campo ADT aparece en una sola columna. En sus celdas, el valor de campo se representa encerrando entre paréntesis los valores de los campos más simples; estas celdas no son editables. Las columnas de un *TDBGrid* tienen una propiedad *Expanded*, que en este caso vale *False*. Cuando se pulsa sobre la flecha que aparece a la derecha de la columna *Probabilidad*, la columna se expande en sus componentes, que sí se pueden modificar:

SUCESO	PROBABILIDAD	NUMERADOR	DENOMINADOR
Acerter Lotería		1	14000000
Pagar impuestos		1	1
Pillar un resfriado		3	16

Un campo de tipo *TADTField* puede mostrarse, en modo sólo lectura, en cualquier otro control de datos, como un *TDBEdit*. Sin embargo, es más lógico que estos controles se conecten a los componentes simples del ADT. Por ejemplo, si quisiéramos editar la tabla anterior registro a registro, necesitaríamos tres controles de tipo *TDBEdit*, uno para el suceso, otro para el numerador y el último para el denominador.

Vamos ahora con las tablas anidadas y el campo *TDataSetField*. Este campo tampoco tiene previsto editarse directamente en controles de datos. Ahora bien, cuando una columna de una rejilla está asociada a uno de estos campos, al intentar una modificación aparece una ventana emergente con otra rejilla, esta vez asociada a las filas anidadas de la fila maestra activa, como en la siguiente imagen:

NOMBRE	CONTINENTE	CIUDADES
España	Europa	(DATASET)
France	Europa	(DATASET)
United States of America	América	(DATASET)

NOMBRE	POBLACION	AIRERESPIRABLE
New York		N
Miami		N
L.A.		N

¿De dónde ha salido la tabla anidada? Delphi ha configurado un objeto de tipo *TNestedTable* y lo ha acoplado a la propiedad *NestedDataSet* del campo *TDataSetField*. Sin embargo, es más frecuente que configuremos nosotros mismos un componente *TNestedTable* para mostrar la relación uno/muchos explícitamente. La propiedad principal de este componente es *DataSetField*, y no tenemos que indicar base de datos ni nombre de tabla, pues estos parámetros se deducen a partir del campo.

Nombre	Continente	Nombre
España	Europa	Madrid
France	Europa	Barcelona
United States of America	América	Sevilla

**ADVERTENCIA**

He tenido problemas con el BDE que acompaña a Delphi 6 para trabajar con tablas anidadas. Me ha vuelto a aparecer un error ya conocido de otras versiones: algunas filas se duplican en el lado cliente. Supongo, de todos modos, que pasará lo de siempre: Borland sacará un parche más tarde o temprano.

Es conveniente, además, tener un poco de precaución al editar tablas anidadas, pues no siempre se refresca correctamente su contenido al realizarse inserciones. ¿Motivo?, el hecho de que el BDE siga dependiendo de las claves primarias para identificar registros, cuando al menos en este caso debería utilizar el identificador de fila, o *rowid*. Mi consejo es, mientras no aparezca otra técnica mejor, llamar al método *Post* de la tabla principal en el cuerpo de los eventos *BeforeInsert* y *AfterPost* de la tabla anidada.

Los campos de referencia, *TReferenceField*, son muy similares a las tablas anidadas; de hecho, esta clase descende por herencia de *TDataSetField*. La única diferencia consiste en que la tabla anidada que se le asocia contiene como máximo una sola fila. Delphi permite extraer de forma sencilla los valores del objeto asociado a la referencia, el equivalente del operador **deref** de Oracle:

```
ShowMessage('Cliente: ' +
    tbPedidoRefCliente.Fields[0].DisplayText);
```

Sin embargo, para hacer que un campo de referencia apunte a un objeto existente necesitamos apoyarnos en una consulta o procedimiento almacenado que devuelva la referencia a dicho objeto. Antes habíamos mostrado un pequeño ejemplo en el que los pedidos tenían una referencia a una tabla de objetos clientes. Supongamos que en nuestra aplicación, al introducir un pedido, el usuario selecciona un cliente mediante una ventana emergente o algún mecanismo similar. Para asignar la referencia puede utilizar el siguiente código:

```
with TQuery.Create(nil) do
try
    DatabaseName := tbPedidos.DatabaseName;
    SQL.Text := 'select ref(Cli) from Clientes Cli ' + ;
                'where Cli.Nombre = ' + tbClientesNombre.Value;
    Open;
    // Esta es la asignación deseada
    tbPedidoCliente.Assign(Fields[0]);
finally
    Free;
end;
```

## Control explícito de transacciones

Como ya el lector se habrá dado cuenta, el BDE en su modo de trabajo habitual considera que cada actualización realizada sobre una tabla está aislada lógicamente de las demás posibles actualizaciones. Para base de datos locales, esto quiere decir sencillamente que no hay transacciones involucradas en el asunto. Para un sistema SQL se

utilizan transacciones implícitas, a no ser que el *SQLPASSTHRU MODE* de la conexión valga *SHARED NOAUTOCOMMIT*.

El objeto encargado de activar las transacciones explícitas es el componente *TDatabase*. Los tres métodos que ofrece *TDatabase* para el control explícito de transacciones son:

```
procedure TDatabase.StartTransaction;
procedure TDatabase.Commit;
procedure TDatabase.Rollback;
```

La propiedad *InTransaction*, disponible en tiempo de ejecución, nos avisa si hemos iniciado alguna transacción sobre la base de datos activa:

```
procedure TForm1.TransaccionClick(TObject *Sender)
begin
    miIniciarTransaccion.Enabled := not Database1.InTransaction;
    miConfirmarTransaccion.Enabled := Database1.InTransaction;
    miCancelarTransaccion.Enabled := Database1.InTransaction;
end;
```

Una transferencia bancaria que se realice sobre bases de datos de escritorio, por ejemplo, podría programarse del siguiente modo:

```
// Iniciar la transacción
Database1->StartTransaction();
try
    if not Table1.Locate('Apellidos', 'Einstein', []) then
        DatabaseError('La velocidad de la luz no es un límite');
    Table1.Edit;
    Table1['Saldo'] := Table1['Saldo'] - 10000;
    Table1.Post;
    if not Table1.Locate('Apellidos', 'Newton', []) then
        DatabaseError('No todas las manzanas caen al suelo');
    Table1.Edit;
    Table1['Saldo'] := Table1['Saldo'] + 10000;
    Table1.Post;
    // Confirmar la transacción
    Database1.Commit;
except
    // Cancelar la transacción
    Table1.Cancel;
    Database1.Rollback;
    Table1.Refresh;
    raise;
end;
```

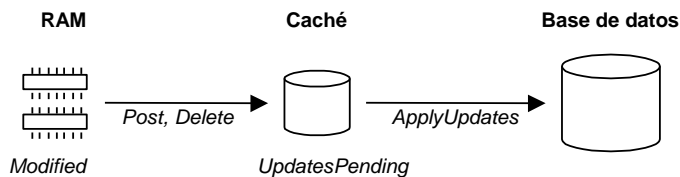
Observe que la presencia de la instrucción **raise** al final de la cláusula **except** garantiza la propagación de una excepción no resuelta, de modo que no quede enmascarada; de esta forma no se viola la Tercera Regla de Marteens.

## Actualizaciones en caché

Todos los conjuntos de datos del BDE tienen una propiedad lógica llamada *Cached-Updates*. Cuando se activa, las operaciones *Post* y *Delete* no aplican directamente los cambios en el servidor SQL, sino que registran la modificación en una caché interna. El programador deberá decidir entonces cuándo desea aplicar todos los cambios pendientes sobre la base de datos; veremos también que es posible cancelar las modificaciones de forma selectiva o total.

Son muchas las ventajas de utilizar actualizaciones en caché. En primer lugar, el tiempo requerido por una transacción se acorta, y, como sabemos, mientras más breve sea una transacción, mejor. Por otra parte, el número de paquetes enviados por la red puede disminuir drásticamente. Cuando no están activas las actualizaciones en caché, cada registro grabado provoca el envío de un paquete de datos. Cada paquete, además, va precedido de cierta información de control, y lo más probable es que se desaproveche parte de su capacidad. También se benefician los sistemas SQL que utilizan bloqueos internamente para garantizar las lecturas repetibles, porque los bloqueos impuestos estarán activos mucho menos tiempo, sólo durante la ejecución del método *ApplyUpdates*. Finalmente, veremos que al activar la caché el BDE nos permite más control sobre la forma exacta en que se ejecutan las actualizaciones en el servidor; se trata, no obstante, de un “efecto secundario” de la técnica.

Se puede conocer si existen actualizaciones en la caché, pendientes de ser aplicadas, consultando la propiedad de tipo lógico *UpdatesPending* del conjunto de datos. Note que *UpdatesPending* sólo informa acerca de las actualizaciones realizadas con *Post* y *Delete*; si la tabla se encuentra en alguno de los modos de edición *dsEditModes* y se han realizado asignaciones a los campos, esto no se refleja en *UpdatesPending*, sino en la propiedad *Modified*, como siempre.



La activación de la caché de actualizaciones es válida únicamente para el conjunto de datos implicado. Si activamos *CachedUpdates* para un objeto *TTable*, y creamos otro objeto que se refiera a la misma tabla física, los cambios realizados en la primera no son visibles desde la segunda hasta que no se realice la confirmación de los mismos.

Una vez que las actualizaciones en caché han sido activadas, los registros del conjunto de datos se van cargando en el cliente en la medida en que el usuario va leyendo y realizando modificaciones. Es posible, sin embargo, leer el conjunto de datos completo desde un servidor utilizando el método *FetchAll*.

```
procedure TDBDataSet.FetchAll;
```

De esta forma, se logra replicar el conjunto de datos en el cliente. No obstante, este método debe usarse con cuidado, debido al gran volumen de datos que puede duplicar.

### ADVERTENCIA

Por desgracia, hay fallos importantes dentro del BDE relacionados con la caché de actualizaciones, que se ponen de manifiesto cuando trabajamos con relaciones maestro/detalles. El uso de conjuntos de datos clientes es el sustituto recomendado frente a las actualizaciones en caché nativas.

## Confirmación de las actualizaciones

Existen varios métodos para la confirmación definitiva de las actualizaciones en caché. El más sencillo es el método *ApplyUpdates*, que se aplica a objetos de tipo *TDatabase*. *ApplyUpdates* necesita, como parámetro, la lista de tablas en las cuales se graban, de forma simultánea, las actualizaciones acumuladas en la caché:

```
procedure TDatabase.ApplyUpdates (
  const DataSets: array of TDBDataSet);
```

Un detalle interesante, que nos puede ahorrar código: si la tabla a la cual se aplica el método *ApplyUpdates* se encuentra en alguno de los estados de edición, se llama de forma automática al método *Post* sobre la misma. Esto implica también que *ApplyUpdates* graba, o intenta grabar, las modificaciones pendientes que todavía residen en el *buffer* de registro, antes de confirmar la operación de actualización.

A un nivel más bajo, los conjuntos de datos tienen implementados los métodos *ApplyUpdates* y *CommitUpdates*; la igualdad de nombres entre los métodos de los conjuntos de datos y de las bases de datos puede confundir al programador nuevo en la orientación a objetos. Estos son métodos sin parámetros:

```
procedure TDBDataSet.ApplyUpdates;
procedure TDBDataSet.CommitUpdates;
```

*ApplyUpdates*, cuando se aplica a una tabla o consulta, realiza la primera fase de un protocolo en dos etapas; este método es el encargado de grabar físicamente los cambios de la caché en la base de datos. La segunda fase es responsabilidad de *CommitUpdates*. Este método limpia las actualizaciones ya aplicadas que aún se encuentran en la caché. ¿Por qué necesitamos un protocolo de dos fases? Si realizamos actualizaciones sobre varias tablas, y pretendemos grabarlas atómicamente, tendremos que enfrentarnos a la posibilidad de errores de grabación, ya sean provocados por el control de concurrencia o por restricciones de integridad. Por lo tanto, en el algoritmo de confirmación se han desplazado las operaciones fallibles a la primera fase, las llamadas a *ApplyUpdates*; en cambio, *CommitUpdates* nunca debe fallar, a pesar de Murphy.



El método *ApplyUpdates*, de la clase *TDatabase*, aprovecha la división en dos fases. Para aplicar las actualizaciones pendientes de una lista de tablas, la base de datos inicia una transacción y ejecuta los métodos *ApplyUpdates* individuales de cada conjunto de datos. Si alguno falla no pasa nada, pues la transacción se deshace y los cambios siguen residiendo en la memoria caché. Si la grabación es exitosa en conjunto, se confirma la transacción y se llama sucesivamente a *CommitUpdates* para cada conjunto de datos. El esquema de la implementación de *ApplyUpdates* es el siguiente:

```

StartTransaction;                                // Self = la base de datos
try
  for I := Low(DataSets) to High(DataSets) do
    DataSets[i].ApplyUpdates;                    // Pueden fallar
  Commit;
except
  Rollback;
  raise;                                         // Propagar la excepción
end;
for I := Low(DataSets) to High(DataSets) do
  DataSets[i].CommitUpdates;                    // Nunca fallan

```

Es recomendable usar siempre el *ApplyUpdates* de la base de datos para confirmar las actualizaciones, en vez de utilizar los métodos de los conjuntos de datos, aún en el caso de una sola tabla o consulta.

Por último, una advertencia: como se puede deducir de la implementación del método *ApplyUpdates* aplicable a las bases de datos, las actualizaciones pendientes se graban en el orden en que se pasan las tablas dentro de la lista de tablas. Por lo tanto, si estamos aplicando cambios para tablas en relación *master/detail*, hay que pasar primero la tabla maestra y después la de detalles. De este modo, las filas maestras se graban antes que las filas dependientes. Por ejemplo:

```
Database1.ApplyUpdates([tbPedidos, tbDetalles]);
```

En contraste, no existe un método predefinido que descarte las actualizaciones pendientes en todas las tablas de una base de datos. Para descartar las actualizaciones pendientes en caché, se utiliza el método *CancelUpdates*, aplicable a objetos de tipo *TDBDataSet*. Del mismo modo que *ApplyUpdates* llama automáticamente a *Post*, si el conjunto de datos se encuentra en algún estado de edición, *CancelUpdates* llama implícitamente a *Cancel* antes de descartar los cambios no confirmados.

También se pueden cancelar las actualizaciones para registros individuales. Esto se consigue con el método *RevertRecord*, que devuelve el registro a su estado original.

## El estado de actualización

La función *UpdateStatus*, declarada por *TDataSet*, indica el tipo de la última actualización realizada sobre el registro activo. *UpdateStatus* puede tomar los siguientes valores:

Valor	Significado
<i>usUnmodified</i>	El registro no ha sufrido actualizaciones
<i>usModified</i>	Se han realizado modificaciones sobre el registro
<i>usInserted</i>	Este es un registro nuevo
<i>usDeleted</i>	Este registro ha sido borrado (ver más adelante)

La forma más sencilla de comprobar el funcionamiento de esta propiedad es mostrar una tabla con actualizaciones en caché sobre una rejilla, e interceptar el evento *OnDrawColumnCell* de la rejilla para mostrar de forma diferente cada tipo de registro:

```

procedure TForm1.DBGrid1DrawColumnCell(
    Sender: TObject; const Rect: TRect; DataCol: Integer;
    Column: TColumn; State: TGridDrawState);
begin
    with TDBGrid(Sender), Canvas.Font do begin
        case DataSource.DataSet.UpdateStatus of
            usModified: Style := Style + [fsBold];
            usInserted: Style := Style + [fsItalic];
            usDeleted: Style := Style + [fsStrikeOut];
        end;
        DefaultDrawColumnCell(Rect, DataCol, Column, State);
    end;
end;

```

También puede aprovecharse la función para deshacer algunos de los tipos de cambios en el conjunto de datos:

```

procedure TForm1.DeshacerInsercionesClick(Sender: TObject);
var
    BM: TBookmarkStr;
begin
    Table1.DisableControls;
    BM := Table1.Bookmark;
    try
        Table1.First;
        while not Table1.Eof do
            if Table1.UpdateStatus = usInserted then
                Table1.RevertRecord
            else
                Table1.Next;
        finally
            Table1.Bookmark = BM;
            Table1.EnableControls;
        end;
    end;

```

¿Qué sentido tiene el poder marcar una fila de una rejilla como borrada, si nunca podemos verla? Pues sí se puede ver. Para esto, hay que modificar la propiedad *UpdateRecordTypes* del conjunto de datos en cuestión. Esta propiedad es un conjunto que puede albergar las siguientes constantes:

Valor	Significado
<i>rtModified</i>	Mostrar los registros modificados
<i>rtInserted</i>	Mostrar los nuevos registros

Valor	Significado
<i>rtDeleted</i>	Mostrar los registros eliminados
<i>rtUnModified</i>	Mostrar los registros no modificados

Combinemos ahora el uso de los filtros de tipos de registros con este nuevo método, para recuperar de forma fácil todos los registros borrados cuya eliminación no ha sido aún confirmada:

```

procedure TForm1.bnRecuperarClick(Sender: TObject);
var
    URT: TUpdateRecordTypes;
begin
    URT := Table1.UpdateRecordTypes;
    Table1.UpdateRecordTypes := [rtDeleted];
    try
        Table1.First;
        while not Table1.Eof do
            Table1.RevertRecord;
    finally
        Table1.UpdateRecordTypes := URT;
    end;
end;

```

No hace falta avanzar el cursor hasta el siguiente registro después de recuperar el actual, porque el registro recuperado desaparece automáticamente de la vista del cursor. Para completar el ejemplo, sería necesario restaurar la posición inicial de la tabla, y desactivar temporalmente la visualización de los datos de la misma. Se lo dejo como ejercicio.

## Un ejemplo integral

El siguiente ejemplo integra las distintas posibilidades de las actualizaciones en caché de modo tal que el lector puede verificar el funcionamiento de cada una de ellas. Necesitamos un formulario con una tabla, *Table1*, una fuente de datos *DataSource1*, una rejilla de datos y una barra de navegación. Da lo mismo la tabla y la base de datos que elijamos; lo único que necesitamos es asignar *True* a la propiedad *CachedUpdates* de la tabla.

También nos hará falta un menú. Estos son los comandos que crearemos:

Caché	Ver
Grabar	Originales
Cancelar	Modificados
Cancelar actual	Nuevos
	Borrados

Para hacer más legible el código que viene a continuación, he renombrado coherentemente las opciones de menú; así, la opción *Caché*, cuyo nombre por omisión sería

*Cach1*, se ha transformado en *miCache* (*mi* = *menu item*). En primer lugar, daremos respuesta a los tres comandos del primer submenú:

```

procedure TForm1.miGrabarClick(Sender: TObject);
begin
    if not Table1.Database.IsSQLBased then
        Table1.Database.TransIsolation := tiDirtyRead;
        Table1.Database.ApplyUpdates([Table1]);
end;

procedure TForm1.miCancelarClick(Sender: TObject);
begin
    Table1.CancelUpdates;
end;

procedure TForm1.miCancelarActualClick(Sender: TObject);
begin
    Table1.RevertRecord;
end;

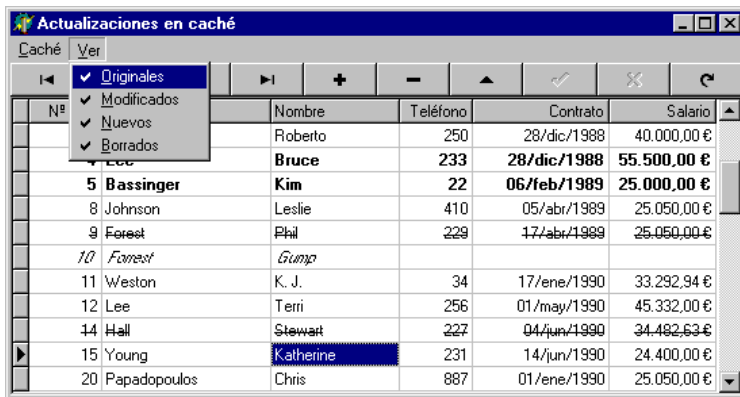
```

La primera línea en el primer método sólo es necesaria si se ha elegido como tabla de prueba una tabla Paradox o dBase. Ahora necesitamos activar y desactivar las opciones de este submenú; esto lo hacemos en respuesta al evento *OnClick* de *miCache*:

```

procedure TForm1.miCacheClick(Sender: TObject);
begin
    miGrabar.Enabled := Table1.UpdatesPending;
    miCancelar.Enabled := Table1.UpdatesPending;
    miCancelarActual.Enabled := Table1.UpdateStatus <> usUnModified;
end;

```



Luego creamos un manejador compartido para los cuatro comandos del menú *Ver*:

```

procedure TForm1.ComandosVer(Sender: TObject);
var
    URT: TUpdateRecordTypes;
begin
    TMenuItem(Sender).Checked := not TMenuItem(Sender).Checked;
    URT := [];
    if miOriginales.Checked then Include(URT, rtUnmodified);
    if miModificados.Checked then Include(URT, rtModified);

```

```

    if miNuevos.Checked      then Include(URT, rtInserted);
    if miBorrados.Checked   then Include(URT, rtDeleted);
    Table1.UpdateRecordTypes := URT;
end;

```

Por último, debemos actualizar las marcas de verificación de estos comandos al desplegar el submenú al que pertenecen:

```

procedure TForm1.miVerClick(Sender: TObject);
var
    URT: TUpdateRecordTypes;
begin
    URT := Table1.UpdateRecordTypes;
    miOriginales.Checked := rtUnmodified in URT;
    miModificados.Checked := rtModified in URT;
    miNuevos.Checked := rtInserted in URT;
    miBorrados.Checked := rtDeleted in URT;
end;

```

Si lo desea, puede incluir el código de personalización de la rejilla de datos, para visualizar el estado de cada registro.

## Cómo actualizar consultas “no” actualizables

Cuando una tabla o consulta ha activado las actualizaciones en caché, la grabación de los cambios almacenados en la memoria caché es responsabilidad del BDE. Normalmente, el algoritmo de actualización es generado automáticamente por el BDE, pero tenemos también la posibilidad de especificar la forma en que las actualizaciones tienen lugar. Esto es especialmente útil cuando estamos trabajando con consultas contra un servidor SQL. Las bases de datos SQL se ajustan casi todas al estándar del 92, en el cual se especifica que una sentencia **select** no es actualizable cuando contiene un encuentro entre tablas, una cláusula **distinct** o **group by**, etc. Por ejemplo, InterBase no permite actualizar la siguiente consulta, que muestra las distintas ciudades en las que viven nuestros clientes:

```

select distinct City
from   Customer

```

No obstante, es fácil diseñar reglas para la actualización de esta consulta. La más sencilla es la relacionada con la modificación del nombre de una ciudad: deberíamos modificar el nombre de la ciudad en todos los registros de clientes que la mencionan. Más polémica es la interpretación de un borrado. Podemos eliminar a todos los clientes de esa ciudad, tras pedirle confirmación al usuario. En cuanto a las inserciones, lo más sensato es prohibirlas, para no vernos en la situación de Alicia, que había visto muchos gatos sin sonrisa, pero nunca una sonrisa sin gato.

Delphi nos permite intervenir en el mecanismo de actualización de conjuntos de datos en caché al ofrecernos el evento *OnUpdateRecord*:

```

type

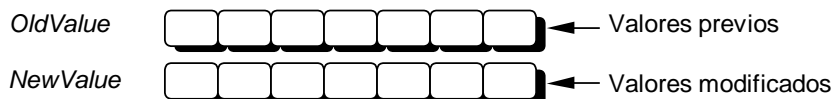
```

```
TUpdateRecordEvent = procedure (  
    DataSet: TDataSet; UpdateKind: TUpdateKind;  
    var UpdateAction: TUpdateAction) of object;
```

El parámetro *DataSet* representa al conjunto de datos que se está actualizando. El segundo parámetro puede tener uno de estos valores: *ukInsert*, *ukModify* ó *ukDelete*, para indicar qué operación se va a efectuar con el registro activo de *DataSet*. El último parámetro debe ser modificado para indicar el resultado del evento. He aquí los posibles valores:

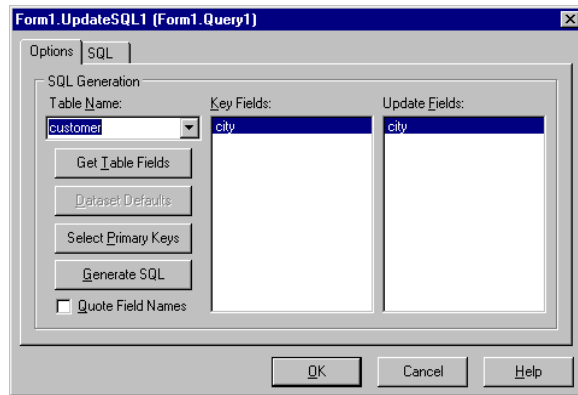
Valor	Significado
<i>uaFail</i>	Anula la aplicación de las actualizaciones, lanzando una excepción
<i>uaAbort</i>	Aborta la operación mediante la excepción silenciosa
<i>uaSkip</i>	Ignora este registro en particular
<i>uaRetry</i>	No se utiliza en este evento
<i>uaApplied</i>	Actualización exitosa

Dentro de la respuesta a este evento, podemos combinar cuantas operaciones de actualización necesitemos, siempre que no cambiemos la fila activa de la tabla que se actualiza. Cuando estamos trabajando con este evento, tenemos acceso a un par de propiedades especiales de los campos: *OldValue* y *NewValue*, que representan el valor del campo antes y después de la operación, al estilo de las variables *new* y *old* de los *triggers* en SQL.



Sin embargo, lo más frecuente es que cada tipo de actualización pueda realizarse mediante una sola sentencia SQL. Para este caso sencillo, las tablas y consultas han previsto una propiedad *UpdateObject*, en la cual se puede asignar un objeto del tipo *TUpdateSQL*. Este componente actúa como depósito para tres instrucciones SQL, almacenadas en las tres propiedades *InsertSQL*, *ModifySQL* y *DeleteSQL*. Primero hay que asignar este objeto a la propiedad *UpdateObject* de la consulta. La operación tiene un efecto secundario inverso: asignar la consulta a la propiedad no publicada *DataSet* del componente *TUpdateSQL*. Sin esta asignación, no funciona ni la generación automática de instrucciones que vamos a ver ahora, ni la posterior sustitución de parámetros en tiempo de ejecución.

Una vez que el componente *TUpdateSQL* está asociado a un conjunto de datos, un doble clic sobre él activa un editor de propiedades, que nos ayuda a generar el código de las tres sentencias SQL. El editor necesita que le especifiquemos cuál tabla queremos actualizar, pues en el caso de un encuentro tendríamos varias. En nuestro caso, se trata de una consulta muy simple, por lo que directamente pulsamos el botón *Generate SQL*, y nos vamos a la página siguiente, para retocar las instrucciones generadas si es necesario. Las instrucciones generadas son las siguientes:



```

delete from Customer          /* Borrado */
where City = :OLD_City

update Customer              /* Modificación */
set   City = :City
where City = :OLD_City

insert into Customer(City)    /* Altas */
values (:City)

```

Las instrucciones generadas utilizan parámetros especiales, con el prefijo *OLD\_*, de modo similar a la variable de contexto *old* de los *triggers* de InterBase y Oracle. Es evidente que la instrucción **insert** va a producir un error, al no suministrar valores para los campos no nulos de la tabla. Pero recuerde que de todos modos no tenía sentido realizar inserciones en la consulta, y que no íbamos a permitirlo.

## El evento *OnUpdateRecord*

Si un conjunto de datos con actualizaciones en caché tiene un objeto enganchado en su propiedad *UpdateObject*, y no se ha definido un manejador para *OnUpdateRecord*, el conjunto de datos utiliza directamente las instrucciones SQL del objeto de actualización para grabar el contenido de su caché. Pero si hemos asignado un manejador para el evento mencionado, se ignora el objeto de actualización, y todas las grabaciones deben efectuarse en la respuesta al evento. Para restaurar el comportamiento original, debemos utilizar el siguiente código:

```

procedure TmodDatos.Query1UpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  UpdateSQL1.DataSet := Query1;
  UpdateSQL1.Apply(UpdateKind);
  UpdateAction := uaApplied;
end;

```

En realidad, la primera asignación del método anterior sobra, pero es indispensable si el objeto de actualización no está asociado directamente a un conjunto de datos me-

diante la propiedad *UpdateObject*, sino que, por el contrario, es independiente. Como observamos, el método *Apply* realiza primero la sustitución de parámetros y luego ejecuta la sentencia que corresponde. La sustitución de parámetros es especial, porque debe tener en cuenta los prefijos *OLD\_*. En cuanto al método *ExecSQL*, está programado del siguiente modo:

```
procedure TUpdateSQL.ExecSQL(UpdateKind: TUpdateKind);
begin
    with Query[UpdateKind] do begin
        Prepare;
        ExecSQL;
        if RowsAffected <> 1 then DatabaseError(SUpdateFailed);
    end;
end;
```

¿Se da cuenta de que no podemos utilizar el comportamiento por omisión para nuestra consulta, aparentemente tan simple? El problema es que *ExecSQL* espera que su ejecución afecte exactamente a una fila, mientras que nosotros queremos que al modificar un nombre de ciudad se puedan modificar potencialmente varios usuarios. No tendremos más solución que programar nuestra propia respuesta al evento:

```
procedure TmodDatos::Query1UpdateRecord(DataSet: TDataSet;
    UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
    if UpdateKind = ukInsert then
        DatabaseError('No se permiten inserciones')
    else if UpdateKind = ukDelete and (MessageDlg('¿Está seguro?',
        mtConfirmation, [mbYes, mbNo], 0) <> mrYes) then begin
        UpdateAction := uaAbort;
        Exit;
    end;
    UpdateSQL1.SetParams(UpdateKind);
    UpdateSQL1.Query[UpdateKind].ExecSQL;
    UpdateAction := uaApplied;
end;
```

En algunas ocasiones, la respuesta al evento *OnUpdateRecord* se basa en ejecutar procedimientos almacenados, sobre todo cuando las actualizaciones implican modificar simultáneamente varias tablas.



## BDE: descenso a los abismos

**A** TODOS LOS DESARROLLADORES NOS ATRAE LA idea de especializarnos en algún área esotérica y confusa de la programación, para poder después presumir de “expertos en algo”. Nadie está completamente libre de esta superstición. Al programador de aplicaciones para bases de datos con Delphi puede parecerle interesante descender en un escalón de abstracción y acceder a las profundas y oscuras entrañas del BDE, buscando más control y eficiencia. ¿Hasta qué punto le será útil este esfuerzo?

Hace un par de años, le hubiera recomendado ciegamente que aprendiera al menos los detalles básicos del manejo del API del BDE. Pero a estas alturas no lo tengo tan claro: las sucesivas versiones de la VCL han terminado por encapsular la mayoría de las tareas interesantes del Motor de Datos. Las principales áreas que quedan por cubrir en la VCL son:

- Algunas funciones de lectura y actualización de parámetros de configuración.
- Creación y reestructuración de tablas de Paradox y dBase.
- Algunas posibilidades de ciertas funciones de respuesta.

Estas técnicas nos servirán de excusa para explicar cómo se programa directamente con el Motor de Datos de Borland.

### Inicialización y finalización del BDE

Lo primero es saber cómo tener acceso a las funciones del API del BDE. Todas las declaraciones del Motor de Datos se encuentran en la unidad *BDE*. Esta es una *unidad de importación*, que se limita a declarar las funciones exportadas por las DLLs del Motor de Datos. Aunque Delphi no nos da el código fuente de la unidad *BDE*, sí nos ofrece la parte correspondiente a la interfaz en el fichero *bde.int*, que se encuentra en el subdirectorio *doc* de Delphi.

Después de incluir la unidad *BDE* en alguna de las cláusulas **uses**, hay que aprender a iniciar y descargar de memoria las DLLs del BDE. Cuando trabajamos con la VCL, estas tareas son realizadas automáticamente por la clase *TSession*. Ahora debemos utilizar explícitamente la siguiente función:

```

type
  pDBIEnv = ^DBIEnv;
  DBIEnv = packed record
    szWorkDir      : DBIPATH;
    szIniFile      : DBIPATH;
    bForceLocalInit : WordBool;
    szLang         : DBINAME;
    szClientName   : DBINAME;
    { Los tipos DBIPATH y DBINAME son vectores de caracteres }
  end;

function DbiInit(pEnv: PDbiEnv): DBIResult; stdcall;

```

Es importante conocer que todas las funciones del BDE devuelven un valor entero para indicar si pudieron cumplir o no con su cometido. En la siguiente sección veremos más detalles acerca del control de errores.

La forma más sencilla de llamar a *DbiInit* es pasando el puntero nulo como parámetro, en cuyo caso se inicializa el BDE con valores por omisión:

```

// Inicializar el BDE con parámetros por omisión
DbiInit(nil);

```

Si utilizamos un puntero a una variable de entorno (*DBIEnv*) podemos indicar:

Campo	Significado
<i>szWorkDir</i>	El directorio de trabajo.
<i>szIniFile</i>	El fichero de configuración.
<i>bForceLocalInit</i>	Obliga a una inicialización local.
<i>szLang</i>	El idioma a utilizar.
<i>szClientName</i>	Una identificación para la aplicación.

Son tres los posibles valores de retorno de *DbiInit*:

Valor de retorno	Significado
<i>DBIERR_NONE</i>	Hakuna matata
<i>DBIERR_MULTIPLEINIT</i>	El BDE ya había sido inicializado
<i>DBIERR_OSACCESS</i>	No se puede escribir sobre el directorio actual

Si la función devuelve *DBIERR\_MULTIPLEINIT*, no tiene por qué preocuparse, pues de todos modos puede utilizar las funciones del BDE. Esto es precisamente lo que sucedería si inicializáramos manualmente el BDE en una aplicación de Delphi que utilizase los componentes de acceso a datos de la VCL: la inicialización de la VCL fallaría con el error mencionado, pero no se consideraría un error serio.

Para finalizar el trabajo con el BDE debemos llamar a la siguiente función:

```

function DbiExit: DBIResult; stdcall;

```

Según la documentación del BDE, si lo inicializamos desde una DLL estamos obligados a llamar la función *DbiDllExit* antes de llamar a *DbiExit*.

## El control de errores

Como hemos visto, las funciones del BDE utilizan códigos de error para señalar fallos de contrato. Un lenguaje moderno, sin embargo, debe utilizar funciones para este propósito. Por lo tanto, la VCL utiliza la función *Check*, definida en la unidad *DBTables*, para verificar el código de retorno de las funciones del Motor de Datos, y lanzar una excepción si detecta que algo va mal:

```
Check(DbiInit(nil));
```

*Check* lanza una excepción de clase *EDBEngineError*. Ya hemos explicado, en el capítulo 27, la estructura de esta clase de excepción, y vimos que informa acerca de una pila de errores: el error producido por el servidor SQL (de ser aplicable), la interpretación del BDE, etc. Siempre que pueda, utilice *Check* para el control de errores.

No obstante, puede ser que escribamos en algún momento una pequeña aplicación y no deseemos cargar con nada declarado por la VCL para mantener una huella pequeña en memoria. ¿Cómo extrae *Check* la información compleja que le sirve para construir el objeto *EDBEngineError*? Estas son las funciones del BDE para el manejo de errores:

```
function DbiGetErrorString(rslt: DBIResult;
    pszError: PChar): DBIResult; stdcall;
function DbiGetErrorContext(eContext: SmallInt;
    pszContext: PChar): DBIResult; stdcall;
function DbiGetErrorInfo(bFull: Bool;
    var ErrInfo: DBIErrInfo): DBIResult; stdcall;
function DbiGetErrorEntry(uEntry: Word; var ulNativeError: LongInt;
    pszError: PChar): DBIResult; stdcall;
```

La más sencilla es *DbiGetErrorString*, que traduce un código de error a su representación textual. Si necesita un sustituto barato de *Check*, puede utilizar la siguiente función:

```
procedure CheapCheck(Rslt: DBIResult);
var
    Buffer: array [0.. DBIMAXMSGLEN] of Char;
begin
    if Rslt <> DBIERR_NONE then
    begin
        DbiGetErrorString(Rslt, Buffer);
        raise Exception.Create(Buffer);
    end;
end;
```

Las restantes funciones deben llamarse inmediatamente después de producirse el error, porque la información que extraen siempre se refiere al último error. Es muy interesante, por ejemplo, la función *DbiGetErrorContext*, pues nos permite averiguar

determinadas circunstancias acerca de ese último error. El parámetro *eContext* indica si queremos conocer qué tabla, qué índice, qué usuario... está involucrado en el fallo:

```
var
  Buffer: array [0..DBIMAXMSGLEN] of Char;
begin
  // ...
  DbiGetErrorContext(ecINDEXNAME, Buffer);
  ShowMessage('Indice: ' + StrPas(Buffer));
  // ...
end;
```

Sin embargo, no podemos aprovechar las habilidades de esta función en un manejador del evento *OnPostError*, pues al llegar a ese punto la VCL ya ha eliminado la información de contexto del error generado.

## Sesiones y conexiones a bases de datos

Como sabemos, para aliviar la vida del programador, el BDE reconoce que no es frecuente trabajar con varias sesiones simultáneas, e implementa una sesión activa. Existe una sesión por omisión, que el BDE crea y administra por su cuenta, pero podemos crear sesiones adicionales y activarlas cuando lo consideremos oportuno:

```
function DbiStartSession(pszName: PChar; var hSes: hDBISes;
  pNetDir: PChar): DBIResult; stdcall;
function DbiCloseSession(hSes: hDBISes): DBIResult; stdcall;
function DbiGetCurrSession(var hSes: hDBISes): DBIResult; stdcall;
function DbiSetCurrSession(hSes: hDBISes): DBIResult; stdcall;
```

Si seguimos adelante con la jerarquía de objetos del BDE, tropezamos de narices con las conexiones a bases de datos: el equivalente aproximado de la *TDatabase* de nuestra VCL. La función que abre una base de datos es la siguiente:

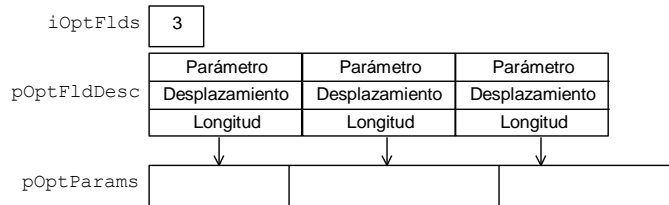
```
function DbiOpenDatabase(pszDbName, pszDbType: PChar;
  eOpenMode: DBIOpenMode; eShareMode: DBIShareMode;
  pszPassword: PChar; iOptFlds: Word; pOptFldDesc: pFLDDesc;
  pOptParams: Pointer; var hDb: hDBIDb): DBIResult; stdcall;
```

Es típico del BDE utilizar un montón de parámetros incluso en las funciones más elementales, aunque para la mayoría de los casos baste el uso de valores por omisión. Por tal motivo, es preferible demostrar con ejemplos el uso de este tipo de funciones. Para abrir una base de datos de Paradox o dBase, y asociarla a un determinado directorio, utilice una función como la siguiente:

```
function AbrirDirectorio(const Directorio: string): hDBIDb
begin
  CheapCheck(DbiOpenDatabase(nil, nil, dbiREADWRITE, dbiOPENSERIALIZED,
    nil, 0, nil, nil, Result));
  CheapCheck(DbiSetDirectory(Result, PChar(Directorio)));
end;
```

La base de datos, claro está, se crea en el contexto definido por la sesión activa. Recuerde que *CheapCheck* ha sido definido por nosotros, para evitar las referencias a la VCL.

Si hay que crear una conexión no persistente a una base de datos SQL con parámetros explícitos, se complica el uso de *DbiOpenDatabase*. El siguiente esquema muestra la forma de suministrar dichos valores, mediante los parámetros *iOptFlds*, *pOptFldDesc* y *pOptParams*:



El tipo *FLDDesc* sirve en realidad para describir un campo, y volveremos a encontrarlo al presentar la creación de tablas. En esta función, sirve para describir un parámetro, y sólo nos interesan tres de sus campos: *szName*, para el nombre del parámetro de conexión, *iOffset*, para especificar el desplazamiento dentro del *buffer* apuntado por *pOptParams* donde se encuentra el valor, e *iLen*, para indicar la longitud del valor dentro del *buffer*. El siguiente ejemplo muestra cómo abrir una base de datos SQL especificando el alias, el usuario y su contraseña:

```
function AbrirSQL(const AliasName, UserName,
  Password: string): hDBIDb;
var
  Params: FLDDesc;
begin
  FillChar(Params, 0, SizeOf(Params));
  StrCopy(Params.szName, 'USER NAME');
  Params.iOffset := 0;
  Params.iLen := Length(UserName) + 1;
  CheapCheck(DbiOpenDatabase(PChar(AliasName), nil, dbiREADWRITE,
    dbiOPENSERIALIZED, PChar(Password), 1, @Params,
    PChar(UserName), Result));
end;
```

Por supuesto, si necesitamos hacer una llamada aislada al BDE dentro de una aplicación que ya utiliza la VCL, es mejor abrir la base de datos mediante un componente *TDatabase* y utilizar su propiedad *Handle* para obtener el identificador de la base de datos utilizado por el BDE.

## Creación de tablas

Una de las áreas del BDE que puede ser interesante dominar a bajo nivel es la creación de tablas para dBase y Paradox. Si se trata de la creación de tablas para sistemas SQL, no deberíamos ni siquiera plantearnos la posibilidad de crearlas directamente

mediante el BDE; en tal caso, la técnica más potente y segura es enviar instrucciones SQL mediante objetos *TQuery*. Cuando se trata de las bases de datos de escritorio, la VCL hace un trabajo meritorio de encapsulación con el método *CreateTable*, pero se deja muchas características en el tintero:

- 1 Definición de máximos y mínimos, y valores por omisión para columnas de tablas en Paradox.
- 2 Definición de relaciones de integridad referencial para Paradox y dBase 7.
- 3 En el formato de dBase 7, la VCL no crea correctamente los índices únicos y las claves primarias.
- 4 Parámetros físicos de tablas, como el lenguaje de la tabla y el tamaño de bloque de Paradox.

Las técnicas de creación de tablas se basan en la función *DbiCreateTable*:

```
function DbiCreateTable(hDb: hDBIDb; bOverWrite: Bool;
    var crTblDesc: CRTblDesc): DBIResult; stdcall;
```

El primer parámetro contiene el *handle* a una conexión de base de datos, el segundo especifica qué debe suceder si ya existe una tabla con ese nombre, mientras que en el tercero se pasa por referencia una gigantesca estructura que contiene todos los detalles necesarios para crear la tabla. La declaración de *CRTblDesc* es como sigue:

```
type
    pCRTblDesc = ^CRTblDesc;
    CRTblDesc = packed record
        szTblName:      DBITBLNAME;      // 261 caracteres
        szTblType:      DBINAME;          // 32 caracteres
        szErrTblName:   DBIPATH;          // 261 caracteres
        szUserName:     DBINAME;          // 32 caracteres
        szPassword:     DBINAME;          // 32 caracteres
        bProtected:     WordBool;         // Sólo si es Paradox
        bPack:          WordBool;
        iFldCount:      Word;              // Descriptores de campos
        pcrFldOp:        pCROpType;
        pfldDesc:        pFLDDesc;
        iIdxCount:      Word;              // Descriptores de índices
        pcrIdxOp:        pCROpType;
        pidxDesc:        pIDXDesc;
        iSecRecCount:   Word;              // Descriptores de seguridad
        pcrSecOp:        pCROpType;
        psecDesc:        pSECDesc;
        iValChkCount:   Word;              // Restricciones y validaciones
        pcrValChkOp:    pCROpType;
        pvchkDesc:      pVCHKDesc;
        iRintCount:     Word;              // Integridad referencial
        pcrRintOp:       pCROpType;
        printDesc:       pRINTDesc;
        iOptParams:     Word;              // Parámetros opcionales
        pfldOptParams:  pFLDDesc;
        pOptData:       Pointer;
    end; // Un poco largo, ¿verdad?
```

No se asuste, amigo, que no es necesario rellenar todos los campos de la estructura anterior. Para empezar, todos los atributos de tipo puntero a *CROPType* se emplean solamente durante la reestructuración de una tabla. En segundo lugar, observe que hay muchos pares de parámetros dependientes entre sí, como *iFldCount*, que contiene el número de campos a crear, y *pFldDesc*, que apunta a un vector con las descripciones de estos campos.

Ya nos hemos tropezado con el tipo *FLDDesc*, que sirve para la descripción de campos. Ahora presentaré su definición completa:

```

type
  pFLDDesc = ^FLDDesc;
  FLDDesc = packed record
    iFldNum: Word;           // Número secuencial del campo
    szName: DBINAME;        // Nombre del campo (32 bytes)
    iFldType: Word;          // Tipo del campo
    iSubType: Word;          // Subtipo
    iUnits1: SmallInt;       // Tamaño, o precisión
    iUnits2: SmallInt;       // Escala (decimales)
    iOffset, iLen, iNullOffset: Word;
    efldvVchk: FLDVchk;      // ¿Tiene validaciones?
    efldrRights: FLDRights;   // Derechos sobre el campo
    bCalcField: WordBool;     // ¿Es calculado?
    iUnused: packed array [0..1] of Word;
  end;

```

Hay que aclarar que la mayoría de los atributos, a partir de *iOffset*, no se utilizan con *DbiCreateTable*, sino que sirven para recuperar información sobre campos de tablas existentes.

Otros tipos importantes son *IDXDsc*, el descriptor de índices, y *VCHKDsc*, que se utiliza para las restricciones de integridad a nivel de campo: campos requeridos, mínimos, máximos y valores por omisión. Pero voy a ahorrarle la declaración de ambos. Curiosamente, con las últimas versiones del BDE no se puede crear una restricción de integridad referencial a la vez que se crea la tabla, sino que hay que añadir estas restricciones con una llamada a *DbiDoRestructure*, la función de modificación de esquemas que veremos en la siguiente sección. De este modo, se simplifica aún más la creación de tablas.

En vez de largar una parrafada acerca de cada uno de los atributos de las estructuras anteriores, es preferible un pequeño ejemplo que muestre el uso típico de las mismas. Cuando queremos crear una tabla y conocemos perfectamente la estructura de la misma en tiempo de diseño, es frecuente que los descriptors de campos, índices y de validaciones se almacenen en vectores estáticos, como se muestra en el siguiente fragmento de código:

```

// Descriptores de campos
FLDDesc fields[] = {
  {1, "Codigo", fldDBLONG, fldUNKNOWN, 0, 0},
  {2, "Nombre", fldDBCHAR, fldUNKNOWN, 30, 0},
  {3, "Alta", fldDBDATETIME, fldUNKNOWN, 0, 0}};

```

```

// Validaciones
VCHKDesc checks[] = {
    {2, TRUE, FALSE, FALSE, FALSE},
    {3, FALSE, FALSE, FALSE, TODAYVAL}};

// Descriptores de índices
IDXDesc indexes[] = {
    // Primario, Unico
    {"", 1, "Primario", "", 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, {0},
     "CODIGO"},
    // Secundario, Unico
    {"", 2, "ByName", "", 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, {0},
     "NOMBRE"}};

procedure TForm1.Button1Click(Sender: TObject);
var
    TableDesc: CRTblDesc;
begin
    FillChar(TableDesc, SizeOf(TableDesc), 0);
    StrCopy(TableDesc.szTblName, 'Clientes.DBF');
    StrCopy(TableDesc.szTblType, szDBASE);
    TableDesc.iFldCount := 3;
    TableDesc.pfldDesc := @fields[0];
    TableDesc.iIdxCount := 2;
    TableDesc.pidxDesc := @indexes[0];
    TableDesc.iValChkCount := 2;
    TableDesc.pvchkDesc := @checks[0];
    Check(DbiCreateTable(Database1.Handle, 1, TableDesc));
end;

```

El método anterior crea una tabla en formato dBase, con tres campos, de tipos entero, carácter y fecha/hora, respectivamente. El segundo campo es no nulo, mientras que el tercero utiliza la fecha actual como valor por omisión. Observe que el primer atributo de cada *VCHKDesc*, de nombre *iFldNum*, se refiere al número de secuencia de determinado campo, que se indica en el atributo homónimo de la estructura *FLDDesc*. También se crea una clave primaria sobre la primera columna, y una clave secundaria sobre la segunda.

## Reestructuración

La otra función que utiliza la estructura *CRTTblDesc* es *DbiDoRestructure*, y sirve para modificar la definición de una tabla:

```

function DbiDoRestructure(hDb: hDBIDb; iTblDescCount: Word;
    pTblDesc: pCRTblDesc; pszSaveAs, pszKeyviolName: PChar;
    pszProblemsName: PChar; bAnalyzeOnly: Bool): DBIResult; stdcall;

```

En teoría, esta función puede reestructurar simultáneamente varias tablas en una misma base de datos, cuyo *handle* se pasa en *hDb*. En la práctica, el segundo parámetro debe ser siempre 1, para indicar que sólo se va a modificar una tabla. En el parámetro *pTblDesc* se pasa la descripción de las operaciones a efectuar sobre la pobre tabla. Podemos también cambiar el nombre a la tabla, guardar información sobre las violaciones de la clave primaria en otra tabla y quedarnos con una copia de los regis-



tros con problemas. El último parámetro, *bAnalyzeOnly*, no funciona y tiene toda la pinta de que nunca llegará a funcionar.

Para mostrar el uso de *DbiDoRestructure* implementaremos una de las restricciones que no pueden crearse mediante métodos de alto nivel con Delphi: las restricciones de integridad referencial de dBase. Pero primero necesitamos una tabla de detalles. En la sección anterior habíamos creado una tabla de clientes genérica; ahora supondremos que estos clientes realizan consultas técnicas, y que necesitamos una tabla *Consultas* para almacenarlas. Esta segunda tabla debe tener un campo *Cliente*, en el cual guardaremos el código del cliente que hace la consulta:

```
// Creación de una tabla de detalles

const
  Fields1: array [0..4] of FLDDesc = (
    (1, "Codigo", fldDBAUTOINC, fldUNKNOWN, 0, 0),
    (2, "Cliente", fldDBLONG, fldUNKNOWN, 0, 0),
    (3, "Fecha", fldDBDATETIME, fldUNKNOWN, 0, 0),
    (4, "Asunto", fldDBCHAR, fldUNKNOWN, 30, 0),
    (5, "Texto", fldDBMEMO, fldUNKNOWN, 0, 0));

  VCHKDesc checks1[] = {
    {2, TRUE, FALSE, FALSE, FALSE},
    {3, TRUE, FALSE, FALSE, FALSE},
    {4, FALSE, FALSE, FALSE, TODAYVAL}};

  IDXDesc indexes1[] = {
    // Primario, Unico
    {"", 1, "Primario", "", 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, {0},
     "CODIGO"},
    // Secundario
    {"", 2, "Cliente", "", 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, {0},
     "CLIENTE"}};

procedure TForm1.Button2Click(Sender: TObject);
var
  TableDesc: CRTblDesc;
begin
  FillChar(TableDesc, 0, SizeOf(TableDesc));
  StrCopy(TableDesc.szTblName, 'Consultas.DBF');
  StrCopy(TableDesc.szTblType, szDBASE);
  TableDesc.iFldCount := 5;
  TableDesc.pfldDesc := @fields1[0];
  TableDesc.iIdxCount := 2;
  TableDesc.pidxDesc := @indexes1[0];
  TableDesc.iValChkCount := 3;
  TableDesc.pvchkDesc := @checks1[0];
  Check(DbiCreateTable(Database1.Handle, 1, TableDesc));
end;
```

Una vez que está creada la tabla de detalles, podemos añadirle entonces la restricción de integridad:

```
// Añadir la restricción de integridad referencial

const
  Operations: array [0..0] of CROpType =
```

```

        (crADD);
    rInts: array [0..0] of RINTDesc =
        ((1, 'RefCliente', rintDEPENDENT, 'Clientes.DBF',
            rintRESTRICT, rintRESTRICT, 1, (2), (1)));

procedure TForm1.Button3Click(Sender: TObject);
var
    TableDesc: CRTblDesc;
begin
    FillChar(TableDesc, 0, SizeOf(TableDesc));
    StrCopy(TableDesc.szTblName, 'Consultas.DBF');
    StrCopy(TableDesc.szTblType, szDBASE);
    TableDesc.pcrRintOp := @Operations[0];
    TableDesc.iRintCount := 1;
    TableDesc.printDesc := rInts;
    Check(DbiDoRestructure(Database1.Handle, 1, @TableDesc,
        nil, nil, nil, False));
end;

```

Observe que debemos rellenar el atributo *pcrRintOp* del descriptor de tablas. En general podemos añadir, eliminar o modificar simultáneamente varias restricciones, aunque aquí solamente añadimos una sola.

## Eliminación física de registros borrados

Una aplicación muy especial de *DbiDoRestructure* es la eliminación física de registros borrados en Paradox y dBase, y la reconstrucción de sus índices. En tal caso, basta con indicar el nombre de la tabla y asignar *True* al atributo *bPack* del descriptor de operaciones:

```

procedure PackParadox(const ADatabase, ATable: string;
var
    PdxStruct: CRTblDesc;
    ADB: TDatabase;
begin
    ADB := Session.OpenDatabase(ADatabase);
    try
        FillChar(PdxStruct, 0, SizeOf(PdxStruct));
        StrCopy(PdxStruct.szTblName, PChar(ATable));
        PdxStruct.bPack := True;
        Check(DbiDoRestructure(ADB.Handle, 1, @PdxStruct,
            nil, nil, nil, False));
    finally
        Session.CloseDatabase(ADB);
    end;
end;

```

Pero existe una función, *DbiPackTable*, que puede utilizarse específicamente para dBase. Esta vez, la función necesita que el cursor de la tabla esté abierto:

```

procedure PackDBF(const ADatabase, ATable: string);
begin
    with TTable.Create(nil) do
        try
            DatabaseName := ADatabase;
            TableName := ATable;

```

```

        Exclusive := True;
    Open;
    Check(DbiPackTable(Database.Handle, Handle, '', '', True));
finally
    Free;
end;
end;
end;

```

También existe la función *DbiRegenIndexes*, que reconstruye los índices de tablas de Paradox y dBase, sin afectar a los registros eliminados lógicamente. Al igual que sucede con *DbiPackTable*, es necesario que la tabla esté abierta en modo exclusivo:

```

procedure Reindexar(const ADatabase, ATable: string);
begin
    with TTable.Create(nil) do
        try
            DatabaseName := ADatabase;
            TableName := ATable;
            Exclusive := True;
            Open;
            Check(DbiRegenIndexes(Handle));
        finally
            Free;
        end;
    end;
end;

```

## Cursores

Una vez que tenemos acceso a una base de datos, contamos con todo lo necesario para abrir una tabla o una consulta para recorrer sus filas. Para representar un conjunto de datos abierto, o *cursor*, se utiliza el tipo de datos *hDBICur*, que contiene el *handle* del cursor. Existen muchos tipos de cursores, y por lo tanto, muchas funciones del BDE que abren un cursor. Las más “tradicionales”, que sirven para abrir tablas y consultas, son las siguientes:

Función	Propósito
<i>DbiOpenTable</i>	Abre un cursor para una tabla
<i>DbiOpenInMemTable</i>	Crea y abre una tabla en memoria
<i>DbiQExecDirect</i>	Ejecuta una consulta sin parámetros
<i>DbiQExec</i>	Ejecuta una consulta previamente preparada.

Pero también existen funciones que abren tablas “virtuales”, que realmente no están almacenadas como tales, pero que sirven para devolver con el formato de un conjunto de datos información de longitud variable. Si quiere conocer cuáles son todas estas funciones, vaya al índice de la ayuda en línea del API del BDE, y teclee *Dbi-OpenList functions*, como tema de búsqueda.

¿Tiene sentido entrar a fondo en el estudio de las funciones de apertura y navegación sobre cursores? ¡No, absolutamente! La VCL encapsula bastante bien esta funcionalidad mediante las clases *TBDEDataSet* y *TDBDataSet*, como para tener que repetir

esta labor. Se trata, además, de funciones complejas, como puede deducirse de la declaración de *DbiOpenTable* (¡12 parámetros!):

```
function DbiOpenTable(hDb: hDBIDb;
  pszTableName, pszDriverType, pszIndexName: PChar;
  pszIndexTagName: PChar; iIndexId: Word; eOpenMode: DBIOpenMode;
  eShareMode: DBIShareMode; exlMode: XLTMode;
  bUniDirectional: Bool; pOptParams: Pointer;
  var hCursor: hDBICur): DBIResult; stdcall;
```

En cualquier caso, si necesita utilizar alguno de los cursores especiales, es preferible derivar un nuevo componente a partir de *TBDEDataSet*, si no necesita una conexión a una base de datos, o de *TDBDataSet*, en caso contrario. Supongamos que queremos obtener en formato tabular las restricciones de integridad referencial asociadas a una tabla. El BDE nos ofrece la función *DbiOpenRintList* para abrir un cursor con dicha información:

```
function DbiOpenRintList(hDb: hDBIDb; pszTableName: PChar;
  pszDriverType: PChar; var hChkCur: hDBICur): DBIResult; stdcall;
```

Pero nosotros, gente inteligente, creamos la siguiente clase:

```
type
  TRITable = class (TDBDataSet)
  private
    FTableName: string;
  protected
    function CreateHandle: hDBICur; override;
  published
    property TableName: string read FTableName write FTableName;
  end;
```

La única función redefinida tiene una implementación trivial:

```
function TRITable.CreateHandle: hDBICur
begin
  Check(DbiOpenRintList(DBHandle, PChar(FTableName), nil, Result));
end;
```

La siguiente imagen demuestra el funcionamiento del componente anterior dentro de una sencilla aplicación, incluida en el CD:

REFINTNUM	NAME	TYPE	OTHTABLE	MODIFYQUAL	DELETEQUAL
1	CLASSIQUE.SYS_C001266	1	CLASSIQUE.TAXES		
	CLASSIQUE.SYS_C001284		CLASSIQUE.ATTRIBUTES		
	CLASSIQUE.SYS_C001322		CLASSIQUE.PRODUCTS		
	CLASSIQUE.SYS_C001415		CLASSIQUE.CLASSWORDS		

He incluido la imagen para mostrar cómo, gracias a la maquinaria interna del tipo *TDBDataSet*, no tenemos que preocuparnos por la definición de los campos; estas definiciones se extraen automáticamente de las propiedades del cursor abierto.

## Un ejemplo de iteración

De todos modos, es provechoso mostrar al menos un ejemplo básico de iteración sobre un cursor. Ahora echaremos mano de la siguiente función:

```
function DbOpenCfgInfoList(hCfg: hDBICfg;
    eOpenMode: DBIOpenMode; eConfigMode: CFGMode; pszCfgPath: PChar;
    var hCur: hDBICur): DBIResult; stdcall;
```

El primer parámetro siempre debe ser el puntero nulo, mientras que el tercero también debe llevar el valor obligado *cfgPersistent*. En el segundo parámetro indicamos si queremos abrir este cursor para lectura o también para escritura. En el cuarto se pasa una cadena con un formato de directorio. El BDE almacena su información de configuración en forma jerárquica, y este parámetro se refiere a un nodo determinado de la jerarquía. Cuando abrimos un cursor para un nodo, obtenemos una colección de registros con el formato (*parámetro, valor*).

Con esta información a nuestro alcance, es fácil diseñar una función que, dada una de estas rutas y el nombre de un parámetro, devuelva el valor asociado:

```
function GetBDEInfo(const Path, Param: string): string;
var
    hCur: hDBICur;
    Desc: CFGDesc;
begin
    DbOpenCfgInfoList(nil, dbiReadOnly, cfgPersistent,
        PChar(Path), @hCur);
    try
        while DbGetNextRecord(hCur, dbiNoLock, @Desc, 0) =
            DBIERR_NONE do
            if (strcmp(Desc.szNodeName, Param.c_str()) == 0)
                return AnsiString(Desc.szValue);
    finally
        DbCloseCursor(&hCur);
    end;
    Result := '<Not found>';
end;
```

El núcleo de la función es la iteración sobre el cursor mediante la función:

```
function DbGetNextRecord(hCursor: hDBICur; eLock: DBILockType;
    pRecBuff: Pointer; precProps: pRECProps): DBIResult; stdcall;
```

En su tercer parámetro debemos pasar un puntero al *buffer* que recibe los datos. El formato de este *buffer* depende del tipo de cursor abierto, en particular, de las columnas que posea, pero para cada cursor especial del BDE ya existe un registro que puede recibir las filas del cursor. En el caso de *DbOpenCfgInfoList*, el tipo apropiado

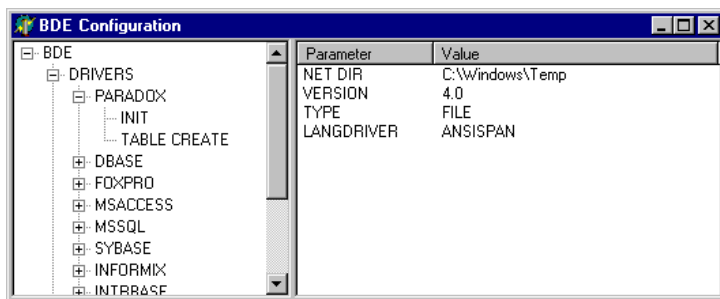
es la estructura *CFGDesc*. De todos modos, hay también un método de *TSession* que hace más o menos lo mismo:

```
procedure TSession.GetConfigParams(const Path, Section: string;
  List: TStrings);
```

Pero con la función del BDE tenemos la posibilidad de modificar cualquiera de sus parámetros, si utilizamos esta variante del algoritmo:

```
procedure SetBDEInfo(const Path, Param, Value: string);
var
  hCur: hDBICur;
  Desc: CFGDesc;
begin
  DbOpenCfgInfoList(nil, dbiReadWrite, cfgPersistent,
    PChar(Path), hCur);
  try
    while DbGetNextRecord(hCur, dbiNoLock, @Desc, nil) =
      DBIERR_NONE do
      if CompareText(Desc.szNodeName, Param) = 0 then
        begin
          StrPCopy(Desc.szValue, Value);
          DbModifyRecord(hCur, @Desc, True);
          break;
        end;
    end;
  finally
    DbCloseCursor(hCur);
  end;
end;
```

En el CD-ROM que acompaña al libro hemos incluido una pequeña aplicación que sirve para explorar la estructura arbórea de los parámetros de configuración del BDE. También se pueden cambiar los valores de los parámetros:



Las siguientes funciones especiales aprovechan los procedimientos anteriores para lograr de una forma más sencilla la modificación de dos de los parámetros del BDE más populares:

```
procedure SetLocalShare(const Value: string);
begin
  SetBDEInfo('\SYSTEM\INIT', 'LOCAL SHARE', Value);
end;
```

```

procedure SetNetDir(const Value: string);
begin
    SetBDEInfo('\DRIVERS\PARADOX\INIT', 'NET DIR', Value);
end;

```

## Propiedades

En el ejemplo anterior hemos leído los datos del registro activo dentro de un *buffer* de longitud fija, cuya estructura es suministrada por el BDE. ¿Y cómo hacemos en el caso general para averiguar la longitud necesaria del *buffer* de un cursor arbitrario? La respuesta es: leyendo las propiedades del cursor.

En realidad, casi todos los objetos del BDE tienen propiedades que pueden leerse y, en ocasiones, ser modificadas. Las propiedades del BDE son un mecanismo primitivo de asociar atributos a objetos cuya verdadera estructura interna está fuera del alcance del programador. Las funciones que manipulan propiedades son:

```

function DbiGetProp(hObj: hDBIObj;
    iProp: LongInt; PropValue: Pointer;
    iMaxLen: Word; var iLen: Word): DBIResult; stdcall;
function DbiSetProp(hObj: hDBIObj;
    iProp: LongInt; iPropValue: LongInt): DBIResult; stdcall;

```

El objeto se indica en el primer parámetro, pero el más importante de los parámetros es *iProp*, que identifica qué propiedad queremos leer o escribir. Cada tipo diferente de objeto tiene su propio conjunto de propiedades válidas. Busque *objects*, *properties* en la ayuda en línea del BDE para un listado exhaustivo de todas las posibles propiedades. El siguiente ejemplo muestra cómo recuperar el nombre de una tabla asociada a un cursor:

```

var
    iLen: Word;
    tblName: DBITBLNAME;
begin
    DbiGetProp(hDBIObj(TBDEDataSet(DataSet).Handle),
        curTABLENAME, @tblName, SizeOf(tblName), iLen);
    ShowMessage(tblName);
end;

```

Cuando el objeto es precisamente un cursor, las propiedades más interesantes pueden obtenerse agrupadas mediante la función *DbiGetCursorProps*:

```

function DbiGetCursorProps(hCursor: hDBICur;
    var curProps: CURProps): DBIResult; stdcall;

```

La estructura *CURProps* tiene 36 atributos, incluyendo el inevitable *iUnused*, así que no le voy a aburrir contándole para qué sirve cada uno. Solamente le diré que el tamaño del *buffer* para el registro se puede obtener por medio del atributo *iRecBufSize*. En definitiva, esto era lo que queríamos averiguar al principio de esta sección, ¿no?

Para terminar, quiero mostrarle alguna técnica con propiedades que *realmente* merezca la pena. Uno de los principales trucos con los cursores del BDE tiene que ver con la posibilidad de limitar el número máximo de filas que éste puede retornar al cliente. Cuando estudiamos la configuración del Motor de Datos vimos el parámetro *MAX ROWS*, común a la mayoría de los controladores SQL. Si modificamos su valor, afectaremos a todas las tablas y consultas que se abran por medio del alias modificado. Sin embargo, podemos hacer que la limitación solamente se aplique a un cursor determinado.

```

procedure LimitRows(DataSet: TDBDataSet; MaxRows: Integer);
var
    Len: Word;
    DBType: array [0..DBIMAXNAMELEN - 1] of Char;
begin
    Check(DbiGetProp(hDBIObj(DataSet.DBHandle), dbDATABASETYPE,
        DBType, SizeOf(DBType), Len));
    if CompareText(DBType, 'STANDARD') = 0 then
        raise EDBEngineError.Create(DBIERR_NOTSUPPORTED);
    Check(DbiValidateProp(hDBIObj(DataSet.Handle), curMAXROWS,
        True));
    Check(DbiSetProp(hDBIObj(DataSet.Handle), curMAXROWS, MaxRows));
end;

```

Aunque la propiedad *curMAXROWS* puede aplicarse también a una tabla, la limitación se aplica a cada una de las consultas que ésta abre tras el telón, por lo cual el comportamiento de la tabla parecerá errático al usuario. Es mejor entonces no utilizar esta opción con las tablas.

## Las funciones de respuesta del BDE

La última de las técnicas que exploraremos en este capítulo es el uso de funciones de respuestas del BDE. El API del Motor de Datos nos permite registrar algunas funciones de nuestra aplicación para que sean llamadas por el propio Motor durante algunas operaciones. De registrar la función de respuesta se encarga la siguiente rutina:

```

function DbiRegisterCallBack(hCursor: hDBICur;
    ecbType: CbType; iClientData: Longint; iCbBufLen: Word;
    CbBuf: Pointer; pfCb: pfDBICallBack): DBIResult; stdcall;

```

El parámetro principal es *ecbType*, que indica qué operación deseamos controlar. Aunque hay unos cuantos más, he aquí algunos de sus posibles valores:

- *cbTABLECHANGED*: Es la que más fantasías despierta en el programador; vanas fantasías, precisaría yo. La función de respuesta se dispara cuando la instancia activa detecta un cambio en los datos de una tabla de Paradox. Esta es su principal limitación: no puede utilizarse con dBase, Access ni con servidores SQL.



- *cbCANCELQRY*: Seguimos con las funciones fantásticas. En este caso, podemos detener la ejecución de una consulta... pero solamente si el servidor es Sybase.
- *cbDELAYEDUPD*: Menciono esta constante para que el lector vea cómo la VCL ya intercepta algunas de las funciones de respuesta. En este caso, se trata de la función que dispara el evento *OnUpdateError* cuando hay actualizaciones en caché.
- *cbRESTRUCTURE*: ¿Ha visto los mensajes de advertencia que lanza Database Desktop cuando modificamos la estructura de una tabla? La función que se registra mediante esta constante es la responsable de esos mensajes.
- *cbGENPROGRESS*: Notifica periódicamente acerca del progreso de operaciones largas, en particular, de los movimientos de registros masivos mediante *DbiBatchMove* y, por extensión, del componente *TBatchMove*.

En cualquier caso, el prototipo de la función de respuesta debe corresponder al siguiente:

```
type
  pfDBICallBack = function (ecbType: CbType; iClientData: Longint;
    CbInfo: Pointer): CbType; stdcall;
```

Los parámetros *CbBuf* y *iCbBufLen* de *DbiRegisterCallBack* deben suministrar un *buffer* cuya estructura depende del tipo de función de respuesta. Cuando el tipo de función es *cbGENPROGRESS*, el tipo del *buffer* debe ser el siguiente:

```
type
  pCBPROGRESSDesc = ^CBPROGRESSDesc;
  CBPROGRESSDesc = packed record
    iPercentDone: SmallInt;
    szMsg: DBIMSG;
  end;
```

- 1 Si *iPercentDone* es igual o mayor que cero, en este atributo se encuentra el porcentaje terminado de la operación.
- 2 En caso contrario, *szMsg* contiene una cadena de caracteres indicando la fase de la operación y un valor numérico que indica el progreso realizado.

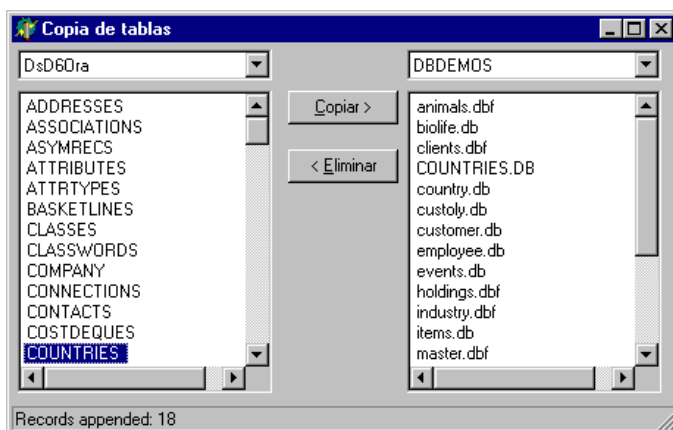
Es fácil entonces definir una función de respuesta de acuerdo a las reglas anteriores:

```
function ProgressCB(ecbType: CbType; ClientData: Integer;
  buf: Pointer): CbType; stdcall;
var
  Msg: string;
begin
  if pCBPROGRESSDesc(buf).iPercentDone = -1 then
    Msg := pCBPROGRESSDesc(buf).szMsg
  else
    Msg := IntToStr(pCBPROGRESSDesc(buf).iPercentDone) + '%';
    wndMain.StatusBar1.SimpleText := Msg;
    Result := cbrCONTINUE;
  end;
```

También podíamos haber devuelto *cbr:ABORT*, para detener la copia de la tabla. Para simplificar, la función anterior se registra y se elimina del registro localmente, dentro una función auxiliar de la aplicación, a la cual hemos llamado *DoBatchMove*:

```
procedure TwndMain.DoBatchMove;
var
    prgDesc: CBPROGRESSDesc;
begin
    Check(DbiRegisterCallBack(nil, cbGENPROGRESS,
        0, SizeOf(prgDesc), @prgDesc, ProgressCB));
    try
        BatchMove1.Execute;
    finally
        Check(DbiRegisterCallBack(nil, cbGENPROGRESS,
            0, 0, nil, nil));
    end;
end;
```

La siguiente imagen corresponde a una de las aplicaciones incluidas en el CD, que registra una función de respuesta para la operación *DbiBatchMove*, para informarnos sobre el progreso de la misma:



## InterBase Express

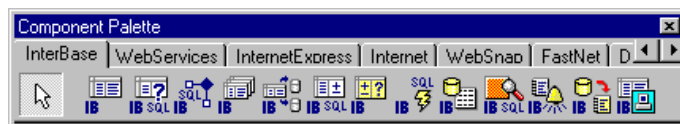
**S**I ME OBLIGÁIS A SER SINCERO, TENDRÉ que reconocer que me ha costado mucho decidirme a escribir el presente capítulo. He ido postergando el momento de comenzar su redacción hasta que casi no me ha quedado tiempo para hacerlo. ¿Motivo? A pesar de todo el tiempo de existencia de InterBase Express y de su antecesor, Free IB Components, siguen existiendo errores garrafales en el código del producto, y la documentación y ejemplos es provocarían un ataque de angustia al mismísimo Sigmund Freud. O quizás de depresión.

Por lo pronto, adminístrese una buena dosis de Prozac, respire profundo y prepárese para sufrir. Como consuelo, piense en que al aprender InterBase Express estará purificando su karma y asegurándose una feliz reencarnación...

### Historia del sufrimiento humano

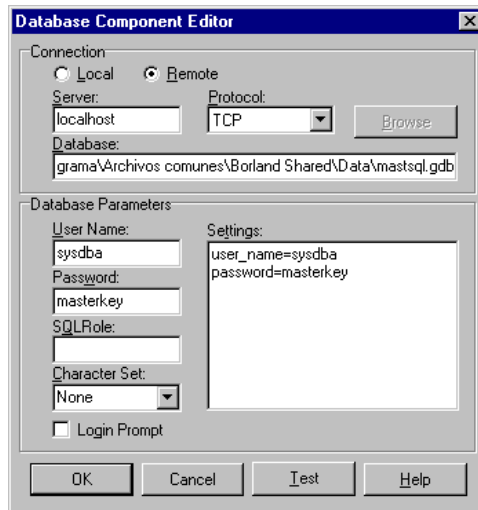
InterBase Express es un conjunto de componentes de acceso a InterBase que acompaña a Delphi desde su versión 5. Está basado en una colección de componentes gratuitos, llamada *Free IB Components*, y desarrollada en solitario por un programador que trabajaba para un despacho de abogados. Su meta principal: deslumbrar con sus tiempos de acceso, algo que garantizaba el uso de la interfaz nativa de InterBase sin intermediario alguno. El producto estaba pensado para desarrollar aplicaciones tradicionales en dos capas, por lo que se añadió una caché de registros, para la navegación bidireccional, y un sistema aparentemente sencillo de actualizaciones sobre el conjunto de datos, basado en la misma caché.

InterBase Express está formado por dos grupos de componentes. El primero se encuentra en la página *InterBase*, en Delphi 6:



Esta otra imagen corresponde también a Delphi 6, pero después de instalar el *Server Media Kit* de InterBase 6.5. Verá que hay cinco nuevos componentes.





Aunque hay varios controles en el grupo *Connection*, se trata sólo de ayudas para componer una cadena de conexión al servidor, que al final se asigna en la propiedad *DatabaseName* del componente. Por ejemplo, el nombre del servidor que se está editando en la imagen corresponderá finalmente al siguiente valor:

```
localhost:c:\Archivos de programa\Archivos comunes\
Borland Shared\Data\MastSQL.GDB
```

Los restantes valores, como ya es tradicional, van a parar a una propiedad llamada *Params*, de tipo *TStrings*; esto incluye al nombre del usuario, su contraseña, el perfil que asumirá y el conjunto de caracteres. Tenemos también una propiedad *LoginPrompt*, para saber si queremos pedir o no el nombre del usuario y su contraseña al establecer la conexión, y una propiedad *Connected* que hace lo que se supone que debe hacer.

## NO ESTAN DOCUMENTADOS

Además de los parámetros básicos que nos ayuda a configurar el editor del componente, las conexiones de InterBase admiten unas cuantas decenas adicionales de parámetros... que no han sido correctamente documentados. Por ejemplo, existe un parámetro *num\_buffers*, pero no se nos dice qué sucede cuando dos clientes se conectan a la misma base de datos e indican valores discordantes. O un interesante *encrypt\_key*, que sugiere la opción de codificar el contenido de una base de datos, pero que en ningún lugar queda claro que no esté actualmente soportado por InterBase. De todos modos, hay otros parámetros con los que parece fácil experimentar, como *dummy\_packet\_interval*: el tiempo que puede estar "callado" el cliente antes de que el servidor se "preocupe" y empiece a tocarle las narices con mensajes del tipo "hola, ¿sigues ahí?". En el *API Guide* hay una lista de estos parámetros, aunque con descripciones muy cortas, pero también se pueden buscar en el código fuente de la unidad *IBDatabase*.

Hay varias propiedades que son únicas en *TIBDatabase*. La primera de ellas se llama *AllowStreamedConnect*, y si vale *True*, se ignora el valor de *Connected* en tiempo de di-

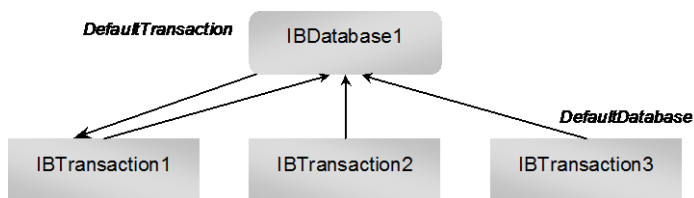
seño. Esto sirve para evitar la apertura automática de la conexión en tiempo de ejecución, si queremos establecer los parámetros dinámicamente.

Hay también una propiedad entera, *IdleTimer*, que contiene un intervalo en milisegundos; la ayuda en línea, por supuesto, no dice si se trata de horas, siglos o eones, pero un vistazo al código fuente confirma mis sospechas. Como dicen en Borland, jusa la fuente, Luke! Lamentablemente, no se nos explica cuándo se activa el temporizador. Tengo el presentimiento que se activa mecánicamente cuando se asigna un valor positivo en *IdleTimer*. En otras palabras, creo que no vale para nada útil.

## Transacciones como amebas

La mayor rareza de InterBase Express, cuando se compara con otras interfaces de acceso a datos, es la forma en que se tratan las transacciones. Por una parte, pueden existir varias transacciones activas e independientes dentro de una misma conexión. Es el efecto que se lograría en otros sistemas de bases de datos teniendo varias conexiones simultáneas; el sistema de InterBase, en cambio, ahorra recursos. Una misma transacción, además, puede englobar a varias bases de datos a la vez. Algunos llaman *transacciones distribuidas* a esta técnica.

Las propiedades que enlazan conexiones, transacciones y conjuntos de datos son algo confusas. La clase *TIBDatabase* tiene propiedades *TransactionCount* y *Transactions*, esta última vectorial, que son las que importan. Pero tiene también una propiedad *DefaultTransaction*, que por lo que he podido ver no vale para mucho; bueno, cuando uno asigna un conjunto de datos a una base de datos y se le olvida asignar también su transacción, se utiliza la transacción por omisión de la base de datos. También hay una propiedad *DefaultDatabase* en la clase *TIBTransaction*, y esto sí tiene sentido, porque normalmente una transacción se ocupará de una sola base de datos:



Pero *TIBTransaction* soporta los métodos *AddDatabase* y *RemoveDatabase*, para que la transacción proteja simultáneamente más de una base de datos. Y se puede consultar la lista de bases de datos de una transacción mediante las propiedades *DatabaseCount* y *Databases*.

Al igual que las conexiones, las transacciones tienen un temporizador interno, y su intervalo de activación se configura en la propiedad *IdleTimer*. Al transcurrir sin jaleo el número de milisegundos especificados *IdleTimer*, se consulta el valor de la propiedad *DefaultAction*, que pertenece al siguiente enumerativo:

```

type
  TTransactionAction = (TARollback, TACommit,
    TARollbackRetaining, TACommitRetaining);

```

Este sistema no me gusta en absoluto: un programador de verdad siempre sabe cuándo iniciar una transacción. Y cómo terminarla:

```

procedure TIBDatabase.Commit;
procedure TIBDatabase.Rollback;
procedure TIBDatabase.CommitRetaining;
procedure TIBDatabase.RollbackRetaining;

```

Acepto incluso que existan métodos separados para *CommitRetaining* y *RollbackRetaining*, aunque me hubiera gustado más tener una propiedad en plan *RetainCursors* en el componente. Es lo mismo que con la memoria: los programadores *de verdad* sabemos liberar toda la memoria que pedimos... ¿qué dice?... ¡buah, Java no cuenta!

### RETENCION DE CURSORES

Recuerde que SQL estándar exige que se invaliden los cursores abiertos cada vez que se termina una transacción. Las variantes de confirmación y cancelación con retención evitan que InterBase cierre los cursores activos al terminar la transacción.

La empanada cerebral no termina aquí: *TIBTransaction* tiene una propiedad *Active*. Estupendo, dirá usted: le asignamos *True* y se inicia una transacción, ¿verdad? Note, por cierto, que el método de acceso de la implementación de *Active* es nada menos que *InTransaction*. Es decir, que nos sobra *Active* porque ya existe *InTransaction*. Es cierto, listillo, que podemos asignar *False* a *Active*, pero ¿qué demonios se supone que tiene que pasar entonces? Yo hubiese apostado a que se consultaba el valor en *DefaultAction*, pero no: siempre se llama a *Rollback*, olvidándose del *retaining*.

¿Quiere más? Los componentes de conexión de DB Express, ADO Express y del BDE tienen una propiedad *KeepConnections*. Si su valor es *False*, la conexión se cierra cuando se cierra el último conjunto de datos asociado a ella. Pero las cosas no iban a ser tan sencillas en IB Express: hay una propiedad *AutoStopAction*, con estos valores:

```

type
  TAutoStopAction = (saNone, saRollback, saCommit,
    saRollbackRetaining, saCommitRetaining);

```

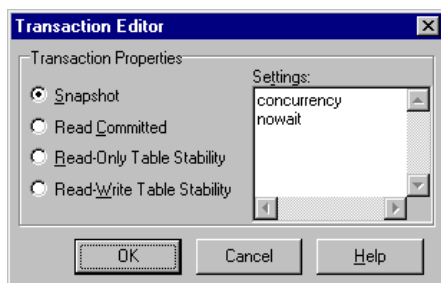
Claro, el diseñador podría haber puesto *KeepConnections* en *TIBTransaction*, y cuando su valor fuese *False*, utilizar la misma propiedad *DefaultAction* que hemos visto antes. ¿Sería que le pagaban por líneas de código?

### ADVERTENCIA

Tenga muchísimo cuidado con la interpretación de *IdleTimer*. No es un evento que se dispare una vez y se desconecte hasta que vuelvan a realizarse cambios. Es un evento que se dispara *periódicamente*. Es decir, que es prácticamente inútil.

## Parámetros de transacciones

Al igual que sucede con los componentes de conexión, las transacciones tienen su conjunto particular de parámetros, y nuevamente la documentación es escasa. Hay dos formas de editarlos: entrando a saco en la propiedad *Params*, o haciendo doble clic sobre el componente para usar un editor.



El valor por omisión es *Snapshot*: lecturas repetibles, y el fallo es inmediato cuando se encuentra un registro bloqueado. No es un mal valor por omisión, pero veamos de todos modos las restantes posibilidades. Es mejor que agrupemos los parámetros por grupos. Tenemos, por ejemplo, las combinaciones que indican el nivel de aislamiento:

- 1 *concurrency*: Este es el nivel serializable, que aprovecha las versiones de registros para lograr lecturas repetibles. Es el nivel por omisión.
- 2 *consistency*: Otro nivel serializable, pero a costa de bloqueos sobre tablas.
- 3 *read\_committed*, *rec\_version*: Permite leer siempre la versión confirmada más reciente de un registro. Se viola el requisito de las lecturas repetibles.
- 4 *read\_committed*, *no\_rec\_version*: Similar al anterior, pero si se intenta leer un registro y ha sido modificado pero no confirmado, se produce un conflicto.

Hay dos formas de actuar cuando se detecta un conflicto de bloqueos:

- 1 *wait*: La transacción espera a que los recursos vuelvan a estar disponibles. Es el modo activo por omisión.
- 2 *nowait*: La transacción retorna con un error inmediatamente.

Por supuesto, se puede indicar el modo de acceso deseado:

- 1 *read*: Sólo lectura.
- 2 *write*: Lectura y escritura. Es además, el valor por omisión.

Por último, tenemos los parámetros más interesantes, que permiten bloquear el acceso a tablas específicas de la base de datos. Hay dos grupos esta vez, y el primero admite las cadenas *shared* ó *protected*. El segundo grupo admite *lock\_read* y *lock\_write*, aunque la novedad es que debemos incluir un nombre de tabla a continuación de estos últimos parámetros. Por ejemplo:

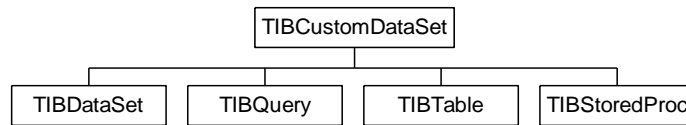


```
protected
lock_write=EMPLOYEE
protected
lock_read=ORDERS
```

Toda esta información, desgraciadamente, hay que deducirla mezclando los fragmentos de sabiduría dispersos entre el *API Guide* y el código fuente de IB Express.

## Conjuntos de datos

Esta es la jerarquía de conjuntos de datos de InterBase Express:



Como ve, es muy sencilla: hay una clase base que implementa gran parte del código necesario, y a partir de ella se derivan casos especiales: cuando el conjunto de datos que queremos contiene todos los registros de una tabla, *TIBTable*, o cuando queremos indicar una sentencia SQL, *TIBQuery*. En este contexto, *TIBStoredProc* es la oveja negra: sí, se utiliza para acceder a un procedimiento almacenado, pero no puede usarse para recuperar las filas de un procedimiento almacenado de selección, con los métodos y propiedades de un conjunto de datos. Es un error que *TIBStoredProc* herede toda la funcionalidad de un conjunto de datos, aunque supongo que se habrá tomado la decisión por inercia, siguiendo el diseño del BDE.

### NOTA

Al igual que con el BDE, para recibir el conjunto de filas de un procedimiento almacenado de selección de InterBase se puede utilizar un *TIBQuery*, en el que la sentencia SQL de consulta llame al procedimiento almacenado desde su cláusula **from**.

La documentación también es confusa cuando tiene que explicar la existencia del componente *TIBDataSet*. A efectos prácticos, es muy parecido a *TIBQuery*. ¿La diferencia? Para actualizar un *TIBQuery* es necesario utilizar un componente adicional, *TIBUpdateSQL*. En cambio, *TIBDataSet* ya trae incorporadas las propiedades necesarias para este tipo de actualizaciones.

Para confundir aún más al programador novato, el diseñador de IB Express decidió poner en la Paleta de Componentes a *TIBSQL*. Este es un descendiente directo de *TComponent* que se utiliza internamente para llamar a las funciones de recuperación de cursores a través de una capa de encapsulación muy fina. Es verdad que se trata de un componente que permite traer datos al cliente con la mayor celeridad, pero también es cierto que su interfaz es tan críptica como el propio API de InterBase. En las últimas versiones de IBX han comenzado a añadirle más funciones, para justificar un poco más su existencia pública.

Quitando a *TIBSQL*, los restantes componentes tienen muchas propiedades en común: hay que asociarlos a una base de datos, a través de la propiedad *Database*, y a una transacción, a través de *Transaction*. Todos ellos tienen una propiedad *Unidirectional*, para que los registros no se almacenen en una caché local y sea más rápida la recuperación de datos. Y hay también una propiedad *CachedUpdates*; cuando vale *False* los registros siguen trayéndose a una caché de navegación, pero si llamamos a *Post* estando modificada la fila activa, se genera inmediatamente una sentencia de actualización que se envía al servidor. Si *CachedUpdates* es *True*, en cambio, el conjunto de datos funciona de forma similar a los conjuntos de datos del BDE en modo de actualizaciones en caché.

### ADVERTENCIA

El *Developer's Guide* de InterBase advierte muy claramente que, si desea utilizar IBX con conjuntos de datos clientes, que es la única forma sensata de arriesgarse a usarlo, debe utilizar un componente de transacción por cada conjunto de datos que configure. Sinceramente, me parece exagerado. Puede que el escritor del manual tenga problemas con las transacciones que se cierran “espontáneamente”. En cualquier caso, experimente un poco antes de lanzarse a desarrollar algo con esta interfaz de acceso.

## La caché de InterBase Express

Tenemos también una propiedad llamada *BufferChunks*. En mi versión de InterBase Express, que es la que viene con InterBase 6.5, vale 1.000 por omisión. Este es el tamaño inicial de la caché, expresado como número de registros. Pero también es el tamaño en que crece la caché cuando se desborda.

Este es uno de los puntos más conflictivos de la implementación de InterBase Express. El diseñador original decidió utilizar una caché lineal: un trozo de memoria consecutiva para almacenar los registros uno a continuación del otro. Por algún motivo que no acabo de comprender, la mayoría de los errores históricos de IBX han sido culpa del código de manejo de la caché.

Pero no se trata sólo de mala suerte. Usar una caché lineal es algo muy ineficiente, en comparación con otras técnicas igualmente sencillas. Si quiere desternillarse un poco, eche un vistazo a la siguiente “perla”, que se ocupa de la ampliación de la caché:

```
procedure IBAlloc(var P; OldSize, NewSize: Integer);
var
  i: Integer;
begin
  ReallocMem(Pointer(P), NewSize);
  for i := OldSize to NewSize - 1 do
    PChar(P)[i] := #0;
  end;
```

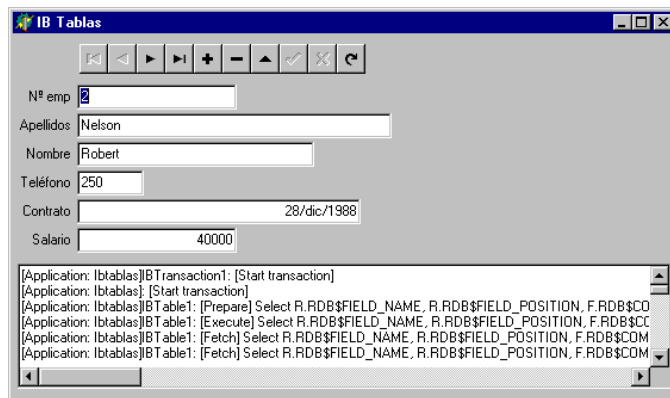
Esto es lo que en buen cristiano se llama “código chapucero”. La inicialización con ceros es digna de un programador de Java. Admito que quizás el tiempo desperdi-

ciado en inicializar cada byte del buffer por separado sea insignificante cuando se sitúa en el contexto de una aplicación final. Pero no habría sido ni más difícil ni menos claro sustituir ese bucle con una llamada a *FillChar*, sobre todo teniendo en cuenta que esa llamada tarda en ejecutarse la décima parte o menos del tiempo consumido dentro del bucle, según mis mediciones.

## El Monitor de IB Express

Cada vez que se descubre a Microsoft ocultando la existencia de una interfaz de programación en alguno de sus productos, las voces de protesta llegan a la bóveda celeste. Con toda justicia. Un sistema operativo, un lenguaje de programación o una base de datos son todos ejemplos de *herramientas*. Admito que alguien me venda un televisor y que selle el aparato, para que sólo lo abra el servicio técnico. Pero, ¿imagina lo que vendría a su cabeza si descubriese un pequeño botón disimulado en el mango de un martillo, que al presionarlo lo transformase en un arma blanca? Vale, me he pasado un poco, pero espero que capte la idea...

Sin embargo, cuando Borland hace lo mismo... supongo que esas mismas voces están ya afónicas de tanto chillar, y sólo es por este motivo que no se arma el alboroto. ¿No me cree? Le narraré entonces el misterioso caso del puntero que nunca existió. Vamos a crear una nueva aplicación; en el CD viene con el nombre de *IBTablas*. No utilizaremos módulos de datos para simplificar. El aspecto final de la aplicación será el siguiente:



Traiga a la ventana un *TIBDatabase*, un *TIBTransaction* y un *TIBTable*. Conecte estos componentes a cualquier tabla en cualquier base de datos de InterBase; yo he recurrido a la clásica tabla de empleados. Configure los campos y cree los controles visuales como en la imagen anterior. ¿Los detalles de la configuración? Da lo mismo; es más, es aconsejable que experimente por sí mismo. Ahora verá por qué...

...es que vamos a utilizar un componente *TIBSQLMonitor*, para espiar qué instrucciones envía la tabla al servidor. Debe traer el componente, que ya vendrá con *Enabled* a

*True* por omisión. Debe seleccionar entonces la propiedad *TraceFlags* de la base de datos, y activar todas las opciones que le interesen. Observe que hay una propiedad del mismo nombre en el monitor, pero ahora necesitamos la de la conexión.

Y ahora viene lo interesante, porque el monitor nos avisa de la llegada de un mensaje mediante un evento:

```
type
  TSQLException = procedure (
    EventText: string; EventTime: TDateTime) of object;

property TIBSQLMonitor.OnSQLException: TSQLException;
```

Pero ese evento se dispara desde un hilo que no es el hilo principal del proceso. Si desde ese hilo paralelo intenta alguna operación sobre los controles visuales, puede tener problemas. En realidad, *ya* he tenido problemas. ¿Qué podemos hacer? Muy sencillo: copiar esos mensajes en alguna estructura de datos internas, y avisar a la ventana dejando un mensaje en su cola de mensajes. De esta forma, el mensaje será procesado por el hilo principal; aprovecharemos entonces para pasar los mensajes pendientes al *TMemo*. ¿Mensajes, en plural? Sí, porque lo normal será que el hilo del monitor produzca más mensajes que lo que pueda consumir la ventana durante el tratamiento del mensaje.

He declarado, en consecuencia, dos variables globales en la unidad:

```
var
  CS: TCriticalSection;
  Mensajes: TStrings;
```

El tipo *TCriticalSection* está declarado en la unidad *SyncObjs*. Las variables se inicializan y destruyen en el código de inicialización y finalización de la unidad principal:

```
initialization
  CS := TCriticalSection.Create;
  Mensajes := TStringList.Create;
finalization
  FreeAndNil(Mensajes);
  FreeAndNil(CS);
end.
```

También he tenido que inventar una constante para el mensaje:

```
const
  WM_MENSAJE = WM_USER + 1234;
```

La respuesta al evento *OnSQLException* del componente monitor es la siguiente:

```
procedure TwndPrincipal.IBSQLMonitor1SQL(EventText: string;
  EventTime: TDateTime);
begin
  CS.Enter;
```

```

try
    Mensajes.Add(StringReplace(
        EventText, #13#10, '', [rfReplaceAll]));
finally
    CS.Leave;
end;
PostMessage(Handle, WM_MENSAJE, 0, 0);
end;

```

Hay que filtrar los cambios de líneas que nos intenta colar el monitor dentro de los mensajes; por eso es que llamo a *StringReplace*. Observe que el acceso a la variable con los mensajes se protege con la sección crítica. Es muy probable que intentemos leer los mensajes desde el hilo principal mientras el monitor añade mensajes nuevos.

Finalmente, hay que declarar un método de manejo de mensajes en la ventana:

```

procedure WMMensaje(var Msg); message WM_MENSAJE;

```

Su implementación es la siguiente:

```

procedure TwndPrincipal.WMMensaje(var Msg);
begin
    CS.Enter;
    try
        Memol.Lines.AddStrings(Mensajes);
        Mensajes.Clear;
    finally
        CS.Leave;
    end;
end;

```

Y ya estamos listos para practicarle la autopsia a InterBase Express.

## El as bajo la manga, o crónicas del juego sucio

Ejecute la aplicación, y verá que sólo con iniciarse, aparecen decenas de instrucciones en el cuadro de edición. Esto significa que la tabla de IBX debe ejecutar un prólogo de lectura del esquema tan largo como el de la injustamente denostada tabla del BDE. Pasemos de puntillas delante de los detalles, y busque la penúltima instrucción, la que realmente abre el cursor con los datos de la tabla de empleados:

```

select EMPLOYEE.*, RDB$DB_KEY as IBX_INTERNAL_DBKEY
from EMPLOYEE
order by EMPNO

```

¿Qué es ese misterioso valor *rdB\$db\_key*? Pues un recurso no documentado de InterBase. Se trata de una pseudocolumna que devuelve un valor de 8 caracteres de longitud. Ese valor actúa más o menos como un identificador de fila para la tabla o consulta base, más o menos como un *rowid* de Oracle. La diferencia consiste en que la clave de fila de InterBase no es un invariante del registro, y sólo tiene validez, por omisión, mientras dure la transacción en la que fue obtenida. No obstante, se puede

extender su tiempo de vida al tiempo que dure la conexión añadiendo al componente de conexión el siguiente parámetro:

```
dbkey_scope=1
```

¿Para qué utiliza InterBase Express la clave de registro? En las primeras versiones, las tablas se implementaban leyendo de golpe todas las claves de registro en memoria. A partir de ese momento, navegar a una fila consistía solamente en recuperar el registro que correspondía a esa clave. Ahora, sin embargo, las filas de las tablas se leen según se necesiten, con todos sus valores. Por lo tanto, no hay ninguna ventaja en rendimiento respecto a las consultas, como sí sabemos que ocurre en el BDE. De todos modos, este truco permite localizar filas para su actualización con mayor fiabilidad. Por ejemplo, el borrado de registros en una *TIBTable* se realiza con una instrucción como la siguiente:

```
delete from EMPLOYEE
where RDB$DB_KEY = :IBX_INTERNAL_DBKEY
```

## Consultas en InterBase Express

Como ya hemos dicho, *TIBDataSet* y *TIBQuery* son muy similares. En el primero, hay que teclear una sentencia **select** en su propiedad *SelectSQL*, mientras que la consulta se teclea en la propiedad *SQL*, a secas, del segundo componente. *TIBQuery* maneja, por omisión, un componente de sólo lectura. Pero si le asociamos un *TIBUpdateSQL* a través de su propiedad *UpdateSQL*, podemos realizar modificaciones con él. La responsabilidad del *TIBUpdateSQL* sería la de proporcionar las instrucciones necesarias para cada tipo de actualización. Por su parte, *TIBDataSet* viene de la fábrica con un *TIBUpdateSQL* incorporado. Es decir, tiene cuatro propiedades en las que se pueden teclear instrucciones de modificación:

```
property TIBDataSet.DeleteSQL: TStrings;
property TIBDataSet.InsertSQL: TStrings;
property TIBDataSet.ModifySQL: TStrings;
property TIBDataSet.RefreshSQL: TStrings;
```

Si no configuramos las propiedades anteriores, *TIBDataSet* manejará también un conjunto de datos de sólo lectura.

Para hacer pruebas, copie el ejemplo anterior en un nuevo directorio, y cámbiele el nombre. Sustituya entonces la tabla por un componente *TIBDataSet*. Asócielo al componente de conexión y a la transacción, y teclee la siguiente instrucción dentro de su propiedad *SelectSQL*:

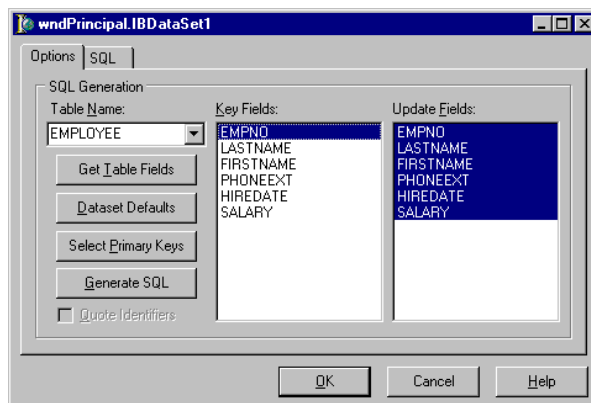
```
select *
from EMPLOYEE
order by EmpNo asc
```

Cree los componentes de acceso a campos y compruebe que ya puede navegar por el conjunto de datos, pero no actualizarlo. Observe también en el monitor que ya no es necesario leer toda la información de esquemas. Esto significa que es mucho más eficiente abrir una consulta o un *TIBDataSet* que una tabla de InterBase Express.

### RECOMENDACIÓN

Para trabajar con DataSnap o conjuntos de datos clientes, en general, se recomienda utilizar un componente *TIBQuery* mejor que un *TIBDataSet*. Así no tenemos que cargar con el sistema de actualización de este último componente, que no será utilizado, de todos modos, por DataSnap.

Para poder actualizar el conjunto de datos, pulse el botón derecho de ratón sobre él, y ejecute el comando *Dataset Editor*.



Como ve, el editor es muy similar al del *UpdateSQL* del BDE, y también al de IBX. Sólo tenemos que indicar qué campos forman parte de la clave primaria y pulsar el botón *Generate SQL*, para que se creen automáticamente las cuatro instrucciones necesarias para añadir, modificar, borrar o recuperar la versión más reciente de un registro.

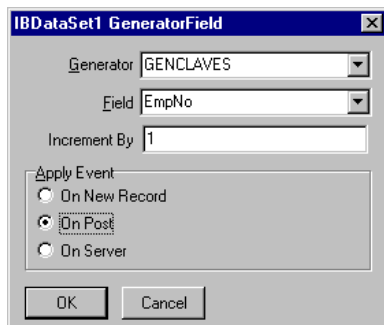
En la misma línea de razonamiento, las últimas versiones de IB Express incluyen una propiedad llamada *GeneratorField* en los componente *TIBDataSet* y *TIBQuery*:

```
property TIBCustomDataSet.GeneratorField: TIBGeneratorField;
```

La clase *TIBGeneratorField* descende de *TPersistent*, y contiene a su vez las siguientes propiedades:

Propiedad	Valor
<i>Field</i>	Nombre del campo que recibirá valores desde un generador
<i>Generator</i>	El nombre del generador
<i>IncrementBy</i>	Incremento o decremento
<i>ApplyEvent</i>	El evento en que se asignará el valor al campo

Pero, por algún motivo que ignoro, el inventor del asunto decidió que no tocásemos directamente esas propiedades en tiempo de diseño, y para modificarlas hay que ejecutar el editor de la propiedad:



Son dos los eventos que pueden utilizarse para asignar un valor proveniente de un generador: *OnNewRecord* y *BeforePost*, que aparece como *On Post* en el editor. El tercer valor, *On Server*, significa en realidad que el valor se asignará en el servidor, mediante un *trigger*.

Por último, existe un modo de actualizaciones en caché para los conjuntos de datos de IB Express, que se activa con la propiedad *CachedUpdates*. Es muy similar a la técnica que estudiamos en el BDE, incluso en el nombre de los métodos. La mayor diferencia es que no hace falta que este modo esté activo para que la consulta pueda utilizar un *TIBUpdateSQL*. Pero el evento *OnUpdateRecord*, parecido también a su homónimo del BDE, sigue necesitando que la caché esté activa para ser disparado:

```
procedure TwndPrincipal.IBDataSet1UpdateRecord(
  DataSet: TDataSet; UpdateKind: TUpdateKind;
  var UpdateAction: TIBUpdateAction);
begin
  // ...
end;
```

#### NOTA

No voy a insistir en el uso de esta técnica. Ya sabe que mi consejo es no utilizar IB Express directamente para las actualizaciones, sino mezclarlo con el uso de conjuntos de datos clientes, o directamente con DataSnap.

## Ejecución directa de comandos

InterBase Express se distingue del resto de las interfaces de acceso a datos soportadas por Delphi en que su componente de conexión no puede ejecutar directamente una instrucción SQL, sino que necesita el apoyo de un componente auxiliar. No hay una instrucción *Execute* que nos eche una mano.

Ayuda no nos faltará. Como debe imaginar, si se trata de un procedimiento almacenado, es preferible que utilicemos un *TIBStoredProc*, sobre todo por el control de



parámetros. Si es una instrucción SQL, ya sea del DDL, DCL o DML, podemos utilizar un *TIBQuery*:

```
with TIBQuery.Create(nil) do
try
  Database := IBDatabase1;
  // Asigna también automáticamente la transacción por omisión
  SQL.Text := 'create generator GenID';
  Transaction.StartTransaction;
  try
    ExecSQL;
    Transaction.Commit;
  except
    Transaction.Rollback;
    raise;
  end;
finally
  Free;
end;
```

El fragmento de código anterior es un poco arriesgado. Deberíamos comprobar antes si la transacción está activa antes de intentar iniciar una nueva.

Pero podemos hacer lo mismo con el componente *TIBSQL*, un espécimen más raro que un ornitorrinco con menopausia. Borland nos lo vende como una forma aún más directa de ejecutar instrucciones SQL. Eso es cierto para instrucciones *select*, pero tengo mis dudas cuando se trata de cualquier otro tipo de sentencia. En cualquier caso, solamente tiene que cambiar la llamada a *ExecSQL* por *ExecQuery* para adaptar el fragmento de código anterior a *TIBSQL*.

*TIBSQL* tiene una característica única: implementa dos métodos para la importación y exportación de ficheros de texto, en varios formatos:

```
procedure TIBSQL.BatchInput(InputObject: TIBBatchInput);
procedure TIBSQL.BatchOutput(OutputObject: TIBBatchOutput);
```

Las clases *TIBBatchInput* y *TIBBatchOutput* son clases abstractas que se utilizan para derivar clases de formato concretas. En este momento existen clases para el formato de texto delimitado, para el formato *raw*, que es similar al que utilizan las tablas externas de InterBase, e incluso para exportar, aunque no importar, a XML.

He incluido una pequeña aplicación en el CD para demostrar la exportación de datos con *TIBSQL*. El código que nos interesa es el que se dispara cuando pulsamos el botón de ejecución de la barra de herramientas:

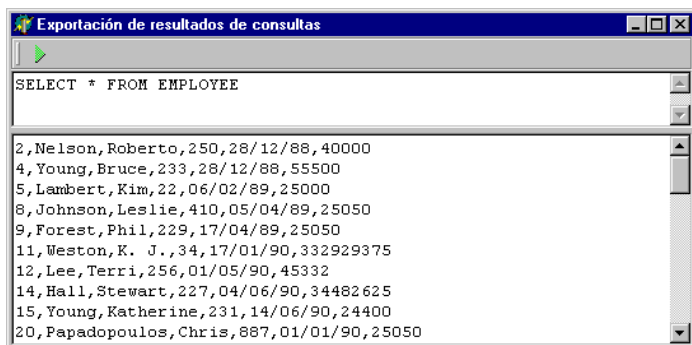
```
procedure TwndPrincipal.EjecutarClick(Sender: TObject);
var
  Salida: TIBOutputDelimitedFile;
  Fichero: string;
begin
  Fichero := ChangeFileExt(Application.ExeName, '.txt');
  IBSQL1.SQL := Memo1.Lines;
  IBDatabase1.Open;
```

```

try
    IBTransaction1.StartTransaction;
    try
        Salida := TIBOutputDelimitedFile.Create;
        try
            Salida.Filename := Fichero;
            Salida.ColDelimiter := ',';
            Salida.RowDelimiter := #13#10;
            IBSQL1.BatchOutput(Salida);
        finally
            FreeAndNil(Salida);
        end;
        IBTransaction1.Commit;
    except
        IBTransaction1.Rollback;
        raise;
    end;
finally
    IBDatabase1.Close;
end;
RichEdit1.Lines.LoadFromFile(Fichero);
end;

```

Aunque el método es un poco largo, en realidad la mayor parte se dedica a abrir la base de datos y la transacción, y a garantizar sus cierres. En el ejemplo he utilizado la clase *TIBOutputDelimitedFile*, para exportar al formato de texto con delimitadores. Para indicar los delimitadores entre campos y entre registros se utilizan las propiedades *ColDelimiter* y *RowDelimiter*.



## Los componentes de administración

A pesar de mis quejas durante todo el capítulo, hay cosas innegablemente útiles en InterBase Express: la principal es la colección de componentes de administración, que encontrará en la página *InterBase Admin* de la Paleta de Componentes:



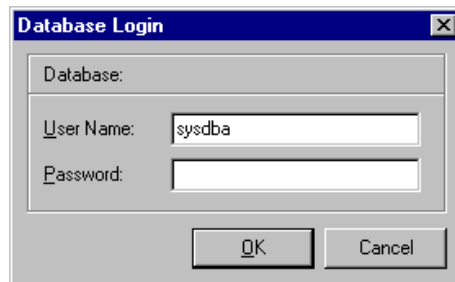
Los componentes de administración que existen en la versión actual son:

- *IBInstall, IBUnInstall*: Un euro si adivina lo que hacen.
- *IBBackupService, IBRestoreService*: Creación y restauración de copias de seguridad.
- *IBServerProperties, IBConfigService, IBLicensingService*: Permiten configurar un servidor, una base de datos en especial, y registrar o modificar las licencias instaladas en un servidor.
- *IBSecurityService*: Administración de usuarios y contraseñas.
- *IBValidationService, IBStatisticalService, IBLogService*: Validan una base de datos, extraen estadísticas de ellas, y recuperan los mensajes grabados en un fichero de registro administrado por el servidor.

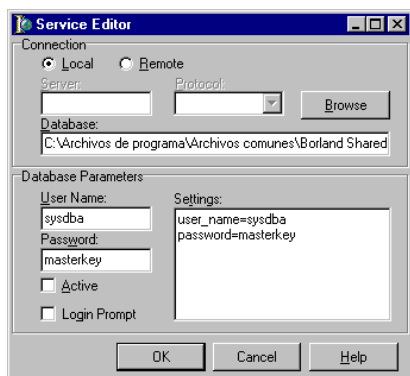
Todos ellos, excepto los componentes de instalación, descienden de la clase abstracta *TIBCustomService*, porque para su funcionamiento deben conectarse a diversos servicios especiales de InterBase. No mostraré la jerarquía, de todos modos, aunque hay clases intermedias interesantes. Estas propiedades se heredan de la base común:

Propiedad	Propósito
<i>Protocol</i>	Los principales son <i>Local</i> y <i>TCP</i>
<i>ServerName</i>	El nombre del servidor, si el protocolo no es local
<i>Params</i>	Parámetros, entre ellos, <i>user_name</i> y <i>password</i>
<i>Active</i>	Activar no es lo mismo que ejecutar: es sólo conectarse al servidor
<i>LoginPrompt</i>	¿Hay que pedir nombre y contraseña?

Los servicios que necesitan además una base de datos, tienen también una propiedad llamada *DatabaseName*; no existe, sin embargo un ancestro común que la introduzca. Y tenga cuidado con *LoginPrompt*, porque al igual que los componentes de conexión, los de administración muestran el conocido cuadro de diálogo sólo cuando *alguien* ha asignado un puntero a procedimiento en la variable global *LoginDialogExProc*. Recuerde que puede incluir la unidad *DBLogDlg* en el proyecto para que inicialice esa variable.



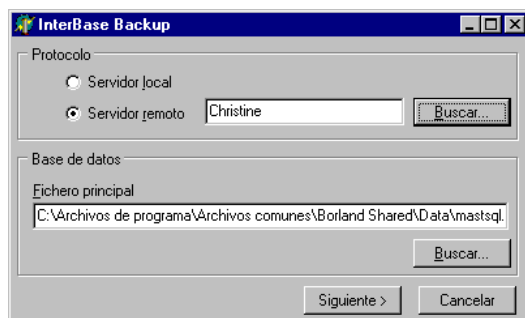
Cuando hacemos doble clic sobre un componente de administración aparece un cuadro de diálogo parecido al de la siguiente imagen:



Los componentes como *TServerProperties* que no necesitan una base de datos, utilizan el mismo diálogo, pero sin el cuadro de edición correspondiente.

## Copias de seguridad

Es muy sencillo usar cualquiera de los componentes de configuración, aunque haya que tener a mano la referencia, o el código fuente, para averiguar el nombre de las opciones y métodos necesarios. No obstante, voy a mostrarle cómo crear una pequeña aplicación para crear copias de seguridad:



La aplicación que he incluido en el CD como ejemplo utiliza una interfaz gráfica al estilo “asistente”, o *wizard*. En la primera página se piden los datos sobre el nombre del servidor y la base de datos de la que se hará la copia. Cuando el usuario pulsa el botón para pasar a la segunda página, aprovecho y copio los datos anteriores en el componente *TIBBackupService*, y activo la conexión al servicio. He dejado que *LoginPrompt* sea *True*, para que en ese momento se pida el nombre del usuario y su contraseña:

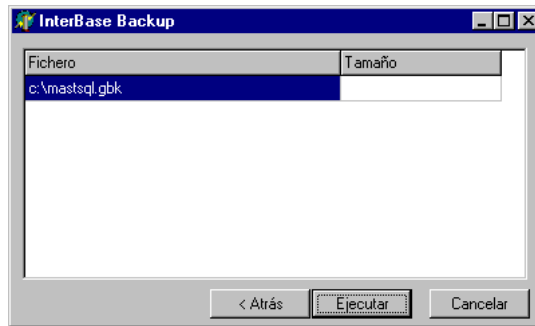
```
procedure TwndPrincipal.Button1Click(Sender: TObject);
begin
  if Trim(edDatabase.Text) = '' then
    raise Exception.Create('Falta el nombre de la base de datos');
  IBBackupService1.Active := False;
```

```

if rbLocal.Checked then begin
    IBBackupService1.Protocol := Local;
    IBBackupService1.ServerName := '';
end
else begin
    IBBackupService1.Protocol := TCP;
    IBBackupService1.ServerName := edServer.Text;
end;
IBBackupService1.DatabaseName := edDatabase.Text;
IBBackupService1.Active := True;
PageControl.ActivePageIndex := 1;
end;

```

Insisto otra vez en que al activar el componente no se ejecuta el servicio, sino que se establece la comunicación con el servidor y la base de datos, comprobando si el usuario tiene los permisos de acceso necesarios.



Para generar la copia de seguridad hay que suministrar el nombre de uno o más ficheros. Si hay más de uno, todos ellos excepto el último deben indicar el tamaño máximo permitido, en KB. Esa información se guarda en la propiedad *BackupFile*, de tipo *TStrings*. Para que el usuario introduzca los nombres de fichero he usado un componente *TValueListEditor*. No es que sea el control más apropiado, pero es fácil de usar. En esta página se encuentra también el botón *Ejecutar*, que dispara la generación de la copia:

```

procedure TwndPrincipal.Button4Click(Sender: TObject);
var
    I: Integer;
    FileName, Size: string;
begin
    IBBackupService1.BackupFile.Clear;
    for I := 0 to ValueListEditor1.Strings.Count - 1 do begin
        FileName := ValueListEditor1.Strings.Names[I];
        Size := ValueListEditor1.Strings.Values[FileName];
        if Size = '' then
            IBBackupService1.BackupFile.Add(FileName)
        else
            IBBackupService1.BackupFile.Add(FileName + '=' + Size);
    end;
    Button7.Enabled := False;
    PageControl.ActivePageIndex := 2;
    IBBackupService1.ServiceStart;

```

```
while not IBBackupService1.Eof do
    Memol.Lines.Add(IBBackupService1.GetNextLine);
    Button7.Enabled := True;
end;
```

El bucle final se ocupa de recuperar los mensajes que va produciendo el servicio durante su ejecución. Esto sucede así porque he activado la propiedad *Verbose* en el componente. Los mensajes se copian en un componente *TMemo* que he puesto en la tercera página del asistente.

## ADO y ADO Express

**A**NO SER QUE EL LECTOR SEA una réplica moderna del Conde de Montecristo, es imposible que no haya oído hablar en algún momento de ActiveX Data Objects, o ADO, como se le conoce más cariñosamente. Tras una larga guerra interna de siglas que pretendían ser interfaces para bases de datos, finalmente parece que es ADO la que se come el pastel, y que ningún programador merece su salario si no controla la nueva interfaz. ¿Qué hay de cierto en esto y qué parte es simple publicidad y aguas turbulentas?

Comenzaremos explicando el funcionamiento básico de ADO, a “bajo nivel”, y sólo después estudiaremos las clases con las que Delphi nos facilita el acceso a esta interfaz. Advierto desde ahora que estoy muy satisfecho con ADO y con los componentes de Delphi que permiten su fácil manejo. Alguien dijo alguna vez que si un gran número de monos aporreaba una cantidad similar de máquinas de escribir, era probable que uno de ellos lograra uno de los sonetos de Shakespeare. Creo que en Redmond son unos cuantos más, pero esta vez lo han hecho bien...

### OLE DB y ADO

OLE DB es el sustituto oficial de ODBC. El antiguo API estaba formado principalmente por funciones globales. OLE DB, por el contrario, consiste en un conjunto de interfaces de objetos basados en COM, lo que facilita la transparencia de la ubicación del proveedor de servicios. Para algunos programadores que se jactan de su pragmatismo, la distinción entre un software orientado a objetos y otro que no lo es no es algo sustancial. Por el contrario, creo que la programación orientada a objetos es esencial para desarrollar aplicaciones legibles y de fácil mantenimiento.

Además, OLE DB es más flexible que ODBC con respecto al origen de los datos. No sólo porque permite ver en forma relacional la información procedente de fuentes que no son verdaderas bases de datos relacionales (direcciones de correo, directorios publicados por servidores HTTP), porque ODBC también lo permite. Principalmente porque es más permisivo con el formato de cada registro individual proveniente de un mismo origen. Por ejemplo, si intentamos ver en forma de relación el listado de un directorio, tropezaremos con el ingrato hecho de que existen varios tipos muy diferentes de filas: las correspondientes a subdirectorios y las que corres-

ponden a verdaderos ficheros. OLE DB permite esta variedad, y así abre el camino a bases de datos orientadas a objetos, que hagan un uso intenso de los conceptos de clase y polimorfismo.

La cara oculta de OLE DB es su complejidad. Al intentar ofrecer un conjunto completo de posibilidades de acceso, OLE DB incluye un elevado número de clases e interfaces, y la programación del ejemplo más básico de acceso en modo lectura ocupa demasiadas líneas de código.

Para paliar este problema, Microsoft introdujo un conjunto de interfaces alternativas, basadas en *IDispatch*, y las denominó ADO, o *ActiveX Data Objects*. La implementación de ADO está totalmente basada en llamadas a OLE DB. Con ADO, además, Microsoft intentó calzar las insuficiencias de su lenguaje de programación más popular, Visual Basic, que no podía trabajar cómodamente con las interfaces de OLE DB que habían sido diseñadas originalmente en C++. Gracias al diseño basado en la automatización OLE, también ADO puede utilizarse sin mayores problemas en lenguajes interpretados de *script*, como JScript y VBScript.

Pero también es importante aprender ADO porque viene incluido como parte del núcleo de Windows 2000/XP, muy al estilo Microsoft. Si está usted utilizando el BDE, piense en las ventajas que tendría si éste formase parte del sistema operativo: se ahorraría la distribución e instalación de unos cuantos megabytes de software. En cualquier caso, mientras la tribu de Bill no domine el mundo, la instalación de ADO se distribuye en el CD de Delphi, y podrá encontrar actualizaciones en la siguiente dirección:

<http://www.microsoft.com/data>

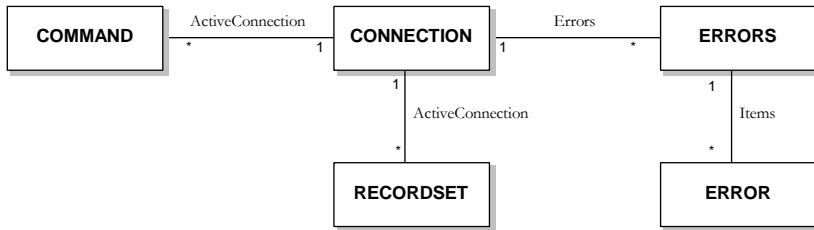
Actualmente OLE DB, y en consecuencia ADO, es capaz de trabajar con SQL Server, Oracle, Access y con cualquier sistema que soporte ODBC, aunque está claro que esta última opción es más lenta que el uso de un controlador OLE DB nativo. El controlador OLE DB para Oracle que se distribuye con ADO ha sido desarrollado por Microsoft, y su principal limitación es que no soporta los tipos de datos LOB (textos grandes, imágenes, sonidos). Oracle, por su parte, después de muchos refunfuños ha desarrollado su propio controlador OLE DB. Las versiones más recientes funcionan a la perfección, pero tenga cuidado si tiene una versión de Oracle anterior a la 8.1, porque puede tener problemas.

Veremos también algunos controladores algo peculiares, como el de MSDataShape, que permite estructurar datos provenientes de otros controladores.

## Las clases de ADO

En claro contraste con OLE DB, ADO ofrece unas pocas clases de muy fácil manejo. Las clases básicas son:





Solamente he incluido las clases más cercanas a la clase principal: la *conexión* (*connection*). Dentro de una conexión pueden ejecutarse varios *comandos* (*command*), y obtenerse, utilizando diversas técnicas, varios *conjuntos de registros* (*recordset*). A diferencia de lo que sucede en el BDE, los errores de ADO se deben buscar en la colección *Errors* que pertenece a la clase de conexiones. Una misma operación, como se deduce, puede provocar varios errores en cascada.

Existen clases también para representar *campos* (*fields*), *parámetros* (*parameters*) y *propiedades* (*properties*) de registros. Hay una clase de *registro* (*register*) que permite almacenar un registro de forma individual, y una clase *stream*, o *flujo*, que sirve principalmente para exportar o importar información a otros formatos; estas dos últimas clases aparecieron en la versión 2.5 de ADO.

La principal ventaja de ADO es que podemos, a pesar de esta descripción tan estructurada, crear cualquier objeto sin necesidad de haber creado antes un objeto del nivel superior. Por ejemplo, no necesitamos una conexión independiente para poder lanzar un comando ejecutable, o para abrir un conjunto de datos. A la inversa, es posible ejecutar un comando directamente a través de una conexión, sin crear explícitamente un objeto de la clase de comandos, y lo mismo sucede si necesitamos abrir un conjunto de registros. Esto nos permite escribir menos, lo cual es especialmente aconsejable para lenguajes como VBScript o JScript.

## Aplicaciones bien conectadas

Comenzaremos por las conexiones. Utilizaremos tres técnicas diferentes de acceso al API de ADO:

- Delphi, sin utilizar bibliotecas de tipos.
- Delphi, utilizando la biblioteca de tipos de ADO.
- JScript.

En el primer sistema, solamente tendremos que añadir la unidad *ComObj* en la cláusula **uses** de una unidad. Esta unidad contiene la función *CreateOleObject*, que podemos utilizar del siguiente modo:

```

var
  Conexión: Variant;
begin

```

```

    Conexion := CreateOleObject('ADODB.Connection');
    // Trabajar con la conexión ...
end;

```

**NOTA**

Todas las clases de ADO comienzan con el prefijo *ADODB* y se diferencian por el segundo identificador que se escribe a continuación.

¿Qué podemos hacer una conexión? ¡Abrirla y cerrarla, por supuesto! Por ejemplo:

```

var
    Conexion: Variant;
begin
    Conexion := CreateOleObject('ADODB.Connection');
    Conexión.Open('Provider=SQLOLEDB.1;Data Source=naraoa;' +
        'User ID=sa;Initial Catalog=MIR');
    // Hacer algo más, y finalmente ...
    Conexion.Close;
end;

```

El parámetro que le estamos pasando a *Open* es una *cadena de conexión*. En la siguiente sección aprenderemos más sobre estas cadenas. Por el momento, ¿hay algo más que podamos hacer con sólo una conexión? ¡Claro que sí! Analice la siguiente función:

```

procedure EjecutarInstruccion(
    const CadenaConexion, Instruccion: string);
var
    Conexion: Variant;
begin
    Conexion := CreateOleObject('ADODB.Connection');
    Conexion.Open(CadenaConexion);
    Conexion.Execute(Instruccion);
    Conexion.Close;
end;

```

Podríamos ejecutarla del siguiente modo:

```

begin
    // ...
    EjecutarInstruccion(
        'Provider=SQLOLEDB.1;Data Source=naraoa;User ID=sa;' +
        'Initial Catalog=pubs',
        'if exists(select * from sysobjects where name='prueba') ' +
        '    drop table prueba ' +
        'create table prueba(i integer)');
    // ...
end;

```

**NOTA**

Observe que no he hecho nada especial para destruir la conexión creada dentro de *EjecutarInstruccion*. Esto se debe a que Delphi se ocupa automáticamente de ello al terminar el bloque sintáctico donde hemos definido la variable *Conexion*. Probablemente, la destrucción de la conexión realice correctamente su cierre, pero no me fío...

## Cadenas de conexión

La sintaxis de las cadenas de conexión, como ya hemos visto, es muy simple, aunque varía de acuerdo al proveedor OLE DB que deseemos utilizar. El primer parámetro, *Provider*, indica cuál es el nombre de ese proveedor. He compilado la siguiente lista de nombres de proveedores a partir de los controladores instalados en el ordenador en el que estoy escribiendo:

Nombre abreviado	Nombre completo
<i>Microsoft.Jet.OLEDB.4.0</i>	Microsoft Jet 4.0 OLE DB Provider
<i>DTSPackageDSO.1</i>	Microsoft OLE DB Provider for DTS Packages
<i>MSDAIPP.DSO.1</i>	Microsoft OLE DB Provider for Internet Publishing
<i>MSDASQL.1</i>	Microsoft OLE DB Provider for ODBC Drivers
<i>MSDAORA.1</i>	Microsoft OLE DB Provider for Oracle
<i>SQLOLEDB.1</i>	Microsoft OLE DB Provider for SQL Server
<i>MSDAO.SP.1</i>	Microsoft OLE DB Simple Provider
<i>MS Remote.1</i>	MS Remote
<i>MSDataShape.1</i>	MSDataShape
<i>OraOLEDB.Oracle.1</i>	Oracle Provider for OLE DB
<i>DTSFlatFile.1</i>	SQL Server DTS Flat File OLE DB Provider

Según el proveedor que hayamos elegido, tendremos que incluir ciertos parámetros básicos en la cadena de conexión. Por ejemplo, para el proveedor OLE DB de SQL Server necesitamos como mínimo:

Parámetro	Valor
<i>Provider</i>	<i>SQLOLEDB.1</i>
<i>Data Source</i>	El nombre del servidor
<i>User ID</i>	La clave de usuario
<i>Password</i>	La contraseña

Podríamos también pasar los siguientes parámetros adicionales:

Parámetro	Valor
<i>Initial Catalog</i>	La base de datos que se debe activar inicialmente
<i>Integrated Security</i>	Activación de la seguridad integrada
<i>Application Name</i>	El nombre de la aplicación que hace la conexión
<i>Workstation ID</i>	Un identificador para el ordenador que se conecta

Note que no es necesario indicar una base de datos. Como sabemos, SQL Server define una base de datos por omisión para cada usuario con permisos de acceso al servidor. Además, siempre podemos cambiar la base de datos activa de la conexión enviando un comando *use* a través de la misma. Observe también que si activamos la seguridad integrada de NT no es necesario que suministremos el nombre del usuario y su contraseña.

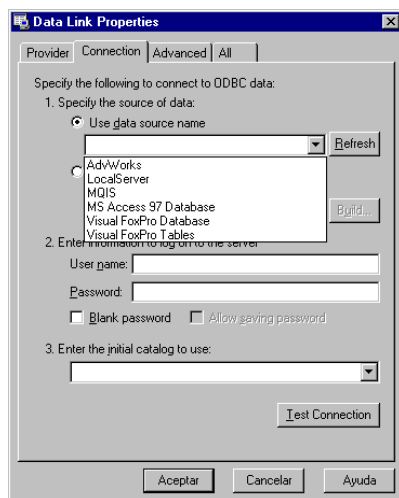
Una conexión a Oracle, a través del proveedor OLE DB de Oracle, utilizaría parámetros similares, pero el nombre del proveedor sería diferente, como muestro a continuación. Y al suministrar un valor para el parámetro *Data Source* estaríamos indicando tanto el servidor como la base de datos:

```
Provider=OraOLEDB.Oracle.1;Data Source=chrissie;
User ID=classique;Password=mozart;Persist Security Info=True
```

## Ficheros de enlaces a datos

¿Cómo he sabido de la existencia de esos parámetros y controladores? ¿Seré adivino, o quizás he leído la documentación de ADO? Si ha creído esto último, es usted una persona de mucha fe y excesiva inocencia. La documentación de ADO es tan mala como de costumbre, pero he utilizado un recurso que permite hacer pruebas fáciles con las conexiones, y que paso a describir.

En ADO existe algo parecido a los alias persistentes del BDE: los ficheros de enlace a datos (*data link files*). Se trata de ficheros que utilizan la extensión *.udl*: *Universal Data Link*, o enlace de datos universal. El Explorador de Windows reconoce esta extensión a partir de la instalación de ADO, y si hacemos doble clic sobre uno de estos ficheros, deber aparecer el siguiente asistente:



El asistente comienza su ejecución a partir de la segunda página. En la primera podemos seleccionar uno de los controladores OLE DB instalados en el equipo; el asistente ha supuesto, sin mucha base, que queremos utilizar OLE DB para controladores ODBC. ¿Ve ese botón que dice *Test Connection*? Es quien nos permite estar seguro de que hemos tecleado correctamente los parámetros de conexión.

Una vez que pulsamos el botón de aceptar, se crea un fichero de texto que contiene literalmente la cadena de conexión que acabamos de configurar visualmente, más una

pequeña cabecera. El texto, sin embargo, no está en formato ANSI, sino en Unicode. Si abrimos un fichero *udl* en el Bloc de Notas de Windows 98 veremos aparentes espacios en blanco (en realidad son caracteres nulos) entre letra y letra.

### ADVERTENCIA

El grave problema de los ficheros *udl* es que se encuentran al alcance de cualquiera. Como además su formato interno es "legible", es una peligrosa invitación a su modificación por cualquier usuario sabihondo y desaprensivo.

¿Cómo se utiliza un fichero *udl*? Si espera que le diga que hay que abrir el fichero, leer su contenido y crear entonces la cadena de conexión ... se equivoca. Basta con utilizar una cadena de conexión con la siguiente sintaxis:

```
FILE NAME=C:\Directorio\Directorio\Fichero.udl
```

Y ya se encarga ADO de leer el fichero de enlace para extraer los parámetros de conexión. Incluso si omitimos el directorio, ADO asume un directorio por omisión que la instalación se encarga de señalar. En mi ordenador, el directorio por omisión para los ficheros de enlace es:

```
C:\Archivos de programa\Archivos comunes\System\Ole DB\Data links
```

Suponiendo que estamos trabajando con un fichero ejecutable, la siguiente función escrita en Delphi devolvería una cadena de conexión haciendo referencia a un fichero de enlace con los datos necesarios para establecer la conexión:

```
function ConnectionString: string;
begin
    Result := 'FILE NAME=' + ChangeFileExt(ParamStr(0), '.udl');
end;
```

He asumido que el fichero de enlace reside en el mismo directorio que el ejecutable, que tiene el mismo nombre que éste, pero con extensión *udl*. Si se tratase de una DLL, tendríamos que utilizar la función *GetModuleFileName*, del API de Windows:

```
function ConnectionString: string;
var
    Buffer: array [0..MAX_PATH] of Char;
begin
    GetModuleFileName(HInstance, Buffer, SizeOf(Buffer));
    Result := 'FILE NAME=' +
        ChangeFileExt(StrPas(Buffer), '.udl');
end;
```

La variable global *HInstance* nos garantiza que obtenemos el nombre del fichero DLL, no el del programa que la llama, que sería el valor obtenido si pasáramos un cero en el primer parámetro.

## La biblioteca de tipos de ADO

Si nos ceñimos al uso de *Variant* y *CreateOleObject* para crear conexiones y ejecutar instrucciones, estaríamos perdiendo la posibilidad de una comunicación más rápida y eficiente con ADO. Lo que es más grave, nos expondríamos a recibir errores en tiempo de ejecución por utilizar nombres de métodos y propiedades inexistentes, pero que el compilador sería incapaz de señalar. Es más rápido y seguro utilizar la información de la *biblioteca de tipos de ADO* (*ADO Type Library*).

Cuando menciono la biblioteca de tipos de ADO, no le estoy sugiriendo que importe usted mismo la biblioteca. Por suerte, no hay necesidad de pedirle a Delphi que la traduzca para nosotros, porque éste ya nos ofrece la unidad *adoint.pas*. Si lo tiene a mano, ábralo dentro del Entorno de Desarrollo y échele un vistazo a su contenido.

Si añadimos la unidad *AdoInt* a la cláusula **uses**, podemos programar la ejecución de instrucciones SQL del siguiente modo:

```
procedure EjecutarInstruccion(
  const CadenaConexion, Instruccion: string);
var
  Conexion: Connection;
  Modificados: OleVariant;
begin
  Conexion := CoConnection.Create;
  Conexion.Open(CadenaConexion, '', '', 0);
  Conexion.Execute(Instruccion, Modificados, 0);
  Conexion.Close;
end;
```

El tipo *Connection* permite declarar variables que apunten a interfaces de objetos de tipo conexión, y está declarado en *AdoInt*. La clase auxiliar *CoConnection* es un añadido de Delphi que nos evita tener que llamar directamente a métodos de nivel más bajo. ¿Ha notado que hay nuevos parámetros en *Open* y *Execute*? Es que eran parámetros opcionales. Ahora, con la traducción que ha hecho Delphi de la biblioteca de tipos, se han convertido en parámetros obligatorios. Pruebe a escribir un nombre de método incorrecto para la conexión, y verá como Delphi detectará el error.

La lectura de la unidad con la biblioteca de tipos, aunque tediosa, puede también aportar información sobre los objetos proporcionados por un servidor, así como sus métodos, propiedades y eventos. Hay límites, claro está: nos enteraremos de que existe un método llamado *OpenSchema*, con tales y cuales parámetros, pero solamente podremos conjeturar qué hace dicho método.

## Conexiones desde JScript

La técnica utilizada por JScript para abrir conexiones, y en general para trabajar con cualquier clase COM, es similar al uso de *Variant* en Delphi que mostramos al principio del capítulo. Naturalmente, cuando hablo de abrir conexiones en JScript, me refiero a código JScript ejecutado *en el servidor*, dentro de páginas ASP. Si JScript se

ejecuta en el contexto de un navegador, en el lado cliente de Internet, sería una locura permitir la activación de objetos de clases arbitrarias residentes en ese ordenador, principalmente por seguridad del propio cliente.

Para hacer la siguiente prueba debe tener acceso a un Personal Web Server para Windows 98, o a un Internet Information Server. Copie el siguiente fragmento de código en un fichero nuevo de extensión *asp*.

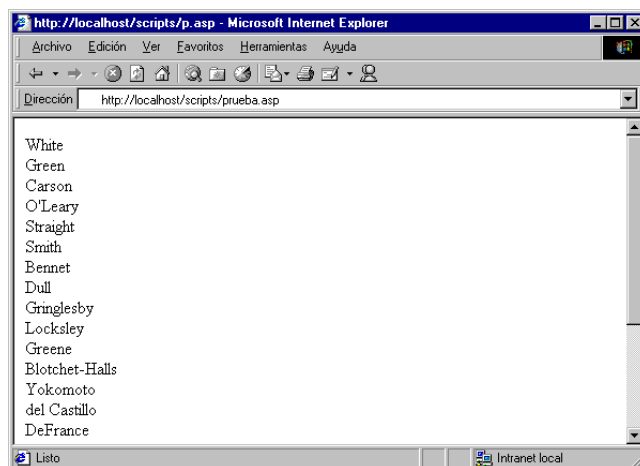
```
<script runat=server language=JScript>
var conexion = Server.CreateObject("ADODB.Connection");
var consulta = Server.CreateObject("ADODB.Recordset");

conexion.Open(
    "Provider=SQLOLEDB.1;Data Source=naroa;Initial Catalog=pubs",
    "sa", "");
consulta.Open("select * from authors", conexion);

while (! consulta.EOF)
{
    Response.Write(consulta("au_lname"));
    Response.Write("<br>");
    consulta.MoveNext();
}
</script>
```

Tenga cuidado con el valor del parámetro *Data Source*, dentro de la cadena de conexión. En el ejemplo he escrito *naroa*, el nombre de mi ordenador, pero debe sustituirlo por el nombre de su propio servidor SQL. Mueva el fichero al directorio *c:\Inetpub\scripts*, o a cualquier directorio en el que exista el permiso de ejecución de secuencias de comandos. Luego vaya a su navegador y pida la ejecución del fichero a través del protocolo HTTP. Por ejemplo, si está trabajando en el mismo ordenador donde está ejecutándose el servidor de Internet, teclee lo siguiente:

<http://localhost/scripts/prueba.asp>



El resultado debe ser una lista de apellidos de los autores registrados en la base de datos de demostración que acompaña a SQL Server.

## Conjuntos de registros

Para hacer menos árido el ejemplo anterior, tuve que colar a escondidas una clase de ADO todavía sin explicar: la clase *Recordset*, que permite manejar conjuntos de registros. Por decirlo con mayor claridad: la clase que nos permite ejecutar consultas sobre una base de datos.

Para simplificar la explicación, veamos antes un ejemplo similar, pero que utiliza Delphi y la biblioteca de tipos de ADO. Para simplificar al máximo el ejemplo, he dispuesto que el resultado se muestre dentro de un control *TListView*, asignando *vsReport* en su propiedad *ViewStyle*, y definiendo dos columnas, en *Columns*:

```
procedure TwndPrincipal.LeerDatos1Click(Sender: TObject);
var
  R: RecordSet;
begin
  R := CoRecordSet.Create;
  R.Open(
    'select au_lname, au_fname from authors',
    'Provider=SQLOLEDB.1;Initial Catalog=pubs;User ID=sa',
    adOpenStatic, adLockPessimistic, adCmdText);
  ListView1.Items.Clear;
  while not R.EOF do
  begin
    with ListView1.Items.Add do
    begin
      Caption := R.Fields['au_lname'].Value;
      SubItems.Add(R.Fields[1].Value);
    end;
    R.MoveNext;
  end;
  R.Close;
end;
```

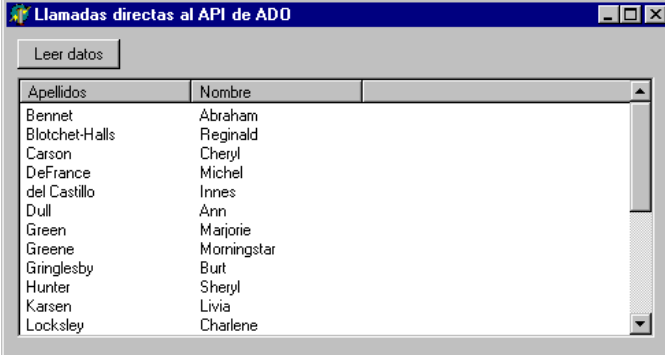
Lo primero que hemos hecho es crear un objeto COM mediante la clase auxiliar *CoRecordset* y obtener un puntero a su interfaz *Recordset*. Inmediatamente después, ejecutamos una consulta mediante el método *Open*, que como veremos, es una verdadera navaja suiza, porque podemos darle muchas interpretaciones alternativas a sus parámetros. Por ejemplo:

- 1 El primer parámetro, en este ejemplo, es una instrucción SQL que devuelve un conjunto de datos. Se interpreta de este modo porque el último parámetro contiene la constante *adCmdText*. Pero podríamos utilizar otras constantes, como *adCmdTable*, y entonces tendríamos que pasar el nombre de una tabla en el primer parámetro.
- 2 El segundo parámetro es una cadena de conexión. Pero el tipo de ese parámetro es *OleVariant*, y podríamos pasar a través de él un puntero a una conexión ya es-



- tablecida. Esta es una de las cochinadas de costumbre en Visual Basic, que podría haber sido resuelta mejor con la sobrecarga de métodos.
- 3 Los parámetros tercero y cuarto admiten menos variaciones. En el tercero se indica el tipo de curso que deseamos abrir, y el cuarto corresponde al tipo de bloqueos que utilizaremos. Explicaremos estos detalles más adelante.
  - 4 Y ya había mencionado el significado del último parámetro, ¿no? Solo que, además del modo de interpretar el primer parámetro, podemos agregar también algunas opciones relacionadas con la ejecución concurrente.

Después de abrir el conjunto de datos, vamos recorriendo cada fila con la ayuda de la propiedad *EOF* y el método *MoveNext*. Para cada fila seleccionada, extraemos el valor del nombre del autor y sus apellidos por medio de técnicas que hacen uso de la propiedad *Fields* del conjunto de registros. Esta propiedad apunta a una interfaz de colección llamada también *Fields*. Y aquí tenemos dos alternativas: o seleccionamos un campo por su nombre, como hacemos con el apellido, o por su posición en el registro, como hacemos con el nombre. Finalmente, el valor del campo se recupera como un *Variant* por medio de su propiedad *Value*. Como el nombre y los apellidos de un autor no pueden ser nulos, no he tenido que preocuparme por ese caso especial.



Apellidos	Nombre
Bennet	Abraham
Blotch-Halls	Reginald
Carson	Cheryl
DeFrance	Michel
del Castillo	Innes
Dull	Ann
Green	Marjorie
Greene	Morningstar
Gringlesby	Burt
Hunter	Sheryl
Karsen	Livia
Locksley	Charlene

Volviendo al ejemplo escrito en JScript, ahora podemos señalar que la apertura del conjunto de registros en ese primer ejemplo hace uso de una conexión ya establecida. La otra diferencia importante es la forma de extraer el valor de un campo. En Delphi escribimos una de estas dos expresiones:

```
R.Fields[0].Value
R.Fields['au_lname'].Value
```

aprovechando que la interfaz *Fields* tiene una propiedad vectorial *Items* que se asume por omisión, y que el índice de *Items* es de tipo *Variant*. En JavaScript tenemos suficiente con escribir:

```
consulta('au_lname')
consulta(0)
```

**NOTA**

Hay otra forma adicional de crear un conjunto de registros: utilizando directamente el método *Execute* de un objeto de conexión. Antes vimos su uso con instrucciones que no devuelven registros. No expliqué entonces que *Execute* puede devolver un puntero a una interfaz *Recordset* si la instrucción ejecutada devuelve filas. No pasa nada grave si no “capturamos” el puntero a la interfaz en el caso de que el método lo devuelva, porque los objetos COM llevan un contador de referencias, y Delphi o JScript destruirían la nueva interfaz al terminar el bloque de instrucciones activo.

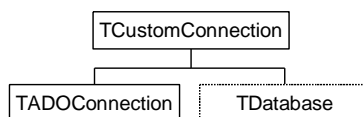
## ADO Express: su jerarquía

Programar con ADO es estúpidamente simple, pero más estúpido sería perderse las ventajas de utilizar la capa de alto nivel que ofrece Delphi con ADO Express: olvidarnos de las bibliotecas de tipos y trabajar con confiables componentes VCL, trabajar con controles *data-aware* que nos permiten comprobar en tiempo de diseño el aspecto que tendrá la aplicación al ejecutarse y acortar así el ciclo de desarrollo. No obstante, todo lo que hemos aprendido en lo que va de capítulo no es conocimiento inútil. Como veremos, la mayoría de las propiedades y métodos que hemos encontrado en ADO se vuelven a repetir en ADO Express. Ahora seguiremos presentado propiedades y métodos avanzados, pero ya desde el punto de vista de su manejo mediante componentes VCL.

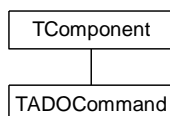
La siguiente imagen muestra los componentes de la página ADO de la Paleta de Componentes de Delphi 6:



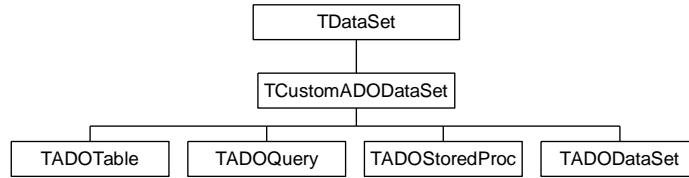
El primer componente de la página *ADO* se llama *TADOConnection*, y encapsula el acceso a la clase *Connection* de ADO. Sus relaciones familiares son estas:



El siguiente componente no tiene análogo en el BDE y se llama *TADOCommand*. Sirve para ejecutar directamente instrucciones SQL que no devuelven datos (creación de objetos, eliminación, actualización e inserción de registros). Encapsula la clase *Command* de ADO, y su jerarquía de herencia es muy simple:



A continuación vienen cuatro componentes que sirven para manejar conjuntos de registros, o *recordsets*:



Como vemos, tres de estos componentes parecen estar relacionados con algunos conjuntos de datos del BDE. Más adelante veremos que es cierto, siempre que no llevemos la analogía hasta sus últimas consecuencias.

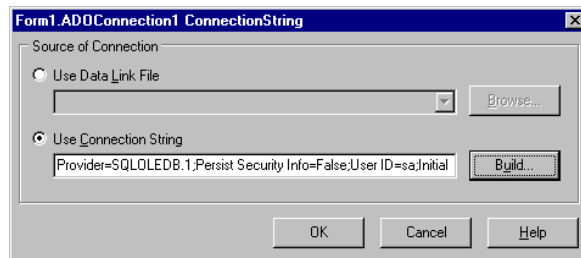
Por último tenemos un misterioso *TRDSCConnection*. *RDS* significa *Remote Data System*, y es el lejano equivalente en Microsoft de la tecnología de DataSnap y sus conexiones remotas.

## Conexiones ADO

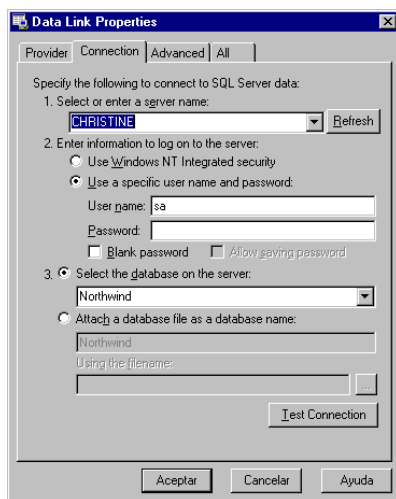
La propiedad fundamental de un *TADOConnection* es *ConnectionString*, que almacena una cadena con la descripción del tipo de conexión. Esta descripción consiste en pares parámetro/valor separados por puntos y comas. Por ejemplo, para conectarme a mi servidor local de SQL Server 7 utilizo la siguiente cadena:

```
ADOConnection1.ConnectionString := 'Provider=SQLOLEDB.1;' +
  'User ID=sa;Initial Catalog=pubs;Data Source=naroa';
```

Ya hemos visto la sintaxis de las cadenas de conexión, al menos para los proveedores OLE DB de SQL Server y Oracle. De todos modos, Delphi nos ofrece un editor de propiedades para no tener que adivinar parámetros de conexión. Si hacemos doble clic sobre un *TADOConnection*, o sobre su propiedad *ConnectionString*, aparecerá el siguiente cuadro de diálogo:



Observe que existen dos posibilidades: utilizar una cadena de conexión o mencionar un fichero de enlace de datos. Si pulsamos el botón *Build* aparecerá el asistente del propio ADO, que ya conocemos:



Si decidimos utilizar un fichero de enlace de datos, al cerrar el editor de propiedades encontraremos que la propiedad *ConnectionString* ha recibido un valor como éste:

```
FILE NAME=FicheroEnlace.UDL
```

Aparentemente, ésta sería la forma más sencilla y segura de configurar una conexión. Observe, sin embargo, lo que sucede cuando cambiamos el valor de la propiedad *Connected* a *True*: el valor de *ConnectionString* se modifica y expande como la cadena de conexión almacenada dentro del fichero *udl*. Si desactivamos nuevamente la conexión, el valor de *ConnectionString* vuelve a la “normalidad”.

## ERROR RESUELTO

En Delphi 5, si se guardaba el proyecto mientras la conexión estaba activa, se perdía irreversiblemente el valor “abreviado” de *ConnectionString* que hacía referencia al fichero de enlace. Este problema ha sido resuelto en Delphi 6.

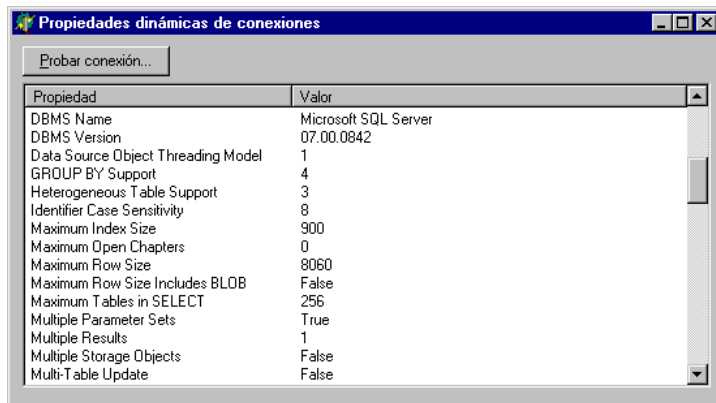
De todos modos, prefiero comprobar la presencia en tiempo de ejecución del fichero de enlace. El momento más adecuado para realizar este cambio es justamente al terminar la carga de las propiedades del componente, al inicializarse el módulo o formulario donde se encuentra. Y para interceptar ese momento necesitaremos reescribir el método virtual protegido *Loaded* del módulo o formulario:

```
procedure TDataModule1.Loaded;
var
  FicheroUDL: string;
  Buffer: array [0..MAX_PATH] of Char;
begin
  inherited Loaded;
  GetModuleFileName(HInstance, Buffer, SizeOf(Buffer));
  FicheroUDL := ChangeFileExt(StrPas(Buffer), '.UDL');
  if FileExists(FicheroUDL) then
    ADOConnection1.ConnectionString := 'FILE NAME=' + FicheroUDL;
end;
```

Estoy asumiendo que vamos a trabajar con la conexión abierta en tiempo de diseño. En ese caso, no nos habría valido interceptar el evento *OnCreate*, porque cuando se dispara ya se habría abierto la conexión. Sin embargo, durante la ejecución de *Loaded* todavía no se ha “aplicado” el valor almacenado en *Connected*, y tenemos la oportunidad de retocar los parámetros de la conexión *ad libitum*. En realidad, también nos valdría el evento *BeforeConnect*, pero prefiero utilizar *Loaded* porque puedo utilizar el mismo truco con otros componentes de acceso a datos.

## Propiedades dinámicas

Como sería imposible predecir todos los parámetros que puede necesitar una conexión con un sistema de bases de datos arbitrario, ADO utilizar *propiedades dinámicas* para configurar estos parámetros, pero también para ofrecernos información especial sobre la conexión. La siguiente imagen muestra algunas de esas propiedades dinámicas en una conexión a un servidor de SQL Server 7:



La lista de propiedades de un *TADOConnection* se almacena en *Properties*. Esta es una interfaz que apunta a una colección de objetos de tipo *Property*. Tanto el tipo de la lista como sus elementos son tipos definidos en la biblioteca de tipos de ADO. Algunas propiedades solamente se crean cuando se abre la conexión, mientras que otras siempre están presentes. El siguiente método muestra cómo recorrer la lista de propiedades y consultar sus nombres y valores.

```

procedure TwndPrincipal.bnTestClick(Sender: TObject);
var
    I: Integer;
begin
    ADOConnection1.ConnectionString := PromptDataSource(Handle, '');
    ADOConnection1.Open;
    try
        ListView1.Clear;
        for I := 0 to ADOConnection1.Properties.Count - 1 do
            with ADOConnection1.Properties.Item[I] do
                with ListView1.Items.Add do
                    begin
                        Caption := Name;
                    end
                end
            end
    finally
        ADOConnection1.Close;
    end
end

```

```

        SubItems.Add(VarToStr(Value));
    end;
finally
    ADOConnection1.Close;
end;
end;

```

*PromptDataSource* es una función global definida por Delphi en la unidad ADODB. Al ejecutarse, muestra el asistente para las cadenas de conexión, y devuelve la cadena generada como valor de retorno.

## Ejecución directa

Cuando trabajamos con SQL Server es muy frecuente que ejecutemos sentencias SQL generadas dinámicamente. A diferencia de InterBase, SQL Server puede ejecutar más de una sentencia en un solo envío. Si tuviésemos que insertar mil registros en InterBase, tendríamos que ejecutar mil veces una sentencia **insert** o un procedimiento almacenado equivalente. En cambio, con SQL Server crearíamos una cadena con varias instrucciones **insert** consecutivas, y la enviaríamos al servidor. Al disminuir el número de envíos por la red y la cantidad de compilaciones en el servidor, la operación duraría mucho menos. De ahí la mayor importancia de la ejecución directa en SQL Server respecto a InterBase.

### ORACLE

El equivalente en Oracle es la ejecución de un bloque anónimo, similar a un procedimiento almacenado, pero sin comenzar por *create procedure...*

La forma más sencilla de ejecutar una instrucción generada en tiempo de ejecución es usar uno de los siguientes métodos de la clase *TADOConnection*:

```

procedure TADOConnection.Execute(
    const CommandText: WideString;
    var RecordsAffected: Integer;
    const ExecuteOptions: TExecuteOptions = [eoExecuteNoRecords]);
function TADOConnection.Execute(
    const CommandText: WideString;
    const CommandType: TCommandType = cmdText;
    const ExecuteOptions: TExecuteOptions = []): _Recordset;

```

La primera de ellas es más apropiada para el lenguaje de definición de datos, y para las instrucciones de actualización; dicho de otra forma, para cualquier sentencia que no sea un **select**. Observe que esa variante no devuelve un *recordset*, reserva un parámetro para comunicar el número de filas afectadas y en las opciones de ejecución predefinidas se pide ignorar cualquier conjunto de filas que devuelva la instrucción.

La segunda versión, en cambio, es más apropiada para el caso en que exista un conjunto de filas como resultado. La función devuelve un puntero de interfaz del tipo *\_Recordset*, definido en la biblioteca de tipos de ADO. Si le hacemos caso a la documentación, una instrucción que no devuelva registros debería devolver un puntero

vacío, pero las pruebas que he realizado me demuestran que no es así. Si va a ejecutar una instrucción y no se sabe si se trata de un **select** o de otro tipo de sentencia, es preferible consultar el estado del *recordset* devuelto:

```
var
  RS: _Recordset;
begin
  RS := ADOConnection1.Execute(Instruccion);
  if Assigned(RS) and (RS.Status = adStatusOpen) then
    // ... la instrucción ha devuelto registros ...
  end;
```

Para que el compilador reconozca la constante *adStatusOpen* tenemos que añadir la unidad *ADOInt* a la cláusula **uses**.

Si la instrucción se conoce desde tiempo de diseño, o si vamos a ejecutarla varias veces con distintos parámetros, quizás sea preferible recurrir a un *TADOCommand*. Hay que asociar el componente a una conexión, a través de su propiedad *Connection*, o crear para él una conexión independiente, en *ConnectionString*. En su propiedad *CommandType* debemos elegir entre varios tipos de contenido a ejecutar. Por ejemplo, si asignamos *cmdText*, tenemos que asignar en *CommandText* una o más instrucciones SQL; pero si *CommandType* vale *cmdStoredProc*, en *CommandText* debemos poner un nombre de procedimiento almacenado. Las opciones de ejecución se asignan en la propiedad *ExecuteOptions*.

Y existe la posibilidad de compilar el comando, si es que lo vamos a ejecutar varias veces. Para compilar, debemos asignar *True* en la propiedad *Prepared*. Si está trabajando con SQL Server, tenga presente que este sistema siempre compila los procedimientos almacenados a código binario, por lo que no tiene mucho sentido “preparar” una llamada a procedimiento. En cambio, si preparamos instrucciones SQL independientes, el servidor crea un procedimiento almacenado temporal para poder compilar las instrucciones.

Por último, tanto *TADOConnection* como *TADOCommand* tienen una propiedad *CommandTimeout*, que indica los segundos de espera que se toleran al ejecutar un comando. Cualquier demora más allá de este intervalo, se considera un error. Se interrumpe la ejecución y se lanza una excepción. El motivo más frecuente para que se produzca un error por espera es tropezar con registros bloqueados por otra transacción. No olvide que SQL Server implementa el aislamiento de transacciones mediante bloqueos.

## Ejecución asíncrona

Muchas de las operaciones de ADO se pueden realizar en forma asíncrona: la apertura de la conexión, la ejecución de instrucciones y la recuperación de registros. La apertura de una conexión es suficientemente rápida como para no merecer la pena hacerlo en paralelo; al menos con SQL Server. Pero sí es útil poder ejecutar instruc-

ciones en paralelo con el hilo principal de la aplicación, incluso cuando hay un servidor de capa intermedia presente.

La opción que activa la ejecución asíncrona es *eo.AsyncExecute*. Debe incluirse en el parámetro *ExecuteOptions*, si utilizamos cualquiera de las variantes del método *Execute* de la clase *TADOConnection*. Si nos decidimos por un *TADOCommand*, hay que añadir la opción a su propiedad *ExecuteOptions*.

Para saber cuándo ha terminado realmente la ejecución en el servidor, ADO dispara el evento *OnExecuteComplete* en el *TADOConnection*. Su prototipo es:

```
type
  TExecuteCompleteEvent = procedure (
    Connection: TADOConnection; RecordsAffected: Integer;
    const Error: Error; var EventStatus: TEventStatus;
    const Command: _Command;
    const Recordset: _Recordset) of object;
```

Es importante saber que el evento *OnExecuteComplete* se dispara dentro del hilo principal de la aplicación; posiblemente, ADO esté usando la cola de mensajes para sincronizar este evento. Saber esto nos puede ahorrar preocupaciones innecesarias si necesitamos manejar este evento.

#### NOTA

Quizás le interese saber también que *OnExecuteComplete* tiene una pareja, *OnWillExecute*, que por cierto, no significa “al ejecutar a William”. Este otro evento se dispara antes de que se envíe el comando al servidor, y nos ofrece la posibilidad de modificar algunos de los parámetros de ejecución.

## Transacciones en ADO

No hay mayor misterio en el manejo de transacciones de ADO. Estos son los métodos que ofrece ADO Express:

```
function TADOConnection.BeginTrans: Integer;
procedure TADOConnection.CommitTrans;
procedure TADOConnection.RollbackTrans;
function TADOConnection.InTransaction: Boolean;
```

Aparte de los cambios de nombre, la mayor novedad es el valor de retorno del método que inicia una transacción, *BeginTrans*. Este valor corresponde al nivel de anidamiento después de iniciar la nueva transacción. ¿Es que SQL Server soporta transacciones anidadas? Bueno, primero hay que recordar que ADO puede acceder a *cualquier* sistema de bases de datos, y que hay sistemas que soportan transacciones anidadas, de las de verdad.

En segundo lugar, sí, SQL Server permite anidar transacciones, aunque la semántica no es la más deseable: un *commit* solamente tiene efecto cuando se trata de la transacción más externa. Un *rollback*, en cambio, anula la transacción, independientemente



del nivel de anidamiento. Estas dos operaciones, junto con el inicio de transacción mantienen un contador interno, *@@tranccount*, que es el que indica el número de niveles activos. La función *InTransaction* devuelve *True* cuando ese valor es mayor que cero.

Lo que sí puede ser un poco confuso es la abundancia aparente de niveles de aislamiento soportados. El nivel de aislamiento se establece en la propiedad *IsolationLevel*, de *TADOConnection*, y estos son los valores admitidos:

Nivel	Significado
<i>ilUnspecified</i>	El servidor ha ignorado el nivel que hemos solicitado
<i>ilChaos</i>	Solo se protegen las escrituras de otras transacciones de más alto nivel de aislamiento que la actual
<i>ilReadUncommitted</i>	Esta transacción no puede sobrescribir las escrituras sin confirmar de ninguna otra transacción, pero puede leerlas
<i>ilBrowse</i>	
<i>ilCursorStability</i>	No se pueden leer ya escrituras sin confirmar, pero se permiten lecturas no repetibles
<i>ilReadCommitted</i>	
<i>ilRepeatableRead</i>	Todo valor leído permanece estable hasta el fin de transacción
<i>ilSerializable</i>	Las restantes transacciones no pueden insertar registros fantasma
<i>ilIsolated</i>	

Como ve, el secreto consiste en que hay varias constantes que significan lo mismo. Y en ese simpático nivel llamado *ilChaos*, que me hace pensar en lo organizado que he dejado hoy mi apartamento.

El programador que conoce InterBase puede también confundirse con el significado de la propiedad *Attributes*, que es un conjunto con dos opciones: *xaCommitRetaining* y *xaAbortRetaining*. Sin embargo, este *retaining* no tiene nada que ver con la opción *commit retaining* de InterBase. Simplemente indican si queremos que una llamada a *CommitTrans* o *RollbackTrans* active inmediatamente después una nueva transacción. ¿Mi consejo? No active esas opciones.

## Control de errores

Para hablar del control de errores debemos saber cómo enterarnos de que ha ocurrido alguno, y cómo reconocer el tipo de error que se ha producido. Comencemos por lo segundo: si se produce un error en cualquier operación relacionada con un componente de ADO, tenemos siempre que buscar la información sobre el error en la conexión, en su propiedad *Errors*. Las siguientes declaraciones simplificadas muestran los tipos que debemos conocer:

```
property TADOConnection.Errors: AdoInt.Errors;

type
  Errors = interface(_Collection);

  Error = interface(IDispatch)
    property Number: Integer;
```

```

property Source: WideString;
property Description: WideString;
property HelpFile: WideString;
property HelpContext: Integer;
property SQLState: WideString;
property NativeError: Integer;
end;

```

Como vemos, la propiedad *Errors* apunta a una colección de punteros a interfaces de tipo *Error*, y tanto la colección como cada error por separado son interfaces COM definidas por ADO. En particular, todas las propiedades del tipo *Error* son sólo para lectura. Y no se entusiasme demasiado con las propiedades *HelpFile* y *HelpContext*, porque al menos con ADO 2.6 y SQL Server 2000 no traen información alguna. Las dos propiedades relevantes para nosotros son *Description*, que es el mensaje de error en sí, y *NativeError*, el código de error definido por el servidor.

La propiedad *Errors* es también de interés porque en ella podemos encontrar los mensajes informativos que se lanzan desde SQL Server con la instrucción **print**. Considere el siguiente procedimiento almacenado escrito en Transact SQL:

```

create procedure clsAddKeyword
    @keyword T_NAME, @keywordID T_IDENTIFIER output as
begin
    set @keywordID = -1
    select @keywordID = KeywordID
    from KEYWORDS
    where KeywordName = @keyword

    if (@keywordID = -1)
    begin
        insert into KEYWORDS (KeywordName)
        values (@keyword)
        set @keywordID = @@identity
        print 'Palabra clave creada'
    end
end

```

El procedimiento debe comprobar si ya existe determinada palabra clave, para devolver su identificador numérico. Si no existe, se crea en ese momento. Si al diseñador de la base de datos le hubiese interesado detectar los casos en que tiene lugar la creación, podría haberlo resuelto jugando con el valor de retorno, o con parámetros de salida adicionales. Pero la idea se le ha ocurrido tardíamente, cuando ya existen aplicaciones que utilizan el procedimiento almacenado. Para no cambiar las aplicaciones clientes, ha decidido notificar de la inserción mediante la instrucción **print**.

Las aplicaciones que no sean conscientes de esta adición, no notarán nada extraño. Pero si revisan la colección *Errors* de la conexión después de ejecutar el procedimiento, encontrarán un error de baja prioridad, que es como trata SQL Server este tipo de mensajes. No se lanza una excepción como respuesta, ni se interrumpe el flujo de ejecución, sin embargo. También se dispara el evento *OnInfoMessage*, del componente *TADOConnection*:

```

procedure TmodDatos.ADOConnection1InfoMessage (
    Connection: TADOConnection; const Error: Error;
    var EventStatus: TEventStatus);
begin
    GuardarMensaje(Now, Error.Description);
end;

```

### ADVERTENCIA

Según la documentación, *OnInfoMessage* se dispara una sola vez aún cuando **print** se haya ejecutado varias veces en el servidor. En cambio, en *Errors* tendríamos siempre todos los mensajes informativos emitidos. En la práctica, no siempre *Errors* conserva todos los mensajes, especialmente si han sido emitidos desde el mismo procedimiento almacenado.

## Conjuntos de datos en ADO Express

El principal error al iniciarnos con ADO es llevar la analogía con el BDE demasiado lejos. Sabemos que la implementación que el BDE proporciona a los componentes *TTable* y *TQuery* es radicalmente distinta. Por lo tanto, la experiencia nos lleva a esperar una situación similar para *TADOTable*, *TADOQuery* y sus dos hermanos. ¡Predicción fallida!

En realidad, el componente *TADODataset*, o más bien su ancestro *TCustom.ADO-Dataset*, lleva en sus entrañas todo el código necesario para la implementación posterior de tablas, consultas y procedimientos de selección. Si modificamos convenientemente algunas propiedades de un *TADODataset* obtendremos equivalentes funcionales exactos de los componentes *TADOQuery*, *TADOTable* y *TADOStoredProc*.

Las dos propiedades principales de *TADODataset* son *CommandType* y *CommandText*; la primera pertenece a un tipo enumerativo, mientras que la segunda contiene un valor alfanumérico. De acuerdo al valor almacenado en *CommandType*, cambia el significado de la cadena asignada en *CommandText*, y se modifica el comportamiento del componente:

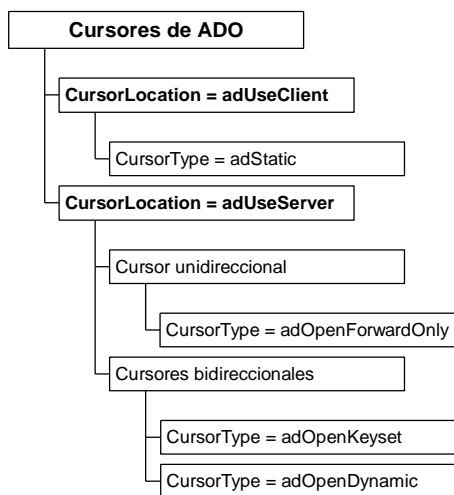
Valor	Significado de <i>CommandText</i>
<i>cmdUnknown</i>	El tipo de comando lo determina ADO automáticamente
<i>cmdText</i>	Contiene una sentencia SQL
<i>cmdTable</i>	Contiene el nombre de una tabla
<i>cmdStoredProc</i>	Contiene el nombre de un procedimiento almacenado
<i>cmdFile</i>	Contiene el nombre de un fichero en formato XML o ADTG
<i>cmdTableDirect</i>	También contiene el nombre de una tabla

Los restantes conjuntos de datos de ADO Express no publican *CommandType* y *CommandText*, e introducen estas otras propiedades:

- *TADOQuery*: *CommandType* vale *cmdText*, y ofrece *SQL*, una lista de cadenas, para que indiquemos la sentencia SQL.

- *TADOTable*: *CommandType* se inicializa con *cmdTable*, y en *TableName* se almacena el nombre de la tabla.
- *TADOStoredProc*: *CommandType* vale *cmdStoredProc*, y *StoredProcName* contiene el nombre del procedimiento.

La implementación real del *recordset* encapsulado por el conjunto de datos se determina por la combinación de dos propiedades: *CursorLocation* y *CursorType*, que estudiaremos a continuación. No todas sus combinaciones tienen sentido. Por este motivo, antes de mostrarle las constantes correspondientes, quiero que estudie el siguiente gráfico, que muestra las combinaciones posibles:



Y antes de que comience a mesarse los cabellos, le advierto que casi siempre se utilizan los cursores en el cliente, de tipo estático. ¿Más tranquilo ahora?

## Ubicación del cursor

El primer criterio para clasificar un cursor o conjunto de registros es el extremo de la red en que se implementa: el lado cliente o el lado del servidor. En el primer caso, *CursorLocation* valdría *clClient*, y en el otro, *clServer*. Concentremos la atención en los cursores del extremo servidor: ¿cómo funcionan? La explicación que daré a continuación puede tener algunas inexactitudes, y los detalles exactos de la implementación pueden cambiar. Pero le permitirá evaluar si debe utilizar o no este tipo de cursores en determinadas circunstancias:

- En primer lugar, al pasar la instrucción SQL al lado del servidor, se crea un objeto en ese extremo que será el responsable de interactuar con la base de datos.
- El objeto remoto recién creado aprovecha los procedimientos almacenados de manejo de cursores de SQL Server. Para que se haga una idea, he aquí el nombre de algunos de dos de esos procedimientos: *sp\_cursoropen* y *sp\_cursorfetch*.

- Paralelamente, en el cliente se crea un *proxy*, que es simplemente un “clon” del objeto remoto, y que la aplicación cliente puede utilizar para manejar el cursor a distancia. Este manejo remoto, naturalmente, está basado en COM.

Por lo tanto, cuando abrimos el conjunto de datos, *no* estamos trayendo todos los registros que puede devolver la consulta. Las llamadas sucesivas a *MoveNext* y a otros métodos de navegación son las responsables de que los registros solicitados viajen a través de la red. La consecuencia más importante de esta técnica es la siguiente:

*“Mientras esté activo el Recordset, el servidor mantendrá un cursor local abierto, consumiendo recursos innecesariamente”*

Event Class	Text	Start Time	Duration	Reads	Writes
TraceStart		19:07:09.770			
ExistingConnection		18:43:36.537	1413353	19283	0
RPC:Starting	sp_cursoropen @P1 output, N'select * fro	19:07:13.443			
RPC:Completed	sp_cursoropen @P1 output, N'select * fro	19:07:13.443	1280	20957	44
RPC:Starting	sp_cursorfetch 540229724, 1, 0	19:07:14.797			
RPC:Completed	sp_cursorfetch 540229724, 1, 0	19:07:14.797	93	15	0
RPC:Starting	sp_cursorfetch 540229724, 16, 1, 1	19:07:14.797			
RPC:Completed	sp_cursorfetch 540229724, 16, 1, 1	19:07:14.797	93	15	0
RPC:Starting	sp_cursorfetch 540229724, 16, 1, 1	19:07:14.893			
RPC:Completed	sp_cursorfetch 540229724, 16, 1, 1	19:07:14.893	4	6	0

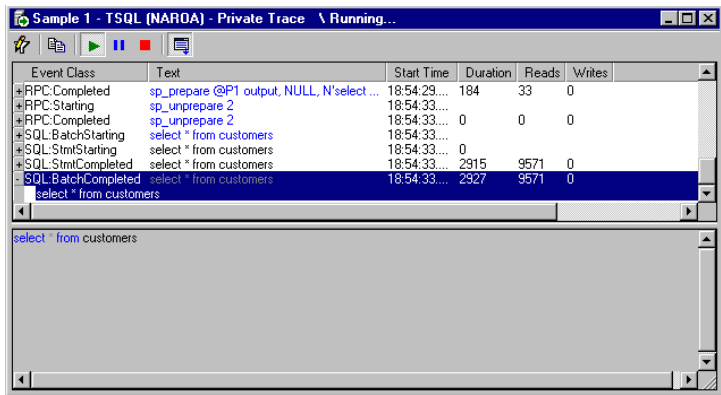
```

declare @P1 int
declare @P3 int
declare @P4 int
declare @P5 int
set @P1=NULL
set @P3=98305
set @P4=311300
set @P5=NULL
exec sp_cursoropen @P1 output, N'select * from customers
* @P3 output, @P4 output, @P5 output
  
```

¿Cómo funciona entonces la alternativa? Cuando abrimos un conjunto de datos con la opción *clUseClient*, también se crea un objeto remoto en el servidor que evalúa la consulta SQL. Pero esta vez, se recorren todos los registros, se empaquetan en una estructura lineal que se transmite inmediatamente por la red hasta el extremo cliente, y se cierra el objeto remoto temporal. Grabe estas palabras en su disco duro cerebral:

*“Un cursor en el cliente trae todas las filas de la consulta de golpe al ordenador donde se ejecuta la aplicación”*

Y eso, ¿es bueno o es malo? Aunque parezca mentira a primera vista, es muy bueno, pero siempre que la aplicación pida solamente consultas con un tamaño razonable de registros. Al recibir todos los registros en un solo envío, ADO puede cerrar el cursor, aunque no cierre la conexión. En particular, esto implica que cualquier bloqueo que se haya solicitado durante el recorrido del cursor se libera inmediatamente después de que el cliente tenga todos los registros.



Usted se preguntará: ¿por qué menciono los bloqueos si lo único que queremos es leer unos cuantos registros? Recuerde que los niveles de aislamiento serializable y de lecturas repetibles deben pedir bloqueos compartidos o de lectura para cada registro que visitemos dentro de una transacción. Por supuesto, podríamos bajar el nivel de aislamiento, pero podríamos encontrarnos con situaciones curiosas de difícil seguimiento si así lo hiciéramos.

Hay más: al cerrarse el cursor, desaparece otra importante estructura de datos que el servidor tenía que mantener para cada cliente. De esa manera, el servidor puede atender más conexiones simultáneas. Y, aunque no hemos cerrado la conexión, siempre podremos hacer trucos con MTS o COM+ para compartirlas con otros usuarios.

## Tipos de cursores

La otra dimensión que se utiliza para clasificar un cursor es el *tipo de cursor*, ya sé que no es un buen nombre. Estos son los valores que puede tomar *CursorType*:

Constante	Significado
<i>ctOpenForwardOnly</i>	Cursor unidireccional
<i>ctKeyset</i>	Cursor basado en claves primarias
<i>ctDynamic</i>	Cursor dinámico; implementación más compleja
<i>ctStatic</i>	Cursor especial para el extremo cliente
<i>ctUnspecified</i>	Dejamos que ADO decida

Pero, como ya hemos visto, la ubicación del cursor determina qué tipos son los aceptables. En particular, si es un cursor en el lado cliente, el único tipo permitido es *ctStatic*. Cuando las consultas que utilizamos devuelven una cantidad razonablemente limitada de registros, es el tipo de cursor que debemos utilizar. En primer lugar, por que se crean más rápidamente, en segundo, porque liberan enseguida los recursos que utilizan en el servidor. Y en tercer y último lugar, porque al quedar almacenados los registros en una caché cliente, veremos que pueden ordenarse a nuestro antojo, filtrarse, y ser víctimas de muchas otras operaciones que nos facilitan nuestra vida como programadores.

Supongamos ahora que el cursor se ubica en el servidor. ADO instruye entonces a OLE DB para que abra un cursor Transact SQL en el servidor, y tenemos las otras tres opciones de implementación a nuestra disposición. La primera posibilidad es abrir un cursor que solamente puede moverse hacia delante, en una sola dirección, utilizando *ctOpenForwardOnly*. Esta opción es, dentro de los cursores en el servidor, la que menos esfuerzo exige al servidor.

### RECOMENDACIÓN

Los cursores de tipo *ctOpenForwardOnly* son útiles en SQL Server cuando necesitamos exportar el resultado de una consulta en un fichero, y para la impresión de informes. Eso sí, debemos aumentar el valor de la propiedad *CacheSize* del cursor a un valor suficientemente alto. Curiosamente, si va a complementar ADO con DataSnap, es más eficiente utilizar cursores en el lado cliente, aunque la analogía con DB Express sugiera lo contrario.

Si necesitamos navegar hacia atrás, o dando saltos, debemos escoger entre *ctKeyset* y *ctDynamic*. Con la primera técnica, en el momento en que abrimos el cursor se evalúan todas las filas que pertenecen a la consulta enviada, y el conjunto de registros del resultado se crea almacenando en el servidor las claves primarias de esos registros. Posteriormente, sin embargo, podrían crearse nuevos registros o ser eliminados algunos de los existentes, y el cursor sería insensible a esos cambios (protestaría por el borrado, como es lógico). Si necesitamos más dinamismo, para eso está *adDynamic* que exige la implementación más complicada y costosa ... y que debemos evitar mientras podamos.

Si usted va a migrar a SQL Server una aplicación escrita en Paradox, dBase o algo similar, y no tiene tiempo para modificar toda la interfaz de usuario, quizás tenga que usar cursores en el servidor, del tipo *ctKeyset*, para simular la navegación sobre tablas con algún éxito. Si la aplicación se debe ejecutar en una red de área local rápida, no tendrá mayores problemas. Pero tengo una experiencia muy negativa con los cursores en el servidor cuando la comunicación se establece a través de una línea lenta, aunque sea dedicada.

## Control de la navegación

Varias propiedades adicionales nos permiten un mayor control de la navegación:

Propiedad	Propósito
<i>MaxRecords</i>	Número máximo de registros que admite el conjunto de datos
<i>CacheSize</i>	Número de registros que lee ADO cada vez
<i>BlockReadSize</i>	El mismo significado que para los conjuntos de datos del BDE

La propiedad *BlockReadSize* pertenece al esqueleto de métodos implementado por la VCLDB. En cambio, *MaxRecords* y *CacheSize* corresponden a propiedades nativas de ADO. *CacheSize* determina la cantidad de registros que utiliza la caché del software proveedor de ADO, pero sólo tiene sentido cuando el cursor se encuentra en el ser-

vidor. Inicialmente su valor es 1. Si asignamos otro valor, digamos que 10, los registros se transfieren del servidor al cliente de 10 en 10. Puede imaginar que esta lectura en bloques es ligeramente más eficiente que la recuperación de los mismos registros de uno en uno.

Otra técnica interesante consiste en utilizar las propiedades *PageSize*, *PageCount* y *AbsolutePage* del conjunto de registros (*recordset*) asociado al componente ADO. Estas propiedades no han sido expuestas por ADO Express, es decir, si queremos aprovecharlas tenemos que utilizar la propiedad *Recordset*, que nos da acceso de bajo nivel al API de ADO. En particular, estas tres propiedades son las preferidas por los libros de programación para Internet con ASP. ¿Hay que dividir un listado por páginas? Según esos libros, basta con establecer un tamaño de página en *PageSize*, y luego asignar en *AbsolutePage* la página a la que queremos saltar. Haga la prueba, sin embargo, de ver qué sucede en el SQL Profiler. Verá entonces que para saltar milagrosamente a la trigésimo octava página, el servidor se verá forzado a evaluar las treinta y siete páginas anteriores. En otras palabras: no pagaría un euro oxidado por esta técnica.

## Una consulta, varias respuestas

En una aplicación típica cliente/servidor, muchas veces es la red la que se convierte en el cuello de botella por su lentitud. Supongamos que hay que mostrar el resultado de dos consultas en pantalla. Lo normal sería, precisamente, utilizar dos componentes de consulta para traer dos conjuntos de registros. Pero SQL Server no es exactamente un sistema “normal”, pues nos permite poner estas dos instrucciones dentro de un mismo *TADODataset* (estas tablas pertenecen a la base de datos *pubs*, que acompaña a SQL Server):

```
select * from authors

select * from titles
```

Si hace la prueba de acoplar una rejilla a un conjunto de datos que tenga esas dos instrucciones, verá sin embargo que solamente se muestran los datos de autores. Bueno, traiga un segundo *TADODataset*, configúrelo para que sus datos, cuando los tenga, se muestren en una segunda rejilla, y escriba las siguientes instrucciones para que se ejecuten en respuesta al evento *OnCreate* del formulario:

```
procedure TwndPrincipal.FormCreate(Sender: TObject);
var
  Recs: Integer;
begin
  ADODataset1.Open;
  ADODataset2.Recordset := ADODataset1.NextRecordset(Recs);
end;
```

La clave estaba en recorrer los potencialmente varios conjuntos de registros que puede devolver una misma consulta en ADO. ¿El objetivo de esta técnica? Pues que



usted pueda enviar en un mismo viaje varias consultas al servidor, y que el servidor puede hacer lo mismo con su respuesta. Así se ahorran viajes innecesarios a través de la red.

au_id	au_lname	au_fname	phone	address
172-32-1176	White	Johnson	408 496-7223	10932 Bigge Rd.
213-46-8915	Green	Marjorie	415 986-7020	309 63rd St. #411
238-95-7766	Carson	Cheryl	415 548-7723	589 Darwin Ln.
267-41-2394	O'Leary	Michael	408 286-2428	22 Cleveland Av. #14

title_id	title	type	pub_id	price
BU1032	The Busy Executive's Database Guide	business	1389	19.99
BU1111	Cooking with Computers: Surreptitious Balance Sheets	business	1389	11.95
BU2075	You Can Combat Computer Stress!	business	0736	2.99
BU7832	Straight Talk About Computers	business	1389	19.99
MC2222	Silicon Valley Gastronomic Treats	mod_cook	0877	19.99

## Filtros, ordenación y recuperación asíncrona

El funcionamiento de los cursores ADO en el cliente es muy similar en ciertos detalles al de los conjuntos de datos clientes de DataSnap. Como se mantiene una caché local desconectada del servidor, podemos efectuar de forma eficiente algunas operaciones dinámicas que serían imposibles si el cursor se encontrase en el servidor.

Una de esas operaciones es la ordenación dinámica. En *TADODataset* se puede establecer un criterio de ordenación “local” modificando la propiedad *IndexFieldNames*. Y lo mismo sucede con un *TADOTable*. En cambio, un componente *TADOQuery* carece, aparentemente, de las propiedades necesarias. Pero lo que sucede es que en ese caso hay que modificar la propiedad *Sort*, declarada en la sección **public** de la clase. En realidad, es *IndexFieldNames* la que se implementa recurriendo a *Sort*. Esto se deja notar en que hay que tener un poco de cuidado con la forma en que separamos los nombres de campos, cuando se usar una clave compuesta. *Sort* solamente admite la coma como separador:

```
ADOQuery1.Sort := 'city,au_lname';
```

En cambio, *IndexFieldNames* acepta también el punto y coma, e incluso una mezcla de separadores:

```
ADOTable1.IndexFieldNames := 'city,au_lname;au_fname';
```

En los dos casos, se permite también indicar la dirección del criterio, si es ascendente o descendente:

```
ADOTable1.IndexFieldNames := 'Prioridad DESC;Apellidos ASC';
```

Pero debe asegurarse de que las palabras *ASC* y *DESC* aparezcan en mayúsculas.

Los cursores ubicados en el cliente son también los únicos en ADO que admiten filtros. Las propiedades son las mismas que en un *TClientDataSet*, pero la sintaxis cambia, porque son implementados por ADO. En este momento, las condiciones que se admiten son comparaciones entre columnas y constantes, enlazadas entre sí con los típicos operadores lógicos:

```
au_fname = 'Ringer'
profesion = 'abogado' and buenaPersona = '1'
```

Como anécdota, hay otra forma de generar un filtro, basada en marcas de posición. La clase *TCustomADODataset* ofrece el siguiente método:

```
procedure TCustomADODataset.FilterOnBookmarks (
  Bookmarks: array of const);
```

Tenga cuidado, porque el vector que se pasa como parámetro es un vector abierto de constantes, que se implementa en realidad como un vector del tipo interno *TVarRec*. Menciono la existencia de este método porque se ajusta en cierto modo a la posibilidad de utilizar la selección múltiple de filas en una rejilla de datos:

```
procedure TwndPrincipal.acFiltrarExecute(Sender: TObject);
var
  I: Integer;
  Marcas: array of TVarRec;
begin
  if acFiltrar.Checked then
  begin
    if DBGrid1.SelectedRows.Count = 0 then Exit;
    SetLength(Marcas, DBGrid1.SelectedRows.Count);
    for I := Low(Marcas) to High(Marcas) do
      Marcas[I].VPointer := Pointer(DBGrid1.SelectedRows[I]);
    ADOQuery1.FilterOnBookmarks(Marcas);
  end
  else
    ADOQuery1.Filtered := False;
end;
```

au_id	au_fname	au_lname	phone	address
172-32-1176	White	Johnson	408 496-7223	10932 Bigge Rd.
213-46-8915	Green	Marjorie	415 986-7020	309 63rd St. #411
238-95-7766	Carson	Cheryl	415 548-7723	589 Darwin Ln.
267-41-2394	O'Leary	Michael	408 286-2428	22 Cleveland Av. #14
274-80-9391	Straight	Dean	415 834-2919	5420 College Av.
341-22-1782	Smith	Meander	913 843-0462	10 Mississippi Dr.
409-56-7008	Bennet	Abraham	415 658-9932	6223 Balem St.

Antes expliqué que cuando abrimos un cursor en el cliente se leen todos los registros de manera automática... pero olvidé mencionar las excepciones. Primero, si activamos en *ExecuteOptions* la opción *eoAsyncExecute*, la evaluación de la consulta en el servidor transcurre de forma asíncrona: no es necesario esperar a que el servidor sepa qué registros devolver. Ahora bien, en el momento en que están listos los registros para

ser enviados, se restablece el sincronismo con el hilo principal, y la recuperación de los registros a través de la red tiene lugar de forma síncrona.

Si realmente queremos recuperar los registros de forma asíncrona, debemos activar una de las opciones *eo.AsyncFetch* o *eo.AsyncFetchNonBlocking*. La primera de las opciones activa la recuperación asíncrona a partir de cierta cantidad inicial de registros mínima, mientras que la segunda no exige ese mínimo de filas. En cualquiera de los dos casos, los eventos *OnFetchProgress* y *OnFetchComplete* nos avisan del progreso de la operación, y del fin de la misma.

## Importación y exportación

Es posible guardar en un fichero el conjunto de filas gestionado por un conjunto de datos de ADO, y recuperarlo posteriormente. Los métodos implicados son:

```
procedure TCustomADODataset.LoadFromFile(  
    const FileName: WideString);  
procedure TCustomADODataset.SaveToFile(  
    const FileName: WideString = '';  
    Format: TPersistFormat = pfADTG);
```

Los formatos aceptados son dos: el previsible XML y ADTG, o *Advanced Datagram*, un formato particular definido por Microsoft. El formato usado en XML es muy diferente al que utiliza Borland para guardar sus conjuntos de datos clientes. Este humilde programador no ha podido realizar la transformación usando el *XML Mapper* de Delphi, pero puede que sea culpa mía.

```
<z:row au_id='172-32-1176' au_lname='White' au_fname='Johnson'  
    phone='408 496-7223' address='10932 Bigge Rd.' city='Menlo Park'  
    state='CA' zip='94025' contract='True'/>
```

Si debe guardar datos en ficheros para utilizarlos solamente en ADO, es preferible utilizar ADTG, por varios motivos: es el formato que se utiliza para la transmisión de datos por la red, el motor de ADO se encarga de su análisis, mientras que el análisis de XML es responsabilidad del motor de Internet Explorer. Por último, los detalles del formato XML han sufrido cambios importantes al pasar de la versión 2.1 a la 2.5.

## Actualizaciones en lote

Las actualizaciones en caché de ADO se conocen con el nombre de *batch updates*, y para activarlas sólo necesitamos alterar una propiedad del conjunto de datos, *LockType*, de modo que valga *ltBatchOptimistic*. Para que el modo sea aceptado, el cursor debe estar en el lado cliente o, si está implementado en el servidor, debe pertenecer al tipo *ctKeyset*. Una vez que *LockType* ha sido modificada, todas las grabaciones se detienen en la caché local del cliente, y no se envían al servidor como es habitual.

Para hacer efectivas las modificaciones hay que ejecutar el método *UpdateBatch* del conjunto de datos ADO; si lo que pretendemos es cancelar los cambios, hay que ejecutar *CancelBatch*. A continuación mostraré un ejemplo de cómo pueden integrarse estos métodos en un cuadro de diálogo de edición:

```

procedure TdlgEdicion.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
begin
  if ModalResult = mrOk then
    ADOQuery1.UpdateBatch
  else if Application.MessageBox('¿Desea abandonar los cambios?',
    'Atención', MB_ICONQUESTION or MB_YESNO) = IDYES then
    ADOQuery1.CancelBatch
  else
    CanClose := False;
end;

```

He supuesto que el formulario *TdlgEdicion* contiene un componente *ADOQuery1* con una o más filas, y que de alguna manera se permite la edición de varias de las filas. El formulario se muestra mediante *ShowModal*, y ofrece los típicos botones para cancelar y aceptar. La grabación se produce durante la respuesta al evento *OnCloseQuery*. Si ocurre algún error durante la misma, el cuadro de diálogo permanece activo en la pantalla y nos permite que hagamos correcciones o que abandonemos cobardemente la edición.

Al igual que sucede con el BDE, existen graves problemas cuando se usan actualizaciones en caché con relaciones maestro/detalles. Los problemas se agravan porque en ADO, a diferencia del BDE, la grabación de la caché es una operación monolítica, y se hace imposible coordinar el estado de las dos cachés. No es un problema para desesperarse, como ya sabe, porque *DataSnap* puede corregirlo.

Si se viese obligado a usar actualizaciones en caché con ADO, quizás le vendría bien echar un vistazo al controlador *MSDataShape* para OLE DB. Se trata de un controlador que se “monta” sobre una conexión física y permite enviar al servidor, ya sea Access o SQL Server, instrucciones como la siguiente:

```

shape      { select * from CUSTOMER }
append    ( { select * from ORDERS }
            relate CustNo to CustNo) as ORDERS

```

El resultado de esta consulta es una tabla con las mismas columnas que la tabla de clientes, más una columna adicional llamada *Orders*, del tipo *TDataSetField*. Si traemos un componente *TADODataset*, podemos asignar ese campo en su propiedad *DataSetField*, para tener acceso directo a los registros de detalles. Como comprenderá, se trata del mismo mecanismo que utiliza *MyBase* para representar entidades complejas.

Si activamos las actualizaciones en lote para una consulta de este tipo, con tablas anidadas, se puede lograr que el mecanismo de caché respete la integridad de la relación entre las dos tablas bases.

El controlador *Data Shape* tiene algunas características adicionales muy interesantes. Podemos, por ejemplo, definir campos de estadísticas como en MyBase, pero utilizando la sintaxis extendida que hemos mencionado antes. Se puede agrupar el resultado de una consulta, sin perder los registros originales. Y, naturalmente, se pueden utilizar parámetros en la definición de la relación.

**ADVERTENCIA**

Para trabajar con *MSDataShape* debe asegurarse de tener, al menos, la versión 2.6 de ADO, para evitar algunos *bugs* existentes en la versión 2.5. Sepa también que la versión de ADO Express en Delphi 5 presenta algunos problemas al utilizar este controlador OLE DB.



# 5

## **DataSnap**

---

- **Proveedores (I)**
- **Proveedores (II)**
- **Resolución**
- **Servidores de capa intermedia**

# Parte





## Proveedores (I)

COMO LAUREL Y HARDY, COMO EL MAR Y LA ARENA, o como la tortilla y la patata, así se comportan DB Express y los conjuntos de datos clientes. Muy pocas aplicaciones interactivas pueden programar utilizando solamente DB Express; no es una limitación, sino una característica “expresamente” diseñada por Borland. En este capítulo daremos el primer paso para casar ambos sistemas, explicando el uso de los *proveedores* (*providers*), componentes que sirven de enlace entre un conjunto de datos de cualquier interfaz de acceso y los conjuntos de datos clientes.

Hay dos asuntos que pospondremos: la actualización de información utilizando proveedores, porque solamente estudiaremos la recuperación de registros, con todas sus variantes, y la creación de aplicaciones físicamente divididas en más de dos capas. Para simplificar la presentación, asumiremos en casi todos los ejemplos que los componentes de acceso de origen y los conjuntos de datos clientes residen en la misma aplicación.

### La conexión básica

Comencemos por la configuración más sencilla posible. Todos estos experimentos iniciales los realizaremos colocando todos los componentes en una misma aplicación, o “capa”. Cuando sea necesario, explicaré cuáles objetos debería ir en cada capa si estuviésemos escribiendo una aplicación en varias capas. Utilizaré InterBase como sistema de bases de datos SQL, y accederemos a él a través de DB Express.

Cree una nueva aplicación y un módulo de datos dentro de la misma. Llámelo *mod-Datos*, para uniformizar las referencias al mismo en el código fuente. Deje caer en el módulo un componente *TSQLConnection* y un *TSQLQuery*. Estableceremos una conexión con la popular base de datos *mastsql.gdb*. Después de conectar la consulta con el *TSQLConnection1*, teclee la siguiente instrucción en su propiedad *SQL*:

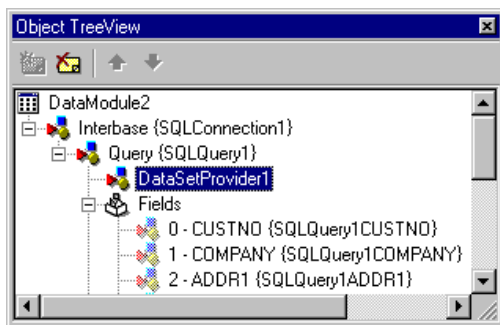
```
select * from CUSTOMER
```

Debemos crear los componentes de acceso a campos de la consulta y configurarlos como si se tratase de una aplicación real. Voy a mencionar solamente algunas de las modificaciones posibles, según el campo:

- *CustNo*: Cambie *DisplayLabel* a '*ID cliente*'; en realidad, debemos retocar las *DisplayLabel* de todos los campos. El tipo del campo es *TFloatField*, pero en la práctica solamente contiene valores enteros. Es conveniente cambiar *DisplayFormat* a la cadena '*0,*' para introducir un separador de millares; estamos forzados entonces a modificar también *EditFormat*, con el valor '*0*' (las comillas no cuentan). Observe que la propiedad *Required* se ha puesto automáticamente en *True*, porque el campo no admite nulos. ADO Express, en su versión actual, no se porta igual de bien con nosotros.
- *TaxRate*: He modificado *DisplayLabel* a '*Impuestos*', *DisplayFormat* a '*0.00%*', para mostrar siempre dos decimales, y *EditFormat* a '*0.00*', para que el signo del porcentaje desaparezca al editar el valor.
- *LastInvoiceDate*: La etiqueta de visualización debe ser '*Ultima factura*', y puede cambiar *DisplayFormat* a '*dd/mm/yyyy*'. Personalmente, siempre asigno *alRightJustify* en la propiedad *Alignment* de los campos de tipo fecha.

Una advertencia: compruebe que la consulta quede cerrada en tiempo de diseño. No es que sea una condición necesaria para el comportamiento de la aplicación, pero si dejamos consultas activas en diseño con aplicaciones reales, estaremos desaprovechando una interesante propiedad del trabajo con proveedores que explicaré más adelante.

Para terminar la primera mitad del experimento, vaya a la página *Data Access* de la Paleta de Componentes y seleccione un *TDataSetProvider*. Puede dejarlo caer sobre la ventana *Object TreeView*, sobre el componente *SQLQuery1*, como se muestra en la siguiente imagen:



De esta forma, la propiedad *DataSet* del proveedor recibe como valor un puntero al componente de consulta. Por supuesto, si lo prefiere puede añadir sin más el componente sobre el módulo de datos, y configurar entonces el valor de *DataSet*. Además de esa propiedad del proveedor, debe modificar estas otras:

- 1 *Name*: Le daremos un nombre “decente”, como *prClientes*. Luego veremos por qué me ha dado lo mismo el nombre de la consulta y, sin embargo, me pongo tan quisquilloso con el del proveedor.

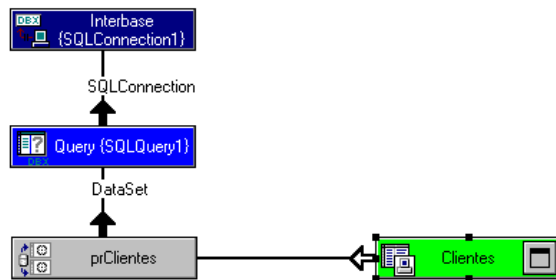
- 2 *ProviderFlags*: Esta propiedad contiene un conjunto de opciones. Añada la opción *poIncFieldProps* (incluir propiedades de campos).

En la práctica, hay más propiedades que se deberían modificar, y más opciones útiles; todo a su tiempo.

Hasta aquí, todos los componentes que hemos añadido deberían ir en la capa intermedia de una aplicación multicapas. El siguiente componente que añadiremos iría normalmente en la capa final, en la aplicación del usuario interactivo.

Ha llegado el momento de la verdad: en la misma página *Data Access*, localice un *TClientDataSet* y añádalo al módulo de datos. Bauticelo como *Clientes* y despliegue el combo de valores de su propiedad *ProviderName*. Verá que aparece *prClientes*, el proveedor que hemos configurado un poco antes. Por último, haga doble clic sobre *Clientes* para que aparezca el Editor de Campos, pulse el botón derecho del ratón sobre el mismo y ejecute el comando de menú *Add all fields*.

El siguiente diagrama ha sido creado arrastrando nuestros cuatro componentes sobre la vista *Diagram* del módulo de datos, y reordenando un poco los componentes:



## Duplicación del esquema relacional

Como comprenderá, el conjunto de datos cliente ha copiado el esquema relacional del componente *SQLQuery1* gracias a la ayuda del proveedor. Sin embargo, si seleccionamos uno cualquiera de los campos del conjunto de datos cliente, comprobaremos que no se han copiado las propiedades que tan trabajosamente habíamos cambiado en los campos de la consulta. No se desanime: active y desactive el conjunto de datos cliente y vuelva a examinar las propiedades de los campos: ¡esta vez sí se han copiado los valores desde los campos de la consulta!

Esto ha sido posible porque hemos añadido la opción *poIncFieldProps* a la propiedad *Options* del proveedor. Para ser exactos, sólo necesitamos esa propiedad para duplicar inicialmente las propiedades de los campos del conjunto de datos de origen. Pero prefiero que *poIncFieldProps* se quede activa definitivamente. Así, cuando realice algún probable cambio en las propiedades de un campo de la consulta, me bastará activar y desactivar el conjunto de datos cliente para propagar los cambios.

Es fácil darse cuenta de que dejar la propiedad activa en tiempo de ejecución nos hará perder un poco de tiempo cuando se abra cada *TClientDataSet*. El tiempo perdido es insignificante, en realidad, aunque si la apertura y cierre es repetitiva, el tiempo total perdido puede llegar a ser notable. Pero lo más importante es que la transmisión de esas propiedades en una aplicación en tres capas físicas aumentará el tráfico en la red. La solución es muy sencilla, y ya debe estar familiarizado con ella por ejemplos anteriores: podemos retocar la configuración de los proveedores en el momento de la carga del módulo, sobrescribiendo el método virtual *Loaded* del formulario o del módulo de datos.

```
procedure TmodDatos.Loaded;
begin
  inherited Loaded;
  for I := 0 to ComponentCount - 1 do
    begin
      C := Components[I];
      if C is TDataSetProvider then
        TDataSetProvider(C).Options :=
          TDataSetProvider(C).Options - [poIncFieldProps];
    end;
  end;
end;
```

De acuerdo con la documentación, las propiedades de campos que se propagan son: *Alignment*, *DisplayLabel*, *DisplayWidth*, *Visible*, *DisplayFormat*, *EditFormat*, *MaxValue*, *MinValue*, *Currency*, *EditMask* y *DisplayValues*. Por desgracia, se trata de otro desfase informativo, porque se propagan muchas otras propiedades. He aquí algunas diferencias respecto al comportamiento documentado:

- Se olvidan de una propiedad muy importante: *Required*, que indica si un campo admite nulos o no. La propagación ocurre en Delphi 5 y 6.
- Con *DefaultExpression* ocurre algo interesante: siempre se propaga, independientemente del valor de la opción *poIncFieldProps*. Sin embargo, el valor de *DefaultExpression* no parece copiarse, en tiempo de diseño, en el campo correspondiente del conjunto de datos cliente. No se alarme, porque eso es normal; forma parte de algo que Borland bautizó pomposamente en Midas 1 como *Constraints Broker*. Tenga cuidado, porque en Delphi 5 hay problemas con esta propiedad con los campos de clase *TBooleanField*.
- Algo similar sucede con *CustomConstraint* y *ConstraintErrorMessage*, que se propagan siempre, con independencia del estado de las opciones del proveedor.

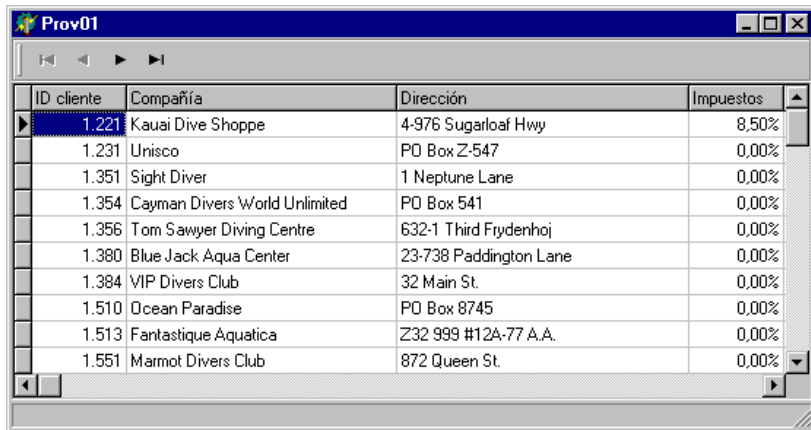
No se angustie ahora intentando memorizar todas estas propiedades. Cuando estudiemos el proceso de grabación de cambios volveremos a estudiar las propiedades de los campos que son aprovechadas por los proveedores.

## Robo de información

La falta de vergüenza de un conjunto de datos cliente llega más lejos. Regrese a la ventana principal de la aplicación y, mediante el comando de menú *File|Use unit*,

incluya la unidad *Datos* en la cláusula **uses** de la unidad *Principal*. Añada un componente *TDataSource*, llámelo *dsClientes* y asigne *modDatos.Clientes* en su propiedad *DataSet*. Recuerde que prefiero que los *TDataSource* utilizados por controles de datos estén en la misma unidad que estos. Los únicos componentes de fuentes de datos que deben estar en un módulo de datos son los utilizados para establecer relaciones maestro/detalles.

A continuación, añada una rejilla de datos y una barra de navegación, y conéctelas a *dsClientes*. Regrese entonces al módulo de datos y asegúrese de que *Clientes*, el conjunto de datos cliente, esté abierto en tiempo de diseño. Cuando vuelva a examinar la ventana principal, notará que ésta ya muestra la información sobre los clientes de la base de datos de ejemplo:



ID cliente	Compañía	Dirección	Impuestos
1.221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy	8,50%
1.231	Unisco	PO Box Z-547	0,00%
1.351	Sight Diver	1 Neptune Lane	0,00%
1.354	Cayman Divers World Unlimited	PO Box 541	0,00%
1.356	Tom Sawyer Diving Centre	632-1 Third Frydenhoj	0,00%
1.380	Blue Jack Aqua Center	23-738 Paddington Lane	0,00%
1.384	VIP Divers Club	32 Main St.	0,00%
1.510	Ocean Paradise	PO Box 8745	0,00%
1.513	Fantastique Aquatica	Z32 999 #12A-77 A.A.	0,00%
1.551	Marmot Divers Club	872 Queen St.	0,00%

No creo que esto le sorprenda o le parezca extraordinario. Pero si ha seguido mis instrucciones, y ha dejado antes la consulta *SQLQuery1* inactiva en tiempo de diseño, sí que podrá sorprenderle lo siguiente:

- A pesar de que *Clientes* está activo, aparentemente *SQLQuery1* se mantiene cerrado todo el tiempo. Aunque abra y cierre varias veces el conjunto de datos cliente, la consulta original parece seguir cerrada, aunque está claro que los datos que muestra el primer conjunto de datos provienen de ella.

¿Investigamos? He añadido un componente *RichEdit1* al formulario principal, lo he alineado a la izquierda y he añadido un *Splitter1* para controlar el tamaño ocupado por el control. Puede consultar los detalles gráficos en el código fuente que se encuentra en el CD; no es lo más importante ahora.

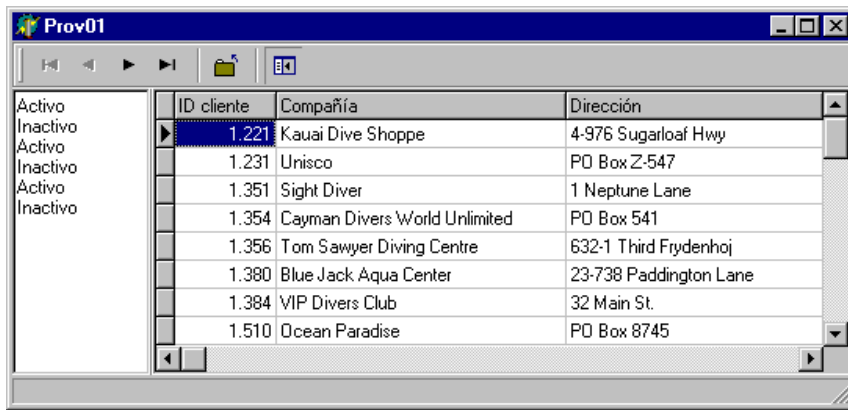
He creado también una acción *acOpenClose*; si no está muy ducho en el trabajo con acciones, puede utilizar un botón o un comando de menú, sin más, siempre que le asocie un manejador de eventos como el siguiente, que abre o cierra el conjunto de datos cliente que se encuentra en el módulo de datos.

```

procedure TwndPrincipal.acOpenCloseExecute(Sender: TObject);
begin
    with modDatos.Clientes do
        Active := not Active;
end;

```

El resultado se puede observar en la siguiente imagen. Ese botón con una carpeta cerrada es el que cierra o abre el conjunto de datos cliente; en la imagen, el componente está activo, y la imagen del botón es una carpeta cerrada, para indicar que podemos cerrar *Clientes*. El último botón de la derecha es un añadido estético, para ocultar o mostrar el panel de la izquierda.



Luego he añadido otro *TDataSource* en el formulario principal. Lo he llamado *dsConsulta* y lo he asociado al componente *SQLQuery1* del módulo de datos. Mi objetivo es poder “espíar” los cambios de estado de la consulta; note que hablo de la *consulta*, no del conjunto de datos cliente, que vamos a abrir y cerrar dinámicamente. Los cambios de estado se pueden interceptar creando un manejador para el evento *OnStateChange* de la fuente de datos:

```

procedure TwndPrincipal.dsConsultaStateChange(Sender: TObject);
begin
    case dsConsulta.State of
        dsInactive: RichEdit1.Lines.Add('Inactivo');
        dsBrowse: RichEdit1.Lines.Add('Activo');
    else
        RichEdit1.Lines.Add('Otro estado');
    end;
end;

```

Esto es lo que pasa cuando realizamos el experimento:

- 1 El cierre del conjunto de datos cliente no afecta en lo más mínimo al estado de la consulta original.
- 2 En cambio, cada vez que abrimos el conjunto de datos cliente la consulta se abre, todos los registros se leen (esto no se deduce del experimento, pero se lo digo

- yo) y finalmente se vuelve a cerrar. Aunque *Cientes* queda activo, *SQLQuery1*, que es quien se conecta directamente a InterBase, permanece cerrado.
- 3 Puede observar también que los cambios que hagamos sobre la rejilla no afecta en absoluto a la consulta. Es cierto también que al final todos los cambios son ignorados: al abrir y cerrar, comprobamos que los registros originales no han sido modificados.

Resumiendo, que cuando utilizamos la configuración básica con un proveedor en medio, el conjunto de datos cliente lee *todos* los registros del conjunto de datos original para almacenarlos en su caché interna (la propiedad *Data*). E inmediatamente cierra el conjunto de datos de origen.

## Coge el dinero y corre

Aunque le cueste creerlo al principio, el hecho de que un proveedor configurado en su forma básica recupere todos los registros de una consulta y luego la desactive, es muy importante, y marca una diferencia importante respecto a la forma en que normalmente se programaban las aplicaciones cliente/servidor en dos capas.

Consideremos, por poner un ejemplo, una aplicación basada en el BDE, que no aprovecha proveedores ni conjuntos de datos clientes, en ninguno de sus sabores. Pongamos por caso que queremos saber qué clientes tenemos en la ciudad de Despeñaperros. Pillamos un componente de consulta, plantamos la instrucción SQL en su interior, le echamos agua, la agitamos y la conectamos con un *TDataSource* a una rejilla de datos. En tiempo de ejecución, la consulta se abre y los datos van pasando, fila a fila, según van siendo necesarios, del lado servidor al lado cliente. Hay 300 clientes en esa ilustre ciudad, aunque la rejilla solamente los muestra de 20 en 20. Si el usuario de la aplicación se queda en la primera o segunda página, es evidente que la consulta debe mantener abierto el cursor en la base de datos para, cuando el usuario salga de su letargo, seguir trayendo datos.

Un cursor abierto consume muchos recursos, principalmente memoria RAM, pero también es posible que ocupe espacio en disco. Peor aún: de acuerdo al tipo de cursor y al nivel de aislamiento de la transacción dentro de la cual se abre el cursor, puede que los registros queden bloqueados, al menos en modo de lectura. Note que no estoy hablando de “conexiones”, sino de cursores abiertos por esas conexiones. Esta es la causa de los principales problemas que se presentan cuando el número de usuarios de una aplicación crece más allá de cierto umbral.

Pero si no hubiera una alternativa a la técnica anterior, no tendría mucho sentido quejarse. Una de las alternativas es, como puede imaginar, el uso de memoria caché en el cliente para terminar la recuperación de datos en el menor tiempo posible, que es la función de nuestros conjuntos de datos clientes. Naturalmente, sólo es factible aplicar esta técnica cuando las consultas devuelven un número razonable de registros. ¿Cuál es un número razonable? No puedo darle una respuesta “definitiva”: hace un

par de años, de 150 a 200 registros era el tope en los sistemas con los que trabajaba entonces; por supuesto, influía mucho el tamaño de cada registro. Estoy hablando de procesadores a 500MHz, memoria de hasta 512MB en los servidores, redes locales con 10/100Mbps, y conexiones remotas dedicadas a 64Kbs. El equivalente actual son los servidores con 1.5GHz o incluso 2GHz de velocidad y, de forma destacada, conexiones DSL que alcanzan varios millones de baudios de velocidad. La misma aplicación puede servir de 800 a 1.000 registros, sin que el usuario note retrasos significativos. Quién sabe cuál será el límite dentro de un año...

## Parámetros en las consultas

Cuando hablamos de “limitar el número de registros”, el primer mecanismo que debe venirnos a la cabeza es el uso de consultas con parámetros modificables en tiempo de ejecución. ¿Pueden los proveedores manejar este tipo de consulta? ¡Por supuesto que sí!

Copie, si lo desea, el proyecto anterior a un nuevo directorio, y cambie el nombre del fichero raíz de proyecto para no confundirse. En el módulo de datos, abra el editor de la propiedad *SQL* de la consulta de clientes, y sustituya la instrucción existente por esta otra:

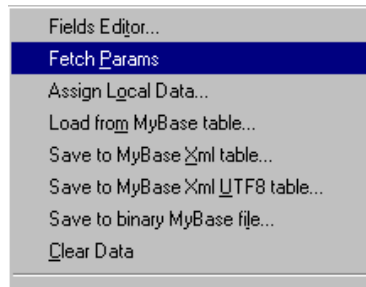
```
select *
from   CUSTOMER
where  Company starting with :patron
```

Como la interfaz de acceso primaria es DB Express, estamos condenados a abrir el editor de la propiedad *Params* de la consulta, para asignar explícitamente la constante *fi.String* en la propiedad *DataType* del único parámetro.

¿Hay algún problema si la consulta inicial tiene parámetros? Sí, aunque puede que no nos demos cuenta en estos ejemplos sencillos que mezclan todas las capas físicas dentro de una aplicación. La dificultad consiste en que estamos acostumbrados a asignar los valores dinámicos de los parámetros utilizando directamente el componente de consulta. Pero si estamos desarrollando una aplicación en tres capas, la asignación deberá programarse en la capa que maneja el usuario final, que es la que contiene solamente los conjuntos de datos clientes. Por lo tanto, debemos aprender cómo asignar valores de parámetros a un *TClientDataSet* para que el proveedor los transmita a la consulta. Y vamos a comprobar que es algo muy sencillo.

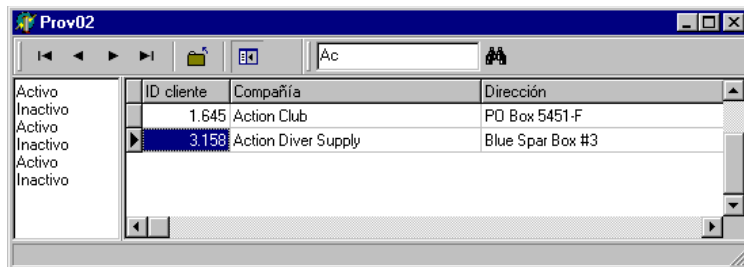
Primer paso: si antes duplicamos el esquema relacional de la consulta en el conjunto de datos cliente, ahora haremos lo mismo con la colección de parámetros. Hay que pulsar el botón derecho del ratón sobre el conjunto de datos cliente, para que aparezca el siguiente menú de contexto:





El comando que debemos ejecutar es *Fetch Params*. Hágalo, y compruebe entonces que el valor de la propiedad *Params* del componente *Clientes* es idéntico al de la misma propiedad de *SQLQuery1*. Incluso, si hubiéramos asignado valores iniciales a los parámetros en la consulta, se habrían copiado dichos valores. No hay que volver a ejecutar este comando, excepto si modificamos la consulta y cambia su colección de parámetros. De todos modos, creo que le interesará conocer que el mismo efecto se puede lograr en tiempo de ejecución llamando al método *FetchParams* del conjunto de datos cliente.

Nos queda solamente asignar valores a los parámetros de *Clientes* en tiempo de ejecución. Observe la siguiente imagen:



He añadido un cuadro de texto, *edSearch*, y una acción, *acSearch*, que he asociado con un botón de la barra de herramientas. Cuando se pulsa el botón, se ejecuta el método asociado al evento *OnExecute* de la acción, que es el siguiente:

```
procedure TwndPrincipal.acSearchExecute(Sender: TObject);
begin
  modDatos.Clientes.Close;
  modDatos.Clientes.Params[0].AsString := edSearch.Text;
  modDatos.Clientes.Open;
end;
```

Por supuesto, las variaciones son infinitas, aunque todas equivalentes. El código hace referencia al parámetro por su posición; está claro que es más seguro que la referencia utilice su nombre. Debe tener en cuenta, sin embargo, que aunque el tipo de *Params* es el mismo tanto en *TClientDataSet* como en *TSQLQuery*, este último componente define una función *ParamByName* que nos ahorra la referencia a *Params*, mientras que los conjuntos de datos clientes carecen de ella:

```

SQLQuery1.Params.ParamByName('patron')...           // CORRECTO
ClientDataSet1.Params.ParamByName('patron')...       // CORRECTO
SQLQuery1.Params.ParamByName('patron')...           // CORRECTO
ClientDataSet1.Params.ParamByName('patron')...       // ; INCORRECTO!

```

### UN CONSEJO

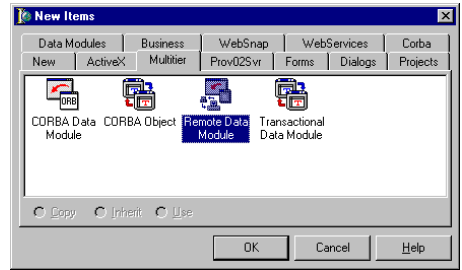
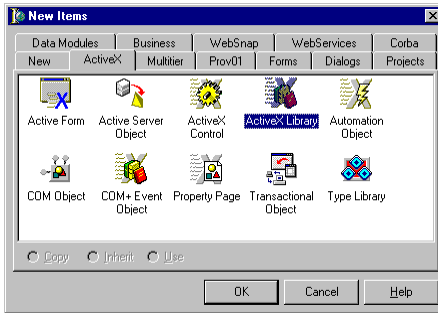
Es perfectamente posible implementar la sustitución de parámetros anterior cambiando el valor de *patron* directamente en la consulta, que por deducción sabemos que estará cerrada en tiempo de ejecución. Luego, sólo tendríamos que cerrar *Clientes* y volver a activarlo. Pero sería un disparate como metodología; si quisiéramos dividir posteriormente la aplicación en dos capas, se nos presentaría un problema grave. Una buena regla a seguir cuando se mezclan capas, tal como estamos haciendo, es que el código relacionado con la interfaz de usuario no debe jamás hacer referencia al sistema primario de acceso a datos; en este caso, los componentes DB Express.

## Mi primera aplicación en tres capas

Mucho hablar sobre aplicaciones en tres capas pero... ¿qué tal si desarrollamos una, aunque sea muy sencilla? Estoy convencido de que todas esas consideraciones abstractas sobre “esto se puede hacer, esto otro es pecado mortal” tendrán más sentido para usted si puede comprobar, de primera mano, qué es lo que sucede realmente cuando se abren las aguas del Mar Rojo ... perdón, cuando el conjunto de datos clientes y la consulta DB Express van a parar a módulos físicos diferentes.

Lo que haremos es situar el módulo de capa intermedia en el tipo más sencillo y eficiente de servidor COM: en una DLL. Mi tesis es que si vamos a desarrollar con DB Express y necesitamos utilizar el sistema de caché de *TClientDataSet*, que será lo más frecuente, merece la pena que hagamos un pequeño esfuerzo adicional, y separemos entonces los componentes DB Express junto con sus proveedores asociados en un servidor COM implementado como una DLL. Por una parte, no perderemos nada importante: la instalación de la DLL será muy sencilla, y no tendremos que pagar un precio significativo en velocidad, porque en definitiva todos los componentes estarán en el mismo espacio de proceso. Por la otra, nuestra aplicación estará automáticamente lista para migrar a un sistema real en tres capas, al ser elemental la transformación del servidor dentro del proceso en un servidor remoto.

Primero, deberá crear una DLL preparada para hospedar una co-clase. No nos vale una DLL cualquiera, pues se trata de implementar un servidor COM. Ejecute el comando de menú *File|New*, seleccione la segunda página en el Almacén de Objetos, y haga doble clic sobre el icono *ActiveX Library*:



Ya tenemos el continente, pero aún nos falta el contenido. Vuelva al Almacén de Objetos, esta vez a la tercera página, y pulse sobre el icono *Remote Data Module*. Esta acción creará una clase dentro de la DLL con soporte para automatización OLE, por lo que nos aparecerá el siguiente cuadro de diálogo:



Vamos a llamar *Datos* a la nueva clase. No debe preocuparse por darle un nombre tan vulgar; recuerde que a este nombre se le añade como prefijo el nombre de la DLL, para generar el identificador de programa. Si sigue mi sugerencia y guarda el proyecto con el nombre de *Prov03Svr*, obtendremos *Prov03Svr.Datos* como nombre final de la clase de componentes.

En este momento, debe tener frente a usted un módulo de datos con apariencia completamente “normal”. Sin embargo, si mira en la declaración de su clase, encontrará las diferencias:

```

type
  TDatos = class (TRemoteDataModule, IDatos)
  private
    { Private declarations }
  protected
    class procedure UpdateRegistry(Register: Boolean;
      const ClassID, ProgID: string); override;
  public
    { Public declarations }
  end;

```

Ya tendremos tiempo para detenernos en detalles. Lo que haremos ahora es copiar en el módulo los componentes *SQLConnection1*, *SQLQuery1* y *prClientes* del proyecto anterior. Asegúrese de que la consulta está inactiva. Guarde todo el proyecto en

disco, compílelo y ejecute entonces el comando de menú *Run | Register ActiveX Server*, para que la clase quede registrada en ese ordenador. Y ya está listo nuestro primer servidor de capa intermedia.

Ahora vamos al cliente. Cree una aplicación normal; para abreviar, ni siquiera hará falta un módulo de datos. Vaya a la página *Data Snap* de la Paleta de Componentes y traiga un componente *DCOMConnection1* al formulario. Despliegue el combo asociado a su propiedad *ServerName*. Debe aparecer, entre otros, el servidor que acabamos de registrar: *Prov03Svr.Datos*. Si no es así, es que ha olvidado registrarlo en el paso anterior. Seleccione ese servidor, e inmediatamente verá cómo el propio componente asigna un valor en su propiedad *ServerGUID*; evidentemente, ha leído el Registro de Windows para encontrar el GUID asociado al identificador de programa.

Entonces debe cambiar el valor de la propiedad *Connected* a *True*. No se encenderán fuegos artificiales, pues el servidor es una DLL que se carga silenciosamente dentro de la aplicación activa... que en ese preciso momento es el Entorno de Desarrollo de Delphi. La señal más fácil de encontrar de que el servidor está cargado es que InterBase debe reportar una conexión activa, si es que ha dejado activo el componente *SQLConnection1* (era *SQLQuery1* el que había que dejar cerrado).

Podemos entonces traer un *TClientDataSet* al formulario. Desplegamos el combo de su propiedad *RemoteServer*, y seleccionamos *DCOMConnection1*. Luego desplegamos el combo de la propiedad *ProviderName*. Verá cómo aparece el único proveedor existente en el servidor de capa intermedia: *prClientes*. Selecciónelo, y compruebe que puede abrir y cerrar el conjunto de datos. De ahí en adelante, podrá hacer con el conjunto de datos cliente lo que se le antoje.

## Instrucciones a la medida

Hay algo en la anterior aplicación con la que ilustré el manejo de parámetros que algunos calificarían como *bug*, y otros como *característica*. Como recordará, la sentencia de la consulta utilizaba el operador **starting with**, específico de InterBase. El motivo para recurrir a este operador es que InterBase no puede optimizar el uso del operador **like**, mucho más versátil, cuando el operando derecho es un parámetro. Sin embargo, esta elección provoca que, cuando pasamos una cadena vacía como patrón, la consulta no devuelva registro alguno. Esa es la definición del funcionamiento de **starting with**; que a usted o a mí nos guste, es harina de otro costal. ¿Podríamos evitarlo, y lograr que se devuelvan *todos* los registros si se usa la cadena vacía como prefijo? No, al menos sin renunciar al eficiente comportamiento de este operador.

Lo que pretendo explicar es que no siempre podemos implementar una búsqueda sólo con añadir parámetros a una consulta. Hay un ejemplo mucho más ilustrativo de este problema: la búsqueda por palabras claves. El usuario teclea en un cuadro de edición una lista de palabras claves, separadas por espacios. Si el usuario suministra dos palabras, la instrucción SQL de búsqueda será muy diferente de la que sería ne-

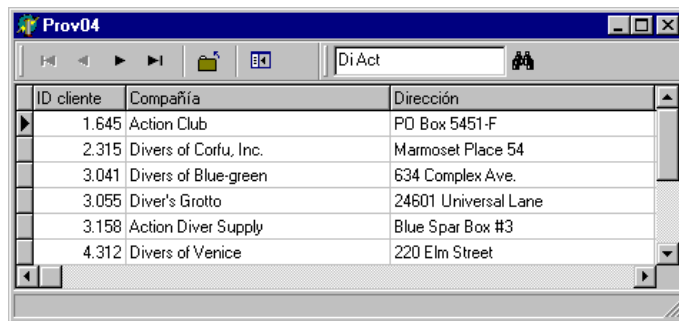
cesaria para cinco palabras. Quizás podríamos diseñar una superinstrucción hipergénica y megacapaz, que contemplase todos los casos posibles, pero es altamente probable de que funcionase de pena. Por lo tanto, se suele implementar este tipo de búsqueda generando al vuelo toda la sentencia SQL.

Ahora podemos identificar la dificultad: en un sistema de tres capas, solamente tenemos a mano, en el lado cliente, el componente *TClientDataSet*. ¿Cómo arreglárnoslas para modificar la instrucción SQL del componente remoto de consulta? Borland pensó en este problema y añadió la propiedad *CommandText* en los conjuntos de datos clientes con este propósito. De acuerdo con la configuración de las opciones del proveedor, si el conjunto de datos cliente tiene algo asignado en *CommandText*, el proveedor fuerza a la consulta de origen a que ejecute el contenido de *CommandText* en vez de su propia instrucción.

Para mostrar cómo funciona el mecanismo explicado, vamos a simplificar un poco el planteamiento. Dejaremos la implementación de la búsqueda por palabras claves para la sección sobre Internet, y nos conformaremos ahora con un enunciado más modesto. ¿Recuerda la aplicación de búsqueda de clientes por medio de un parámetro? Extenderemos su funcionalidad, de modo que el cliente pueda teclear una serie de variaciones del nombre de la empresa en la condición de búsqueda. Por ejemplo, podríamos teclear lo siguiente:

Dive Action Deep

Y la aplicación debería entender que queremos buscar las empresas cuyos nombres comienzan con *Dive*, o con *Action*, o con *Deep*. Para no empezar desde cero, partiremos desde una copia de la aplicación que implementa la búsqueda por parámetros.



Los pasos para transformar la aplicación son los siguientes:

- 1 Seleccione el proveedor *prClientes*, y expanda su propiedad *Options* en el Inspector de Objetos. Añada la opción *po.AllowCommandText*, para que el proveedor pueda copiar el *CommandText* del conjunto de datos clientes en la propiedad *SQL* de la consulta *SQLQuery1*.

- 2 Ya que mencionamos la consulta, haga doble clic sobre su propiedad SQL, para modificarla del siguiente modo:

```
select *
from   CUSTOMER
where  CustNo = -1
```

Nos dará lo mismo la cláusula **where** que utilizemos; su única función es que la consulta inicial no devuelva fila alguna. En cambio, hay que tener mucho cuidado con las columnas mencionadas en la cláusula **select**, puede deben coincidir en número, tipo y posición con las columnas de la consulta generada dinámicamente. En este caso, es fácil, pues se devuelven todas las columnas de la tabla *CUSTOMER*.

- 3 Vaya entonces al conjunto de datos cliente, e introdúzcase en su propiedad *Params*. Elimine el parámetro que va a encontrar, que ya es innecesario.

Ahora tenemos que escribir unas pocas líneas de código. *NextToken* es una función global que podemos definir en dónde mejor nos parezca- Su propósito es separar de la cadena de búsqueda tecleada los correspondientes elementos, asumiendo que hay espacios en blanco entre ellos:

```
function NextToken(var Line: string; out Token: string): Boolean;
var
  I, J: Integer;
begin
  I := 1;
  while (I <= Length(Line)) and (Line[I] = ' ') do
    Inc(I);
  J := I;
  while (J <= Length(Line)) and (Line[J] <> ' ') do
    Inc(J);
  Token := Copy(Line, I, J - I);
  Delete(Line, 1, J - 1);
  Result := Token <> '';
end;
```

*NextToken* será utilizada por *GenerateSQL*, otra función global que genera la instrucción SQL necesaria, y que definiremos a continuación:

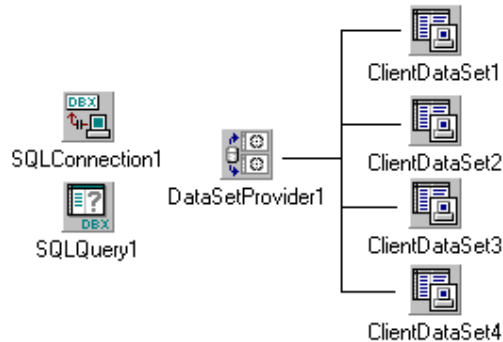
```
function GenerateSQL(const SearchString: string): string;
var
  Words, Word, Prefix: string;
begin
  Result := 'select * from CUSTOMER';
  Prefix := ' where ';
  Words := SearchString;
  while NextToken(Words, Word) do
    begin
      Result := Result + Prefix + 'Company starting with ' +
        QuotedStr(Word);
      Prefix := ' or ';
    end;
end;
```

Para terminar, nos queda modificar la respuesta al evento *OnExecute* de la acción *acSearch*, que se dispara cuando el usuario pulsa sobre el botón de búsqueda del formulario:

```
procedure TwndPrincipal.acSearchExecute(Sender: TObject);
begin
    modDatos.Clientes.Close;
    modDatos.Clientes.CommandText := GenerateSQL(edSearch.Text);
    modDatos.Clientes.Open;
end;
```

## Abusos peligrosos

Cuando un programador descubre la existencia de *CommandText*, por lo general termina abusando de él. Piense en una aplicación de capa intermedia que consista solamente en una consulta, sin instrucción asignada ni campos persistentes, y un proveedor con la opción *poAllowCommandText* activa. No hay nada que impida que varios conjuntos de datos clientes utilicen el mismo proveedor, especificando incluso distintas instrucciones en sus propiedades *CommandText*:



Resulta incluso que es muy cómodo trabajar de este modo; puedo atestiguarlo. Supongamos que hay un proveedor configurado de esta forma en la capa intermedia. Estoy trajinando sobre la aplicación del usuario final, cuando de repente me doy cuenta que necesito un conjunto de datos con los 10 productos más vendidos durante un período de tiempo. ¡Uf! Tendría que abrir el servidor de capa intermedia y añadir una consulta, configurarla, añadir un proveedor, conectarlo... ¡Qué demonios! Tiro un *TClientDataSet* con malos modos sobre el módulo de datos de la aplicación final, lo engancho a mi proveedor mágico, tecleo la instrucción SQL y ¡zas!, asunto concluido.

¿Qué hay de malo en ello? Mucho, estimado amigo: está llenando la capa final de la aplicación de sentencias SQL, escritas en el dialecto específico de su servidor SQL actual, y haciendo referencia a un esquema relacional que muy probablemente sufrirá cambios a lo largo del proceso de desarrollo. Cambia el servidor, o se añade soporte para un nuevo formato de base de datos, y hay que localizar y cambiar todas las ins-

trucciones. Se añade o se elimina un campo de una tabla, y vuelta a revisar la aplicación de arriba abajo. Lo peor de todo, es que es muy difícil localizar a simple vista los conjuntos de datos clientes que se están saltando el aislamiento de la capa intermedia.

### **NO DIRE EL NOMBRE...**

... pero hay un producto dentro de la categoría "sustitutos de Midas" que, al menos las versiones que he examinado, estimulaban una metodología de trabajo similar a la que acabo de criticar. Utilizarlo es más o menos equivalente a disponer de una pseudo interfaz SQL más potente y eficiente que las habituales. Pero nos hace muy dependientes del esquema relacional y del sistema SQL concreto que hay detrás.

## **Mentiras, mentiras cochinas y estadísticas**

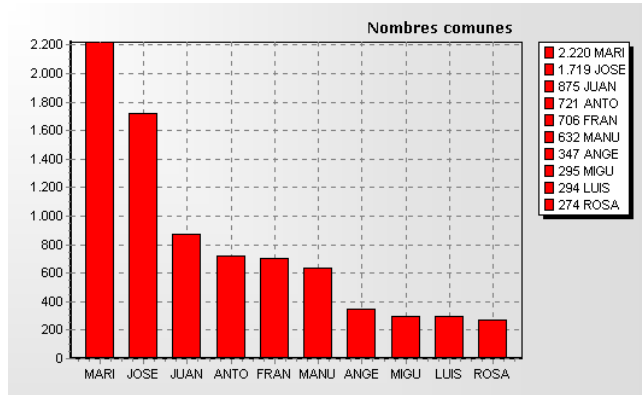
La forma de programar que estoy predicando consiste en lanzar peticiones breves al servidor, obtener *todos* los datos de la consulta, limpiar nuestras huellas y retirarnos como ladrones en la noche. Para que este estilo de vida funcione, no podemos ser demasiado avariciosos. Si intentamos llevarnos 500.000 registros de golpe, lo más probable es que nos pillen en la tarea. Por lo tanto, para que nos beneficiemos de este evangelio, hay que diseñar nuestras aplicaciones de modo que se basen en pequeños conjuntos de registros.

¿Hasta qué punto es razonable esperar que las aplicaciones reales se adapten a estas hipótesis? Sostengo, sin dudar, que siempre se puede exigir el cumplimiento de las mismas. El caso típico: tenemos un millón de clientes en nuestra base de datos. ¿Qué necesitamos, vender nuestros productos a uno de esos clientes registrados? Antes, localícelo. En vez de abrir un cursor con el millón de registros, pregunte su nombre a su cliente, y traiga a su aplicación un conjunto de resultados pequeño: los 200 primeros que se llamen de así... Vale, el cliente se llama José o María, y hay demasiados registros. Pregúntele entonces sus apellidos.

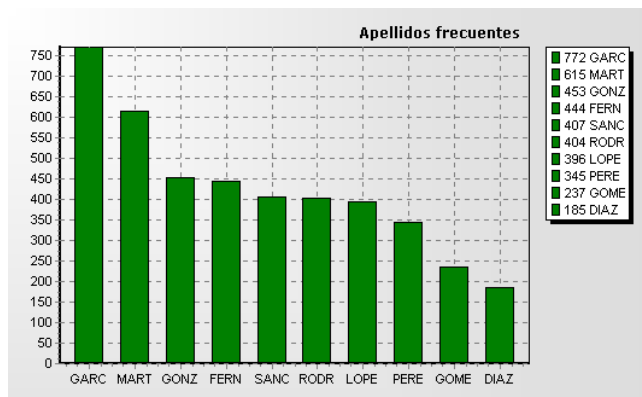
Tengo a mano una base de datos en SQL Server con datos reales sobre 17.489 personas de ambos sexos, principalmente españoles. Para estimar la frecuencia de la distribución de nombres y apellidos he creado una pequeña aplicación auxiliar, que muestra sus resultados en forma de gráfico (¡no está en el CD ROM, como podrá suponer, porque son datos reales!). En realidad, he realizado el análisis sobre las cuatro primeras letras de nombres y apellidos, ignorando mayúsculas y minúsculas, además de los acentos.

El siguiente gráfico corresponde a la distribución de nombres:





Como esperábamos, María y José se llevan el premio. Esta es la frecuencia de los apellidos, algo mejor repartidos:



En definitiva, que en la base de datos de 17.489 registros hay 91 personas llamadas Mari Garc. Extrapolando a un millón de registros, tendríamos en el peor de los casos unas 5.203 personas con el mismo nombre y apellidos. Pocos y manejables en la base de datos pequeña, pero demasiados en la del millón...

Pero no se asuste. En primer lugar, me he limitado a sólo cuatro letras. En el caso del nombre, hay que reconocer que la mayoría de las Marías son siempre Mari Algo<sup>27</sup>. Y nos queda algo muy importante: ¡el segundo apellido! En cuanto el segundo apellido entra en juego, el conjunto de resultados se reduce considerablemente. Si asumimos que la distribución de los segundos apellidos es similar a la de los primeros, mi calculadora me dice que tendríamos 247 registros de un millón en el caso más recalci-trante. Esos son pocos registros hoy, pero también hace tres años.

<sup>27</sup> La chiquilina de la dedicatoria se llama María, sin más, y es la nena más guapa del mundo entero y de parte de Andorra...

## Lectura incremental

No sé si lo habré convencido; sí estoy seguro de que un cliente nunca se dejará convencer. Por regla general, quien nos contrata siempre desconfía de argumentos matemáticos, piensa que las cosas irán el triple de mal, y en cualquier caso quiere que trabajemos diez veces más de lo que puede comprar su dinero.

Le digo a mi cliente: si quieres ver datos de un cliente, teclea algo para limitar el área a examinar. Si tecleas una soberana tontería, como pedir los datos de todas las marías de este mundo, debes saber que solamente te traeré las doscientas primeras filas. Mi cliente finge una expresión inteligente; él mismo se considera un tipo listo, pero sabe que la inteligencia es otra cosa. Y me espeta, en tono autoritario: “sí, pero quiero un botón para obtener los siguientes 200 registros”. Podría estrangularlo y luego dormir la siesta tranquilamente, pero necesito trabajar para comer... y de todos modos, hay una luz de esperanza.

El algoritmo de recuperación que hemos utilizado hasta ahora, leer todos los registros de golpe y cerrar la consulta de origen inmediatamente, se basa en los valores por omisión de un par de propiedades. La primera de ellas, *FetchOnDemand* vale inicialmente *True*, e indica que el conjunto de datos clientes se encarga de pedir sus datos al proveedor sin intervención alguna por parte nuestra. Si su valor fuese *False*, tendríamos que leer los datos explícitamente con ayuda de este método, cuyo estudio dejaremos para más adelante:

```
function TClientDataSet.GetNextPacket: Integer;
```

La segunda propiedad es *PacketRecords*, y vale  $-1$  por omisión. Indica el número de registros que se leen en cada petición de datos que se le haga al servidor. Si su valor es menor que cero, quiere decir que necesitamos todos los registros. Además, el valor especial 0 se utiliza para recuperar solamente el esquema relacional, sin registros.

Ahora explicaré qué sucede cuando esas propiedades tienen los siguientes valores:

Propiedad	Valor por omisión	Valor necesario
<i>FetchOnDemand</i>	<i>True</i>	<i>True</i>
<i>PacketRecords</i>	$-1$	20

*FetchOnDemand* conserva su valor por omisión; sólo hemos cambiado *PacketRecords*, asignándole un valor positivo. He elegido 20 registros porque es el número aproximado de registros que puede mostrar una rejilla en una página, en un espacio de pantalla de 800x600; si lo desea, puede experimentar con valores superiores. Cuando configuramos un *TClientDataSet* de esta manera, ocurre lo siguiente:

- 1 El conjunto de datos, al abrirse, pide al proveedor tantos registros como exija su *PacketRecords*. El proveedor abre la consulta a la que está asociado y lee esos re-

- gistros... pero, a diferencia de lo que hemos visto hasta el momento, no cierra la consulta a continuación.
- 2 Podemos desplazarse impunemente por el conjunto de registros recibido. No obstante, si intentamos sacar las patas del tiesto, desplazándonos más allá del último registro, se dispara otra petición automática al proveedor, si es que *FetchOnDemand* vale *True*.
  - 3 Cuando el proveedor recibe esa segunda petición y comprueba que la consulta está abierta todavía, asume además que el registro en que se encuentra el cursor es el primero que tiene que devolver. A partir de él, procesa el número de registros solicitados, para enviarlos al cliente.
  - 4 En algún momento, la consulta llegará al final. El proveedor se da cuenta y, junto con el paquete de los registros, envía esa información al cliente. A partir de ese momento, el conjunto de datos cliente sabe que, si llega al final, tiene que fastidiarse porque no hay más filas.

Agazapado dentro de la descripción anterior, hay un par de hechos muy importantes, y que lamentablemente pueden estropear el buen trabajo realizado hasta el momento por los conjuntos de datos clientes:

- 1 Mientras no se traigan todos los registros de la consulta al cliente, no se cerrará el cursor abierto sobre la base de datos. El tiempo que el cursor permanece activo depende exclusivamente del tiempo que quiera perder el usuario final mientras se recrea en la lectura de los datos (o mientras comenta el partido de fútbol con sus camaradas).
- 2 El algoritmo asume que, entre una llamada al proveedor y la siguiente, nadie utiliza esa consulta para nada más. La consulta mantiene su estado interno mientras tanto. Esto no es difícil de garantizar cuando cada instancia de la aplicación final tiene un servidor intermedio de datos independiente; pero impide que las instancias del servidor de capa intermedia sean reutilizadas. Se pierde entonces el atractivo de la técnica para sistemas con un número de usuarios muy grande.

Esta forma de trabajo existe desde la primera versión de Midas, la que apareció con Delphi 3. Naturalmente, no tiene sentido alguno recuperar datos de esta manera, que implica además la presencia de intermediarios sin utilidad aparente. Pero vamos a modificar un poco las cosas para eliminar los inconvenientes descritos.

## La variante explicada por Borland

Vamos a crear dos aplicaciones diferentes como ejemplos de lectura “a trozos”. Mediante la primera, explicaré la técnica que Borland recomienda seguir en la ayuda en línea. Y al final, cuando la aplicación esté terminada, le mostraré que no funciona correctamente. Luego explicaré la técnica que he estado utilizando en los últimos años, mucho más sencilla, pero que siempre va bien. No la he inventado yo, sino que me he basado en varios artículos de Dan Miser.

¿Está listo el nuevo proyecto, con un módulo de datos vacío? Comenzaremos, igual que hasta el momento, por los componentes de la presunta capa intermedia, que acceden a la capa SQL. Después de añadir el componente de conexión a la base de datos, configuraremos un componente *SQLQuery1* con la siguiente consulta paramétrica:

```
select *
from   ORDERS
where   OrderNo > :orderNo      /* Estrictamente mayor */
order by OrderNo asc
```

En vez de pedir todos los registros de la tabla, solicitaremos solamente los registros a partir de cierto valor de su clave primaria. Preste mucha atención: la consulta está ordenada por la clave primaria, *OrderNo*, y el valor de la columna debe ser *estrictamente mayor* que el del parámetro *orderNo*, que debemos marcar explícitamente como de tipo entero. Es muy importante lo de “estrictamente”, como veremos en breve.

Podríamos ordenar la consulta por cualquier otra combinación de campos, pero la explicación se complicaría innecesariamente. Dejo a su cargo la extensión de la técnica en estos casos.

A pesar de que la consulta tiene un parámetro, esta vez no vamos a controlar su valor desde el conjunto de datos cliente, sino que vamos a asignarlo según los resultados de la propia consulta en cada paso, utilizando manejadores de eventos disparados por el proveedor.

El primer evento que interceptaremos será *AfterGetRecords*, del objeto proveedor que asociamos a la consulta:

```
procedure TmodDatos.prPedidosAfterGetRecords(Sender: TObject;
var OwnerData: OleVariant);
begin
    SQLQuery1.Close;
end;
```

El objetivo del método anterior es cerrar la consulta después de cada petición. Así, el tiempo total que estará abierto el cursor SQL será mínimo. Pero al cerrar la consulta perdemos algo importante: la posición donde se encontraba el cursor, que ya sabemos que es necesaria para que el proveedor pueda recuperar el siguiente paquete de registros.

Esto nos va a obligar a interceptar el evento *BeforeGetRecords* del proveedor, la próxima vez que se reciba una petición de registros:

```
procedure TmodDatos.prPedidosBeforeGetRecords(Sender: TObject;
var OwnerData: OleVariant);
begin
    SQLQuery1.Params[0].AsInteger :=
        StrToIntDef(VarToStr(OwnerData), -1);
end;
```

Aquí nos basamos en una suposición: que el parámetro *OwnerData* contiene el valor de la clave primaria del último registro enviando en la anterior petición. ¿Cómo haremos para que *OwnerData* contenga ese valor? Paciencia, lo veremos enseguida. *OwnerData*, por su parte es de tipo *OleVariant*, y antes de convertirlo a entero, lo transformamos en una cadena de caracteres mediante la función *VarToStr*, que en Delphi 6 está declarada en la unidad *Variants* (¡hay que incluirla a mano!). Con esta conversión previa nos evitamos problemas cuando *OwnerData* contenga el valor variante especial *Null*. ¿Y por qué diablos puede tener ese valor? Si *OwnerData* contiene la clave del último registro, nos veremos en un brete la primera vez que se dispare este evento.

No tenemos que cerrar la consulta para que el parámetro surta efecto. Ya sabemos que la consulta estará cerrada, porque de ello nos hemos encargado al interceptar el evento *AfterGetRecords* del proveedor. Cuando el proveedor lo considere oportuno, además, se encargará de volverla a abrir.

¿Quién es el encargado de asignarle un valor a *OwnerData*? Es aquí donde hay una diferencia entre “mi” algoritmo y el de Borland. Los padres de la criatura opinan, al parecer, que el conjunto de datos cliente debe ocuparse del asunto, y que el mejor momento es justo antes de pedir el próximo grupo de registros. Hablando en plata, en el evento *BeforeGetRecords* del *TClientDataSet*:

```

procedure TmodDatos.PedidosBeforeGetRecords(Sender: TObject;
var OwnerData: OleVariant);
begin
  if Pedidos.Active then
    with TClientDataSet.Create(nil) do
      try
        CloneCursor(Pedidos, True);
        FetchOnDemand := False;
        Last;
        OwnerData := FieldByName('OrderNo').AsInteger;
        Close;
      finally
        Free;
      end
    else
      OwnerData := Null;
end;

```

¡Ja, ja! Espero verle fruncir las cejas, porque aquí hay un truco muy sucio de Borland. Primero lo más sencillo: si el conjunto de datos clientes realiza su primera petición es porque está cerrado. En ese caso, asignamos un nulo a *OwnerData* que, como sabemos, el proveedor interpretará como un -1. Como la clave primaria es siempre positiva, se recuperan los primeros registros de la tabla.

En caso de que el conjunto de datos ya esté activo, se crea un clon suyo. Sí, abra bien los ojos. Hay que saber cuál es el registro con clave más alta de los recibidos hasta el momento. Asumiendo que el conjunto de datos cliente está también ordenado por el número de pedido, ese registro debería ser el último. ¡Pero no se le ocurra llamar a *Last* sobre *Pedidos*, porque intentaría ir al último registro *real*! Además, movería inne-

cesariamente el cursor. De ahí que construyamos un clon, que comparte el mismo vector de datos pero nos permite utilizar una posición de cursor independiente. Además, cambiamos su propiedad *FetchOnDemand* a *False*, de modo que *Last* signifique solamente el último registro disponible. Copiamos la clave de ese registro en *OwnerData*, destruimos el clon, porque es sabido que los superjuguetes duran sólo hasta el fin del verano, y nos largamos del evento.

## Mi variante preferida

Ahora viene la parte más divertida: probar que la solución de Borland tiene serios problemas (póngase muy serio cuando lea la palabra “serio”). En el ejemplo del CD he añadido eventos para que en la barra de estado de la aplicación aparezca el total de registros leídos hasta el momento y la posición del registro actual. Verá que si nos desplazamos por los registros de uno en uno, hacia atrás y hacia delante, no encontraremos problema alguno. Intente, sin embargo, pulsar el botón de navegar directamente al final cuando aún no se hayan leído los 205 registros que tiene la tabla de pedidos. Comprobará que el conjunto de datos no llega a leer todos los pedidos existentes. ¿La explicación?, no la conozco. No he perdido mi tiempo buscándola. Sospecho, en todo caso, que se debe a alguna chorrada que haga internamente el método *Last* que entre en conflicto con el sistema de clonación de cursores.

### FALSA SOLUCION

Se nos podría ocurrir una alternativa sencilla: crear un campo de estadísticas (*aggregate field*) para que calcule por nosotros la clave máxima recuperada hasta el momento. ¿No era para obtener ese valor que creamos un clon? De hecho, en la copia del ejemplo anterior incluido en el CD he añadido dicho campo por si quiere experimentar con él. Pero le advierto que también falla cuando se llama al método *Last*. Si tiene tiempo, ponga un punto de ruptura en el evento *BeforeGetRecords* del conjunto de datos cliente. Durante la navegación secuencial, verá que todo funciona de maravillas. Ahora bien, cuando llame a *Last* podrá comprobar que nuestros componentes se desmadran. En particular, el campo que calcula el máximo nos dirá que su valor es nulo.

Es mi turno. Cree una nueva aplicación, añada un módulo de datos, una conexión DB Express a InterBase, y una consulta con la siguiente instrucción:

```
select *
from   ORDERS
where  OrderNo >= :orderNo    /* ;Mayor o igual! */
order by OrderNo asc
```

Note que ahora utilizamos la comparación mayor o igual, en vez de la comparación estricta. Muy pronto comprenderemos por qué.

La respuesta al evento *BeforeGetRecords* del proveedor es la misma de antes:

```
procedure TmodDatos.prPedidosBeforeGetRecords(Sender: TObject;
var OwnerData: OleVariant);
```

```

begin
    SQLQuery1.Params[0].AsInteger :=
        StrToIntDef (VarToStr (OwnerData), -1);
end;

```

Lo que cambia considerablemente es el código ejecutado en el evento *AfterGetRecords* del mismo componente:

```

procedure TmodDatos.prPedidosAfterGetRecords(Sender: TObject;
    var OwnerData: OleVariant);
begin
    if SQLQuery1.Active then
    begin
        OwnerData := SQLQuery1.FieldByName('OrderNo').AsInteger;
        SQLQuery1.Close;
    end
    else
        OwnerData := MaxLongInt;
    end;

```

Este código se basa en una hipótesis de trabajo, que hay que verificar. He realizado esa verificación primero mediante experimentos, pero al final mirando el código fuente. La hipótesis es verdadera en Delphi 4, Delphi 5 y Delphi 6. Cuando aparezca Delphi 7 tendremos que volver a verificarla, por si Borland la invalida. La hipótesis en sí es muy sencilla. Cuando un proveedor ha terminado de leer un paquete de registros, debe cumplirse obligatoriamente una de las siguientes condiciones:

- 1 Si el paquete era el último disponible, es decir, si no hay más registros, la consulta debe estar cerrada.
- 2 Si quedan más registros, el cursor se encuentra situado sobre el primer registro de la próxima hornada.

¿Reconoce ya lo que está haciendo el método anterior? Si encuentra que la consulta sigue activa, hay más registros, y en *OwnerData* se pasa la clave del primer registro del siguiente grupo. Con el método de Borland, *OwnerData* tenía la clave del último registro del grupo actual. Por eso es que hemos cambiado el tipo de comparación en la consulta. Lo que más me gusta de este sistema es que tanto el cálculo de *OwnerData* como su uso posterior en el parámetro de la consulta se efectúan ambos en la capa intermedia, mientras que antes el conjunto de datos cliente tenía un protagonismo peligroso. Supongamos que hay un equipo de trabajo implementando una aplicación en tres capas. Si utilizamos la variante de Borland para implementar las lecturas incrementales, los programadores de la capa intermedia deben coordinar muy bien sus esfuerzos con los de la capa final visual, algo que en la práctica no suele funcionar muy bien.

De todos modos, sigue siendo necesario que el conjunto de datos cliente colabore para conservar el valor de *OwnerData* entre llamadas. Pero esta vez se le exige mucho menos:

```

procedure TmodDatos.PedidosBeforeGetRecords(Sender: TObject;
    var OwnerData: OleVariant);
begin
    OwnerData := FValorRecibido;
end;

procedure TmodDatos.PedidosAfterGetRecords(Sender: TObject;
    var OwnerData: OleVariant);
begin
    FValorRecibido := OwnerData;
end;

```

La variable *FValorRecibido* se declara dentro de la clase del módulo de datos:

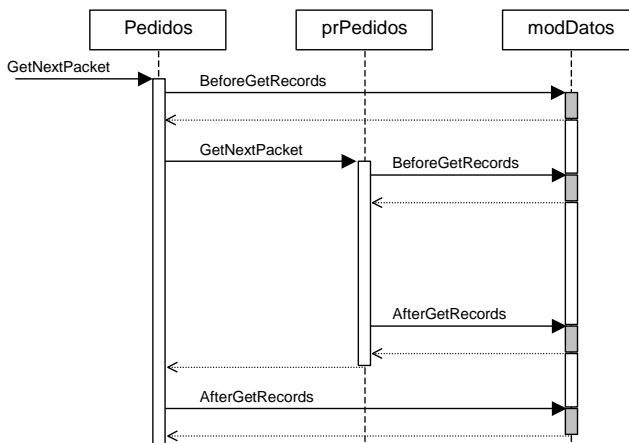
```

type
    TmodDatos = class(TDataModule)
        // ...
    private
        FValorRecibido: OleVariant;
    end;

```

Cuando el módulo de datos es creado, *FValorRecibido* se inicializa con *Unassigned*. La función *VarToStr* aplicada sobre esa constante devuelve también una cadena vacía, sin lanzar una excepción, que es lo que sucedería si intentásemos la conversión explícita. Luego *StrToIntDef* devolvería el valor por omisión indicado en su segundo parámetro, -1. Y la consulta se abriría a partir de los verdaderos primeros registros.

Para terminar, quiero mostrar el aspecto de un diagrama de secuencias de UML, con la representación de lo que sucede cuando un conjunto de datos cliente necesita el próximo grupo de registros. Para simplificar, he dejado al conjunto de datos cliente y a su proveedor residiendo en el mismo módulo de datos. *Pedidos* es el *TClientDataSet*, *prPedidos* es el *TDataSetProvider*, y *modDatos* es *modDatos*. No voy a extenderme en la explicación del diagrama: se supone precisamente que estos son intuitivos y más fáciles de comprender que el código fuente real. Tengo mis dudas, pero ahí está el diagrama, como constancia de mi esfuerzo:





## Peticiones explícitas

Hasta el momento, hemos dejado que sea el propio *TClientDataSet* el responsable de pedir sus registros al proveedor. Pero puede interesarnos controlar esta operación. En ese caso, el primer paso consistiría en borrar el valor de la propiedad *ProviderName* del conjunto de datos cliente. Que se olvide que existe algo llamado “proveedor”. Podemos entonces traer todos los registros del proveedor y asignarlos en el conjunto de datos cliente mediante una instrucción como la siguiente:

```
Pedidos.Data := prPedidos.Data;
```

En otras palabras, la propiedad *Data* del proveedor devuelve una matriz variante, con todas las filas a las que puede acceder el componente.

### ¿Y PARA QUE?

Puede que le parezca una tontería traer los datos de esta forma. Es que realmente es una tontería... pero veremos, en el siguiente capítulo, que algún *bug* en Delphi nos obligará a trabajar de esta manera. En concreto, cuando tengamos que traer registros de una relación maestro/detalles y hayan muchos registros maestros, veremos que es preferible traer por separados los registros maestros y de detalles, y establecer la relación en el cliente. Lamentablemente, al enlazar dos conjuntos de datos clientes encontraremos problemas si ambos apuntan a un proveedor.

No obstante, podemos ser más sutiles al pedir registros. Bajando un poco el nivel de abstracción, encontramos dos métodos interesantes en *TCustomProvider* para recuperar paquetes de datos:

```
function TCustomProvider.GetRecords(Count: Integer;
  out RecsOut: Integer; Options: Integer): OleVariant; overload;

function TCustomProvider.GetRecords(Count: Integer;
  out RecsOut: Integer; Options: Integer;
  const CommandText: WideString;
  var Params, OwnerData: OleVariant); OleVariant; overload;
```

El primero de ellos nos permite indicar la cantidad de registros que queremos traer en su parámetro *Count*. Si pasamos  $-1$ , significa que nuestra ambición no tiene límites, y queremos todos los registros. Si el valor de *Count* es positivo, representa el número de registros que queremos traer. Naturalmente, una cosa quiere el borracho y otra el camarero, y hay que mirar entonces el valor que se devuelve en el parámetro *RecsOut*, para verificar si el proveedor nos ha hecho caso. ¿Y si *Count* es cero? Sorpréndase: el proveedor nos devuelve los *metadatos* del conjunto de datos subyacente; quiero decir, su estructura de campos e índices. Esta información es la que se usa en tiempo de diseño para crear datos en un *TClientDataSet* sin necesidad de traer un solo registro al cliente.

¿Y qué pasa con los metadatos en el caso en que *Count* es positivo? Ah, para eso está el parámetro *Options*, de tipo *Integer*. En realidad, este parámetro debería pertenecer al tipo *TGetRecordOptions*:

```
type
  TGetRecordOption = (grMetaData, grReset, grXML, grXMLUTF8);
  TGetRecordOptions = set of TGetRecordOption;
```

Y nos bastaría con incluir *grMetaData* en el conjunto de opciones para que el vector devuelto por *GetRecords* incluyese siempre la información del esquema relacional. El problema es que este método debe servir para implementar una interfaz COM llamada *LAppServer*, y ya sabemos que el *marshaling* de interfaces remotas funciona sin que tengamos que intervenir solamente cuando limitamos el tipo de los parámetros a unos cuantos tipos bien definidos. Y los tipos de conjuntos de Delphi no entran en ese grupo. ¿Solución de Borland? Como un conjunto de menos de 32 opciones siempre cabe dentro de un entero, declaramos el parámetro como un entero; asunto solucionado. Se trata de una enorme chapuza por supuesto. Una cosa es el tipo de interfaz, en el que, efectivamente, estamos obligados a usar *Integer*. Y otra muy diferente es un método que Delphi siempre llamará internamente.

Pero no cambie de canal, que siguen los desatinos. Hasta Delphi 5, teníamos que pasar en *Options* expresiones como la siguiente:

```
Ord(grMetaData) or Ord(grReset)
```

Alguien se dio cuenta de que se podía recuperar parte de la elegancia perdida si nos echaban una mano y declaraban constantes para esos valores. Ya en Delphi 6 podemos sustituir la expresión anterior por ésta, más legible:

```
MetadataOption + ResetOption
```

¿Correcto? Espere y eche un vistazo a la declaración de las constantes:

```
const
  ResetOption: Integer = byte(grReset);
  MetadataOption: Integer = byte(grMetaData);
  XMLOption: Integer = byte(grXML);
  XMLUTF8Option: Integer = byte(grXMLUTF8);
```

Je, je, puede reírse conmigo, porque es un error de principiantes. En primer lugar, han declarado constantes *con tipo*, que ocupan memoria en el segmento de datos, y que obligan a cargarlas en un registro antes de poder pasarlas como parámetros. En segundo lugar, está esa asquerosa conversión directa con el tipo *Byte*. ¡Niklaus Wirth, cierra tus ojos y no veas este despropósito! ¡Por Belcebú, si para esto fue que el viejo Wirth introdujo la función *Ord* en Pascal!

No obstante, falta todavía lo mejor. Hasta aquí usted puede pensar que se trata de detalles de estilo sin mayor trascendencia. En definitiva, el coste en eficiencia no sobrepasa los 32 bytes entre código y datos, y un par de ciclos de reloj adicionales.

Pero el crimen no paga: lo más terrible de esta historia es que alguien, no sé si el que programó *midas.dll* o el que creó la unidad *Provider*, metió la pata hasta la ingle, y los valores de las constantes *grMetadata* y *grReset* ¡están intercambiados! Como dice el refrán: errar es de humanos, pero para cagarla de verdad hace falta un ordenador.

Para ilustrar el uso de *GetRecords*, vamos a utilizar la segunda variante del método, que como puede ver, nos permite pasar un texto de comando, parámetros o incluso datos a la medida (*OwnerData*). Observe la siguiente imagen:

N° pedido	N° cliente	Fecha venta	Fecha envío	N° empleado	Forma pago	Total
1.023	1.221	01/07/1988	02/07/1988	5	Check	\$4.674,00
1.024	3.151	02/07/1988	03/07/1988	2	Check	\$6.897,00
1.025	1.510	03/07/1988	04/07/1988	8	AmEx	\$930,00
1.026	1.624	07/07/1988	08/07/1988	44	AmEx	\$2.920,00
1.027	1.384	07/07/1988	08/07/1988	34	Visa	\$25.210,00
1.028	1.651	07/07/1988	08/07/1988	11	Visa	\$343,80
1.029	1.645	18/07/1988	19/07/1988	110	MC	\$20.108,00
1.030	3.615	25/07/1988	26/07/1988	107	MC	\$559,60
1.031	2.118	28/07/1988	01/08/1988	127	Credit	\$12.685,00

El ejemplo es una variante de la aplicación de carga incremental de registros. La versión original iba añadiendo los paquetes sucesivos de registros a los datos ya existentes. Cuando explico este mecanismo en mis cursos, siempre hay alguien que se preocupa por lo que puede pasar si el usuario deja pulsada la tecla de avance: como comprenderá, si el conjunto de datos origen es suficientemente grande, puede que nos quedemos sin memoria en el cliente. No hay que perder el sueño, porque se necesitan *muuuuuuchos* registros para llegar a ese extremo, pero es una preocupación lógica. ¿Qué tal entonces si hacemos que cada grupo de registros desplace al grupo anterior, en vez de concatenarse al final?

Puede que haya otras soluciones, pero ésta es bastante sencilla: he seleccionado el componente *Pedidos*, el conjunto de datos cliente, y he roto su vínculo con el proveedor. También he eliminado la respuesta a los eventos *Before* y *AfterGetRecords* del conjunto de datos, que servía para propagar el valor de *OwnerData* entregado por el proveedor. Luego he añadido el siguiente método público en la clase del módulo:

```

procedure TmodDatos.PedirRegistros(
    PrimerGrupo: Boolean; Cantidad: Integer);
var
    Params, OwnerData: OleVariant;
    RecsOut: Integer;
begin
    if PrimerGrupo then FValorRecibido := 0;
    OwnerData := FValorRecibido;
    Pedidos.Data := prPedidos.GetRecords(
        Cantidad, RecsOut, MetadataOption + ResetOption,
        '', Params, OwnerData);
    FValorRecibido := OwnerData;
end;

```

Creo que es fácil de explicar: *PedirRegistros* pide cierta cantidad de registros al proveedor, como su nombre indica. Eso sí, pasa en la llamada el último valor de la clave recibido, al igual que en el ejemplo original, y al retomar el control almacena el valor devuelto por el proveedor en ese mismo parámetro. Además, si su parámetro *PrimerGrupo* vale *True*, fuerza la recuperación del primer paquete de registros. Están, además, esas dos constantes del tercer parámetro guiñándonos los ojos: ¡pruebe lo que pasa si quita *ResetOption*!

Por último, he añadido dos acciones en la ventana principal: una para recuperar el primer grupo de registros y la otra, para recuperar el grupo que le sigue:

```
procedure TwndMain.PedirRegistros(Sender: TObject);  
begin  
    modDatos.PedirRegistros(Sender = acPrimerGrupo, 20);  
end;
```

En el capítulo 34 veremos cómo implementar este mecanismo cuando el proveedor se encuentra fuera de nuestro alcance, en un módulo de datos remoto.

## Proveedores (II)

**L**AS TÉCNICAS ESTUDIADAS EN EL CAPÍTULO anterior son solamente la base de muchas otras posibilidades del trabajo con proveedores. En este capítulo, en concreto, veremos cómo Delphi mezcla los conjuntos de datos clientes con otras interfaces de acceso SQL, para ofrecer componentes que permitan crear prototipos de aplicaciones. Pero lo más interesante será ver cómo se comportan los proveedores cuando se asocian a conjuntos de datos SQL configurados como maestro y detalles. Para terminar, estudiaremos cómo se implementa la actualización (*refresh*) de registros en este tipo de aplicaciones<sup>28</sup>.

### Simbiosis

Entre las pocas cosas que recuerdo de mis primeros años de escuela, hay una clase de Botánica. La profesora fue paseando tres frascos por el aula: en uno tenía un alga repugnante y en el segundo, un hongo asqueroso. Antes de enseñarnos el tercero, nos largo un aburrido discurso sobre simbiosis, parasitismo y no recuerdo qué otro tipo de relación vergonzosa entre bichos. Mi mente comenzó a dibujar un monstruo como los de los cómics: baboso, con colmillos afilados y zonas putrefactas de color verde hongo. Cuando al fin nos mostró el frasco con el liquen, sufrí una decepción y a la vez una iluminación: aquella cosa compacta y lechosa no atemorizaba ni a las chicas, pero indicaba que una simbiosis era algo menos superficial que poner algas y hongos en una batidora y darle al interruptor. Presentí que el concepto “simbiosis” jugaría un importante papel durante mi vida adulta y...

Y como hasta el momento no me había vuelto a acordar de la simbiosis, excepto al ver los dibujos animados de Spiderman, ¡qué demonios!, pensé que sería un buen título para bautizar una sección del libro. Imaginad: la cara oculta de Delphi poblada por espantosas simbiosis de algas, hongos y programadores de Java.

Si capturamos un *TCustomClientDataSet*, lo encerramos con un *TDataSetProvider*, apagamos la luz y ponemos música sugerente, al cabo de un rato veremos una nueva generación de pequeñines *TCustomCachedDataSet* retozando por todo el ordenador.

---

<sup>28</sup> *Refresh* es verbo; si tiene un Word 97 con el diccionario de sinónimos inglés, teclee *refreshment*, márquela como palabra inglesa y busque sus sinónimos. Más leña a la leyenda...

No los busque, sin embargo, en la Paleta de Componentes, porque la saga familiar aún no ha concluido. Como en Delphi no hay herencia múltiple, la paternidad hay que atribuírsela al *TCustomClientDataSet*; el proveedor es utilizado internamente para delegar ciertas operaciones del conjunto de datos resultante.

Hay que añadir un tercer ingrediente para lograr un componente realmente utilizable. Las posibilidades son:

- *TCustomCachedDataSet* + *TSQLDataSet*  $\Rightarrow$  *TSQLClientDataSet*
- *TCustomCachedDataSet* + *TBDEQuery*  $\Rightarrow$  *TBDEClientDataSet*
- *TCustomCachedDataSet* + *TIBQuery*  $\Rightarrow$  *TIBClientDataSet*

Al igual que antes, el componente final descende solamente de la clase *TCustomCachedDataSet*, y el segundo componente se mezcla por delegación.

Como puede imaginar, el objetivo de estos simbioses es ahorrarnos pasos intermedios cuando queremos trabajar con la combinación de sus componentes básicos. Para InterBase Express y el BDE, añadir un conjunto de datos cliente como caché no es un problema acuciante; en cualquier caso, lo utilizaríamos para disimular *bugs* en sus respectivos subsistemas de caché y actualización. Pero si queremos programar aplicaciones interactivas con DB Express, estamos casi obligados a recurrir a los conjuntos de datos cliente. Podríamos configurar por separado los componentes de DB Express, el proveedor y el *TClientDataSet*, pero si lo que estamos programando es una prueba o un prototipo, es mucho más rápido tirar de un *TSQLClientDataSet*, a pesar de su interminable nombre.

No obstante:

*“Los conjuntos de datos simbioses deberían utilizarse solamente para pruebas y prototipos”*

Es duro reconocerlo, pero la advertencia tiene su justificación:

- 1 Hay montones de errores en la implementación actual de esos componentes. Estoy escribiendo esto antes del Update Pack 1 de Delphi 6.
- 2 Aunque nuestro objetivo inicial sea crear solamente una aplicación en dos capas físicas, es una tontería perder la posibilidad de dividirla posteriormente en tres.
- 3 Los simbioses no son adecuados para trabajar con relaciones maestro/detalles. Este punto lo demostraremos más adelante.

Como nuestras expectativas de ahorro de trabajo se han visto rebajadas considerablemente, vamos a estudiar solamente una de estas clases mixtas, la que resulta de la mezcla con DB Express. El comportamiento de las otras dos es similar.

## Datos clientes en DB Express

Para mostrar las propiedades más importantes de un *TSQLClientDataSet*, crearemos una nueva aplicación de ejemplo. Comenzaremos añadiendo al formulario principal un *SQLConnection1*, y configurándolo para que apunte a *mastsqlgdb*, la base de datos de InterBase que hemos venido utilizando hasta el momento. Y actívelo. No necesitaremos crear un módulo de datos por separado, porque la aplicación será muy sencilla.

A continuación, en la misma página DB Express de la Paleta de Componentes, seleccione un *TSQLClientDataSet*, el último componente de la fila. En vez de dejarlo caer sobre el formulario, déjelo caer sobre el nodo del componente de conexión, en el *Object TreeView* que está encima del Inspector de Objetos. Compruebe entonces que la propiedad *DBConnection* del componente apunta automáticamente a la conexión. Para que esto suceda, le advierto, debe estar activa la conexión.

*DBConnection* es una propiedad común, con distintos tipos de datos, a todos los conjuntos de datos mixtos. No obstante, *TSQLClientDataSet* tiene una propiedad adicional muy particular, llamada *ConnectionName*, una cadena de caracteres. Si la base de datos a la que vamos a conectarnos coincide con una de las conexiones registradas en el fichero *dbxconnections.ini*, podemos utilizar el nombre registrado para la conexión y evitar así tener un componente *TSQLConnection* independiente. Por supuesto, con esta técnica no podríamos controlar la introducción de la contraseña. Y si tenemos más de un conjunto de datos en la aplicación, estos no podrán compartir la misma conexión, como sucedería con un componente de conexión independiente.

Viene ahora la parte importante: hay que especificar de dónde extraerá su contenido el conjunto de datos. Para este propósito tenemos las propiedades *CommandType* y *CommandText*, al igual que lo que sucede con el *TSQLDataSet*. Para este experimento, asigne *ctQuery*, el valor por omisión, en la propiedad *CommandType*, y luego, en *CommandText*, la siguiente sentencia:

```
select *
from   CUSTOMER
where  Company starting with :prefijo
order by Company asc
```

He querido añadir un comando en la instrucción SQL porque esa será la situación más frecuente en una aplicación real, y así comprobaremos, además, qué tal se comporta el componente Spiderman cuando hay parámetros danzando cerca. Si abrimos el editor de la propiedad *Params* veremos que el conjunto de datos reconoce el parámetro *prefijo* de forma inmediata. Recuerde que si estuviéramos utilizando un proveedor, un *TClientDataSet* y un *TSQLDataSet* por separado, tendríamos que haber ejecutado el comando *Fetch Params* del menú local del conjunto de datos cliente. Lo que no se hace automáticamente es asignarle un tipo al parámetro. *Prefijo* debe ser de tipo

*ftString*, y es conveniente que tecleemos un prefijo inicial en la propiedad *Value* del parámetro. Una letra *A*, por ejemplo; no es necesario poner comillas.

Una vez que haya configurado *SQLClientDataSet1*, añada los campos correspondientes en el *Fields Editor*, y modifique sus propiedades a su antojo. Por variar un poco, para el ejemplo del CD he creado una interfaz como la siguiente, para examinar los registros uno a uno:

Tenemos que ocuparnos de la asignación de valores al parámetro en tiempo de ejecución. Vamos a hacerlo añadiendo una lista de acciones, *ActionList1*, al formulario; pronto comprenderá por qué. No se moleste en traer un *ImageList*, porque esta vez vamos a utilizar componentes *TSpeedButton* para los botones, y estos tienen un sistema propio y arcaico de asignaciones de imágenes: la propiedad *Glyph*. Cree primero una acción con el nombre *acBuscar*, y deje en blanco la propiedad *Caption*, para no tener problemas con el botón que le asociaremos. La respuesta a su evento *OnExecute* debe ser:

```
procedure TwndPrincipal.acBuscarExecute(Sender: TObject);
begin
    SQLClientDataSet1.Close;
    SQLClientDataSet1.Params[0].AsString := edBuscar.Text;
    SQLClientDataSet1.Open;
end;
```

Como debe imaginar, *edBuscar* será un cuadro de edición que añadiremos junto con un *TSpeedButton*. Recuerde asociar la acción al botón modificando la propiedad *Action* de este último.

Con el paso anterior, tendríamos suficiente. Pero vamos a adelantarnos un poco para ver cuán fácil o difícil sería programar la grabación de los cambios que pueda hacer el usuario. Antes, ejecute la aplicación, y modifique cualquier registro con cualquier tontería. Comprobará que, aunque pulse el botón del navegador que ejecuta el mé-



todo *Post*, los cambios se quedan en la parte del conjunto de datos cliente, pero no se aplican realmente a la base de datos SQL.

Añadamos entonces una segunda acción a la lista de acciones, *acGuardar*, y asóciala a otro *TSpeedButton*, al igual que antes. Manejaremos el evento *OnUpdate* de la acción para especificar cuándo se dan las condiciones para su ejecución:

```
procedure TwndPrincipal.acGuardarUpdate(Sender: TObject);
begin
    TAction(Sender).Enabled := SQLClientDataSet1.Modified or
        (SQLClientDataSet1.ChangeCount > 0);
end;
```

Como ve, el código es similar al utilizado para las actualizaciones en MyBase. Primero verificamos si la fila activa tiene modificaciones en ese momento, leyendo la propiedad *Modified*. Y si no, averiguamos el número de actualizaciones en la caché mediante nuestra vieja conocida, la propiedad *ChangeCount*.

Donde sí habrá novedades, respecto a MyBase, será en la respuesta a *OnExecute*:

```
procedure TwndPrincipal.acGuardarExecute(Sender: TObject);
begin
    if SQLClientDataSet1.ApplyUpdates(0) > 0 then Abort;
end;
```

Mientras MyBase guarda los cambios con *SaveToFile* y, opcionalmente, *MergeChangeLog*, ahora llamamos a un método llamado *ApplyUpdates*, cuyo comportamiento explicaremos en el próximo capítulo.

Sólo nos queda un detalle: por las peculiaridades de este sistema de grabación, si se produjese un error durante la misma, no veríamos el mensaje en pantalla a no ser que intercepiásemos el evento *OnReconcileError* de *SQLConnection1*:

```
procedure TwndPrincipal.SQLClientDataSet1ReconcileError(
    DataSet: TCustomClientDataSet; E: EReconcileError;
    UpdateKind: TUpdateKind; var Action: TReconcileAction);
begin
    Application.ShowException(E);
end;
```

Sé que todo esto puede parecer chocante, pero tenga un poco de paciencia, que aclararemos el misterio en el momento oportuno.

## Relaciones maestro/detalles

Prepárese, que llega la parte más interesante del trabajo con proveedores: la configuración en relaciones maestro/detalles. Inicie una aplicación, y añádale un módulo de datos vacío. Comencemos por añadir y configurar los componentes SQL. Primero un *SQLConnection1*, asociado como hasta ahora a la base de datos *mastsql.gdb* de In-

terBase. Después, la consulta maestra, un componente de tipo *TSQLQuery* que llamaremos *qrClientes* y al que asociaremos la siguiente instrucción:

```
select *
from   CUSTOMER
```

Ya sé que lo que vamos a hacer es una barbaridad, porque obligaremos al proveedor a que nos traiga una considerable cantidad de registros, pero no quiero complicar este primer ejemplo.

A continuación, traiga un *TDataSource*, llámelo *dsClientes* y haga que su propiedad *DataSet* apunte a la consulta de clientes. Nos queda entonces la consulta de detalles. Será otro *TSQLQuery*, al que llamaremos *qrPedidos*. Su propiedad *DataSource* debe apuntar a *dsClientes*, y su instrucción SQL debe ser:

```
select *
from   ORDERS
where  CustNo = :CustNo
order by OrderNo asc
```

Resumiendo, tenemos una consulta que devuelve todos los registros de clientes, y le hemos asociado una segunda consulta de detalles, que para cada cliente muestra todos los pedidos que ha realizado. Le sugiero que, aunque va a ser bastante trabajo, configure al menos las propiedades *DisplayLabel* de los campos de ambas consultas.

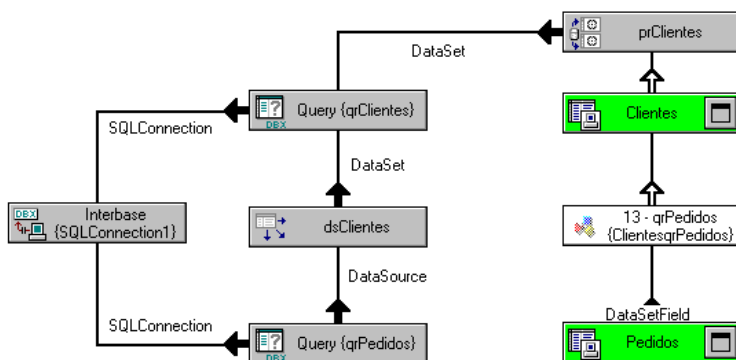
Es el momento de traer un *TDataSetProvider* desde la página *Data Access*, al que renombraremos como *prClientes*. Mucha atención: ¡necesitaremos solamente un proveedor! Curiosamente, sólo tendremos que añadir la opción *poIncFieldProps* en su propiedad *Options*.

Traiga ahora un *TClientDataSet* y cambie su nombre a *Clientes*; como siempre, reservo el mejor nombre para el componente que más vamos a utilizar. Enlázelo al proveedor a través de su propiedad *ProviderName*. Haga doble clic sobre *Clientes* para mostrar el Editor de Campos, y añada todos los objetos de acceso a campos disponibles. Verá todas las columnas que ya sabemos que tiene la tabla de clientes del ejemplo... y una pequeña sorpresa al final de la lista, ¡un campo llamado *qrPedidos*, del tipo *TDataSetField*! En la lista de campos aparece bajo ese nombre; la variable que accede a él se llama *ClientesqrPedidos*, que surge de concatenar el nombre del conjunto de datos con el del campo “virtual” *qrPedidos*, por llamarlo de algún modo.

La sorpresa no es tanta, porque ya conocíamos la existencia de los campos de tipo *TDataSetField*, especialmente para representar conjuntos de datos anidados en MyBase. La novedad está en que el proveedor “invente” uno de esos campos para transmitir al cliente la consulta de detalles asociada al conjunto de datos principal. Puede imaginar cuál será el segundo paso: traiga otro *TClientDataSet* al módulo de datos, ¡y no toque la propiedad *ProviderName*! Si estuviésemos trabajando en tres capas, tampoco deberíamos tocar *RemoteServer*. El motivo es que el nuevo conjunto de datos, al que llamaremos *Pedidos* a secas, no “conversa” directamente con el provee-

dor, sino que recibirá sus datos y su esquema por mediación del campo *CientesqrPedidos*.

Para ayudarle a comprender la nueva configuración, he creado este esquema en la vista *Diagram* del editor de código de Delphi, arrastrando sobre la misma todos los componentes que actúan en este drama:



### ADVERTENCIA

Al ejecutar la aplicación, verá que el formulario principal tarda un poco en aparecer. Se debe, como he dicho al principio de la sección, a que estamos cometiendo una barbaridad al traer todos los registros de clientes y los de detalles de un solo golpe. Tengo todavía que explicarle cuál es el sentido de esta forma de trabajo.

## La verdad sobre perros y gatos

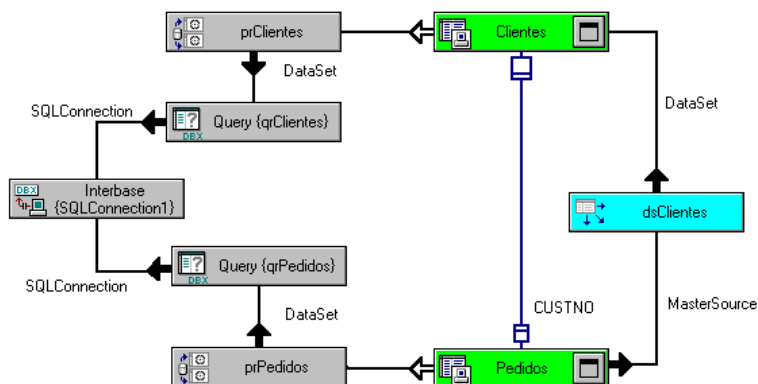
Aunque el experimento de la sección anterior es bastante espectacular, no debemos dejarnos engañar sobre su verdadera utilidad. En realidad, si quisiéramos recuperar un conjunto medianamente grande de clientes y *todos* los pedidos asociados a ellos, la técnica descrita sería la peor, desde el punto de vista de la eficiencia. El problema está en cómo se buscan los pedidos en la capa SQL. Por cada cliente que se encuentra, hay que ejecutar la consulta de detalles para después cerrarla. Si hay 50 clientes, *qrPedidos* se abre, evalúa y cierra 50 veces antes de que el proveedor esté en condiciones de enviar los datos al conjunto de datos cliente. Si no me cree, puede añadir un monitor de DB Express a la prueba y analizar la traza.

Si la técnica es ineficiente, ¿por qué Borland se ha molestado en programar todo esto? Hay dos razones:

- 1 La técnica es ineficiente si hay un número elevado de registros maestros, digamos que 100 o 200. La cifra, por supuesto, depende de las características de las consultas involucradas. Pero si hay pocos registros en la consulta principal, la diferencia no se nota tanto.

- 2 La verdadera utilidad de los conjuntos de datos anidados consiste en que, de esta forma, la caché de registros en el lado cliente tiene una estructura integrada, si la comparamos con la caché que se crearía con dos *TClientDataSet* independientes. Y en que, gracias a esta estructura, el algoritmo de actualización evita ciertos problemas muy graves que son casi inevitables cuando se utilizan cachés independientes.

Cuando hablo, en el segundo punto, sobre cachés integradas y cachés independientes, es porque estoy pensando en una configuración alternativa de conjuntos de datos clientes y proveedores. Analice el siguiente diagrama:



En el diagrama se muestran dos consultas independientes, en la capa SQL: *qrClientes*, para devolver todos los clientes, y *qrPedidos*, que esta vez devuelve todos los pedidos, sin importar a qué clientes pertenecen. Observe también que hay dos proveedores separados, uno para clientes y otro para pedidos. Consecuentemente, también existen dos conjuntos de datos clientes, uno por proveedor. Posteriormente, se establece la relación maestro/detalles, pero esta vez en el lado cliente. Se añade un *dsClientes*, una fuente de datos, para asociarla a *Clientes*, el *TClientDataSet*. En *Pedidos* se modifican las propiedades *MasterSource*, *MasterFields* e *IndexFieldNames*, exactamente igual que si se tratase de relacionar dos conjuntos de datos en DB Express. La idea general es que primero se leen todos los clientes, luego se leen todos los pedidos, y posteriormente se establece la relación con los datos que ya se encuentran en caché.

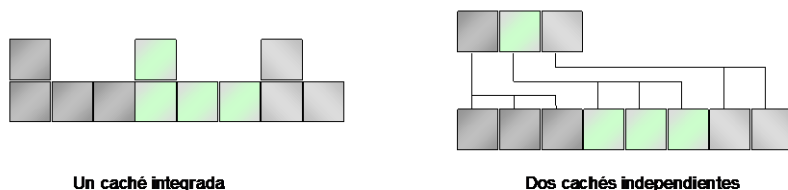
### ADVERTENCIA

Esta configuración produce varios errores en Delphi 6 cuando los conjuntos de datos SQL pertenecen a DB Express o al BDE. En realidad, pude hacerla funcionar sólo con los componentes de ADO Express. Cuando estudiemos los componentes de conexión de DataSnap, explicaré cómo salvar la situación, si estuviéramos obligados a configurar la relación maestro/detalles en el lado cliente.

Por una parte, ahora leeríamos todos los clientes de golpe, al igual que antes. Respecto a los pedidos, antes teníamos también que leerlos todos, pero para ello el proveedor abría y cerraba la consulta *qrPedidos* una vez por cada cliente. Con el nuevo

sistema, también traemos todos los pedidos, pero *qrPedidos* se abre y cierra una sola vez.

La diferencia estructural se muestra a continuación. Cuando estudiemos el algoritmo de resolución de DataSnap, veremos que es más sencillo actualizar la caché integrada de la izquierda.



¿Y qué sucede cuando se configuran dos componentes simbióticos, como *TSQL-ClientDataSet* en una relación maestro/detalles? En ese caso, los detalles se evalúan cada vez que se mueve la fila del conjunto de datos maestro, de forma parecida a como se comportarían dos conjuntos de datos DB Express configurados del mismo modo. Sin embargo, los registros de detalles se conservan en la segunda caché una vez leídos. Si regresáramos al primer registro maestro, no tendríamos necesidad de volver a recuperar sus filas de detalles.

## La teoría de la lectura por demanda

Lo siento mucho, pero tengo que seguir mostrando errores y problemas. Hay otro *bug* relacionado con una de las opciones de los proveedores: la opción *poFetchDetailsOnDemand*, de la propiedad *Options*. Hemos visto que, por omisión, los conjuntos de datos clientes asociados a configuraciones maestro/detalles siguen trayendo de golpe todos los registros maestros y de detalle. En el capítulo anterior mencioné la existencia de una propiedad llamada *FetchOnDemand* en los conjuntos de datos clientes, y que si le asignábamos *False* como valor, teníamos que pedir explícitamente los registros para el *TClientDataSet*.

*poFetchDetailsOnDemand* no es tan rígida. En primer lugar, se especifica en el proveedor, no en el conjunto de datos cliente. En segundo, es independiente del valor que tenga *FetchOnDemand* en el *TClientDataSet*. Por último, afecta solamente a la recuperación automática de las filas de detalles. Teóricamente, si usamos esta opción podríamos leer todos los registros de clientes sin necesidad de traer también de golpe todas las filas de pedidos. Solamente cuando seleccionáramos un cliente por primera vez, el conjunto de datos cliente pediría los detalles que no hubiese recuperado ya.

Desgraciadamente, la lectura de detalles según demanda hace aguas por todas partes. Si quiere probarlo, abra nuevamente el proyecto que nos sirvió de ejemplo para mostrar el uso de proveedores con relaciones maestro/detalles, y añada la opción *poFetchDetailsOnDemand* a la propiedad *Options* del proveedor. Descubrirá que, a partir

de ese momento, ya no podrá abrir *Clientes*, el conjunto de datos maestro, porque obtendrá el siguiente mensaje de error:

*"Cannot perform this operation in a closed dataset"*

Es extraño, sin embargo, que sí podríamos abrir la tabla de detalles, *Pedidos*; aparentemente, también se abriría la tabla maestra. Pero este hecho no tiene utilidad alguna para nosotros. Si espiásemos las instrucciones generadas por el proveedor, veríamos que, a pesar de la nueva opción activa, el proveedor sigue enviando al cliente todos los registros de detalles. Mala suerte que tenemos.

Pero no todo son problemas. Hay otra opción en el proveedor, *poFetchBlobsOnDemand*, que está íntimamente relacionada con la de los detalles, ¡y para variar, funciona! Suponga que tenemos una tabla con fotos; como es usual, cada foto ocupa bastante espacio. Accedemos a esa tabla a través de un proveedor. Si utilizamos las opciones por omisión el proveedor se comportará glotonamente e intentará enviar al conjunto de datos todos los registros con sus correspondientes fotos. Es decir, demasiada información.

En cambio, si activamos *poFetchBlobsOnDemand* en el proveedor, solamente se envían al cliente aquellos campos de la tabla que no sean de tipo blob. Cuando el conjunto de datos cliente cambia su fila activa, se realiza peticiones puntuales al servidor para traer la imagen correspondiente. Si nunca pasamos por determinado registro, nunca tendremos que recuperar su imagen.

Y aunque no podamos utilizarlos ahora, sepa que existen dos métodos relacionados con las opciones que acabo de explicar:

```
procedure TClientDataSet.FetchDetails;  
procedure TClientDataSet.FetchBlobs;
```

Si la propiedad *FetchOnDemand* del conjunto de datos cliente vale *False*, y el proveedor no incluye detalles o campos blobs de manera automática, podemos ejecutar explícitamente estos métodos para recuperar la información que nos falta.

## El problema de las consultas de detalles

¿Qué tal si le cuento la historia de un *bug* que ha desaparecido en Delphi 6? El error afectaba el funcionamiento de las consultas enlazadas como maestro y detalles.

Como es posible que este libro sea comprado por programadores que todavía estén con Delphi 5, voy a explicar en qué consiste y cuál es la solución que se le daba en aquella versión.

Volvamos un momento a la primera configuración maestro/detalles: un par de consultas SQL enlazadas entre sí, un solo proveedor, y dos conjuntos de datos clientes; el segundo de ellos conectado a través de su propiedad *DataSetField* a un campo de

tipo *dataset* del primero. Cuando el conjunto de datos cliente maestro se activa, el proveedor lee todos los registros, tanto los maestros como los de detalles, y los envía de una vez al *TClientDataSet*. Y, aunque este último componente se mantenga activo, el proveedor cierra las dos consultas SQL al terminar la lectura de sus registros.

Sin embargo, en Delphi 5 se quedaba abierta la consulta de detalles, con lo que perdíamos uno de los atractivos de la programación con proveedores. Por suerte, era fácil corregir el error. Supongamos que, al igual que antes, la consulta maestra se llamaba *qrClientes*, y la de detalles *qrPedidos*. Podíamos interceptar entonces dos eventos de *qrClientes*: *AfterOpen* y *AfterClose*:

```
procedure TmodDatos.qrClientesAfterOpen(DataSet: TDataSet);
begin
    qrPedidos.Open;
end;

procedure TmodDatos.qrClientesAfterClose(DataSet: TDataSet);
begin
    qrPedidos.Close;
end;
```

Tengo unas cuantas aplicaciones funcionando en las más diversas condiciones, por lo que puedo garantizar el correcto desempeño de estos pequeños “parches”.

Ahora bien, en una aplicación real las cosas no son tan fáciles. Es típico tener módulos con decenas de conjuntos de datos enlazados entre sí en las más extrañas combinaciones. Si tuviéramos que crear y mantener manejadores de eventos como los que acabo de mostrar para todos los grupos de consultas relacionadas, podríamos volvernos locos. Lo que hago entonces es introducir código para que, en tiempo de ejecución, las consultas maestras se asocien automáticamente a métodos como los anteriores.

El siguiente ejemplo pertenece a un servidor de capa intermedia escrito en Delphi 5 y basado en ADO Express; es muy fácil adaptar el código para otras interfaces de acceso. Primero redefino el método protegido *Loaded* del módulo de datos donde sitúo todas las consultas, para asignar un manejador de eventos genéricos a ciertos conjuntos de datos:

```
// ;Sólo es necesario en Delphi 5!

procedure TWebData.Loaded;
var
    I: Integer;
    C: TComponent;
begin
    inherited Loaded;
    for I := 0 to ComponentCount - 1 do
        begin
            C := Components[I];
            if (C is TDataSource) and (TDataSource(C).DataSet <> nil) then
                begin
                    TDataSource(C).DataSet.AfterOpen := SharedAfterOpenClose;
```

```

        TDataSource(C).DataSet.AfterClose := SharedAfterOpenClose;
    end;
end;
end;

```

Recorro todos los componentes situados en el módulo, pero sólo me detengo en los de tipo *TDataSource*. Examinó entonces la propiedad *DataSet* del componente, y si hay algo asociado, modifico los punteros a eventos de ese componente. Eso significa que solamente modifico aquellos conjuntos de datos que son maestros en alguna relación maestro/detalles.

### ADVERTENCIA

Tenga cuidado, porque yo no acostumbro a poner componentes *TDataSource* en un módulo de datos, a no ser que intervenga en una relación maestro/detalles.

El nombre *SharedAfterOpenClose* pertenece a un método que declaro en la zona privada o protegida de la declaración de clase del módulo:

```

// ;Sólo es necesario en Delphi 5!

procedure TWebData.SharedAfterOpenClose(DataSet: TDataSet);
var
    I: Integer;
    Q: TADOQuery;
begin
    for I := 0 to Database.DataSetCount - 1 do
    begin
        Q := TADOQuery(Database.DataSets[I]);
        if Assigned(Q.DataSource) and
            (Q.DataSource.DataSet = DataSet) then
            Q.Active := DataSet.Active;
        end;
    end;
end;

```

Note que, en vez de recorrer todos los componentes del módulo, me limito a los conjuntos de datos, para lo cual aprovecho la propiedad *DataSets* de la conexión; en este ejemplo, el componente de conexión se llama *Database*. Mucho cuidado con esto, porque la propiedad *DataSets* del componente *TDatabase* del BDE solamente almacena las referencias de los conjuntos de datos que se encuentren activos en ese momento. Para cada conjunto de datos, compruebo si se trata o no de una consulta ADO, y en caso afirmativo, verifico si apunta indirectamente, a través de su propiedad *DataSource* al conjunto de datos que se está abriendo o cerrando, y que se pasa como parámetro al método.

Insisto: Delphi 6 corrigió este error, y la solución que acabo de explicar ya es innecesaria.

## Los valores más recientes

A pesar de la tan cacareada riqueza del idioma castellano, me encuentro frecuentemente con conceptos que en inglés se representan con dos palabras diferentes, y con



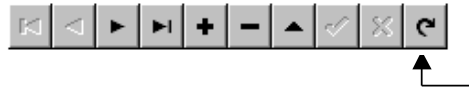
un solo vocablo en español. Creo que el problema tiene que ver con la obstinación con la que ciertos “ilustrados” combaten el uso de neologismos, aunque se trate de temas técnicos.

Por ejemplo, vamos a estudiar ahora una operación sobre conjuntos de datos que en inglés se denomina *refresh*. Si por mí fuera, diría *refrescar* ... y santas pascuas. Sin embargo, hay una traducción semioficial, que se utiliza en la versión en castellano del Internet Explorer y del Explorador de Windows: *actualizar*. Pero si le digo entonces que vamos a ver cómo se actualiza un conjunto de datos, usted pensará inmediatamente en *update*. Con *refresh* tenemos suerte, porque puedo sustituirlo por *releer*. No es el matiz exacto, pero nos la apañaremos.

Hay al menos dos operaciones de relectura de datos aplicables a un conjunto de datos clientes conectado a un proveedor. La más brutal es la siguiente:

```
procedure TClientDataSet.Refresh;
```

Es la misma operación que se ejecuta cuando pulsamos el botón del extremo derecho en una barra de navegación:



Una aclaración importante: *Refresh* vuelve a leer todos los registros que corresponden al conjunto de datos original. Menciono este punto porque sé que hay quien piensa que sólo se releen los registros “visibles”, es decir, aquellos que se están mostrando en algún control de datos en el momento en que se llama a *Refresh*.

Hasta cierto punto, *Refresh* es equivalente a cerrar y abrir el conjunto de datos cliente, con el añadido de que se intenta mantener la posición del registro activo. Aunque no siempre sea posible mantener esa posición, pues puede que el registro activo haya sido eliminado de la base de datos. Una consecuencia de esta interpretación es no se puede llamar a *Refresh* cuando existen cambios pendientes en el conjunto de datos cliente. Si lo hiciéramos, se produciría una excepción. Desafortunadamente, el botón de relectura de *TDBNavigator* no realiza esta comprobación. Si quisiéramos crear nuestra propia acción para releer todos los registros, deberíamos utilizar eventos similares a los siguientes:

```
procedure TForm1.acReleerTodoUpdate(Sender: TObject);  
begin  
    TAction(Sender).Enabled := ClientDataSet1.ChangeCount = 0;  
end;  
  
procedure TForm1.acReleerTodoExecute(Sender: TObject);  
begin  
    ClientDataSet1.Refresh;  
end;
```

## Relectura de registros

Siendo *Refresh* una operación tan poco refinada, hay más probabilidades de que utilizemos este otro método:

```
procedure TClientDataSet.RefreshRecord;
```

Como su nombre indica, solamente se relee el registro activo del conjunto de datos cliente. A diferencia de *Refresh*, este método no se preocupa por saber si hay cambios pendientes o no en el registro activo; de haberlos, los perderíamos irremisiblemente. Es buena idea, por lo tanto, invocar esta operación a través de una acción con manejadores de eventos similares a los siguientes:

```
procedure TForm1.acReleerUpdate(Sender: TObject);
begin
    TAction(Sender).Enabled :=
        not ClientDataSet1.Modified and
        (ClientDataSet1.UpdateStatus = usUnmodified);
end;

procedure TForm1.acReleerExecute(Sender: TObject);
begin
    ClientDataSet1.RefreshRecord;
end;
```

Alternativamente, podríamos pedirle una confirmación al usuario.

Además de ser más preciso, *RefreshRecord* tiene una implementación más interesante. Piense un momento, ¿cómo sabe el conjunto de datos cliente, o el proveedor, que tiene que volver a leer tal o más cual registro? Con *Refresh* era muy sencillo, porque se leían todos, a lo bestia. Ahora, con *RefreshRecord*, es necesario saber cuáles campos de la consulta pueden actuar como clave primaria o única.

¿Puede el componente proveedor averiguar por sí mismo la estructura de la clave primaria de un conjunto de datos? Al parecer, en ciertas situaciones sí. Por ejemplo, si el conjunto de datos SQL asociado al proveedor es una tabla de DB Express, el proveedor puede extraer esa información de los metadatos de la tabla; recuerde que un *TSQLTable* recupera información sobre los índices al activarse. Pero es preferible no confiarse y echarle siempre una mano al proveedor, modificando el valor de la propiedad *ProviderFlags* de los campos del conjunto SQL que pertenecen a la clave, o a lo que pretendemos utilizar como clave.

*ProviderFlags* es una propiedad definida en la clase *TField* como un conjunto basado en las siguiente opciones:

Opción	Significado
<i>pfInUpdate</i>	Indica si el campo es actualizable o no
<i>pfInWhere</i>	Indica si las condiciones de búsqueda deben incluir a este campo
<i>pfInKey</i>	¡Esta es la opción que necesitamos!

Opción	Significado
<i>pfHidden</i>	Si está activa, oculta el campo a la vista del <i>TClientDataSet</i>

El principal uso de estas opciones lo veremos al tratar la grabación por medio de proveedores, por lo que no insistiré en el papel de tres de ellas. Ahora solamente nos interesa *pfInKey*: debemos activarla para cada campo que forme parte de la clave primaria.

La información sobre los campos de la clave sirve para que el proveedor genere una consulta de manera automática, que lea los datos de *un solo registro*, y la ejecute, con la ayuda del conjunto de datos SQL. Quiero que tome plena conciencia de esta “extraña” situación, porque es muy importante para comprender cómo funciona DataSnap. Supongamos que tenemos un *TSQLQuery* con la siguiente consulta:

```
select *
from   CUSTOMER
where  Company starting with :prefijo
```

Se trata de nuestra vieja conocida, la base de datos de InterBase *mastsql.gdb*. La columna *CustNo* tiene valores diferentes en todos los registros; en realidad, es la clave primaria de la tabla, pero quise decirlo en una forma menos brutal. Al crear los componentes de acceso a campo para esa consulta, añadimos *pfInKey* a la propiedad *ProviderFlags* del campo correspondiente a *CustNo*.

Suponga entonces que asociamos a la consulta un proveedor y un conjunto de datos cliente, y que ejecutamos el método *RefreshRecord* de este último. Lo que sucede es que el conjunto de datos cliente delega la operación en el proveedor, y éste genera la siguiente instrucción:

```
select CUSTNO, COMPANY, ADDR1, ADDR2, CITY, STATE, ZIP, COUNTRY,
       PHONE, FAX, TAXRATE, CONTACT, LASTINVOICEDATE
from   CUSTOMER
where  CUSTNO = ?
```

Lo que puede chocarle es cómo puede el proveedor ejecutar esa consulta, si el componente al que está enlazado está configurado con otra sentencia SQL diferente. Pero la respuesta es sencilla: *TSQLQuery* (en realidad, todos los conjuntos de datos que podemos utilizar con DataSnap) implementa internamente una interfaz llamada *IProviderSupport* que, entre otros métodos, define el siguiente:

```
function IProviderSupport.PSExecuteStatement(const ASQL: string;
      AParams: TParams; ResultSet: Pointer = nil): Integer;
```

En particular, la implementación del método en *TSQLQuery* se hereda de la clase *TCustomSQLDataSet*:

```
function TCustomSQLDataSet.PSExecuteStatement(
      const ASQL: string; AParams: TParams;
      ResultSet: Pointer = nil): Integer;
```

```

begin
  if Assigned(ResultSet) then
  begin
    TDataSet(ResultSet^) := TCustomSQLDataSet.Create(nil);
    Result := FSQLConnection.Execute(ASQL, AParams, ResultSet);
  end
  else
    Result := FSQLConnection.Execute(ASQL, AParams);
end;

```

Es decir, el proveedor pasa la instrucción a la consulta DB Express, y la consulta se lava las manos y la deja en el escritorio de la conexión, para que esta última se busque la vida. Como este mundo es muy injusto y cruel, el viaje de la patata caliente que acabo de describir se repetirá muchas veces, como veremos en el siguiente capítulo.

### ACLARACION

La historia que acabo de contarle depende de que el proveedor tenga el valor *False* en su propiedad *ResolveToDataSet*. Pero esa es la situación normal cuando el conjunto de datos original accede a una base de datos SQL.

Otra diferencia entre *RefreshRecord* y *Refresh* a secas es que el primero provoca el disparo de ciertos eventos, tanto en el cliente como en el proveedor. En concreto, los eventos disparados son *BeforeRowRequest* y *AfterRowRequest*. Todos ellos pertenecen al tipo *TRemoteEvent*:

```

type
  TRemoteEvent = procedure (Sender: TObject;
    var OwnerData: OleVariant) of object;

```

El parámetro *OwnerData* sirve para que pasemos información desde el conjunto de datos cliente al proveedor, antes de ejecutarse la operación, y del proveedor al conjunto de datos cliente, al terminar la relectura. Tropezaremos muchas veces con este tipo de viaje de ida y vuelta al estudiar la grabación de datos, y veremos que tiene muchas aplicaciones. Pero no se me ocurre ningún uso particular durante la relectura de registros.

### ADVERTENCIA

Como ya va siendo habitual, he encontrado un par de *bugs* al experimentar con *RefreshRecord*. Esta vez se trata de problemas con ADO Express. En primer lugar, parece que la relectura funciona mal cuando hay campos definidos con el atributo **identity** de SQL Server en el conjunto de datos de origen. Cuando *RefreshRecord* intentaba sustituir los valores de los campos en el registro activo, *TClientDataSet* protestaba diciendo que el campo identidad no podía ser modificado. Logré salvar la situación eliminando el campo original de tipo *TAutoIncField* en el conjunto de datos SQL y sustituyéndolo por un *TIntegerField*. Pero entonces dejó de funcionar el método. Apparentemente, la petición de relectura llega al proveedor, pero la consulta que se genera es incorrecta, y no cambia el contenido del conjunto de datos cliente.

## Resolución

**Y**O QUERÍA ESTUDIAR FÍSICA, no Informática, lo juro. Pero el día de la matrícula, la secretaria tenía un dolor terrible de muelas, y me dio una documentación incorrecta, en la que no quedaban plazas para mi deseada carrera. Desde entonces, he envidiado a los físicos: pueden hablar del radio de Schwarzschild, de las interpretaciones de la función de onda, del efecto túnel y de la ruptura de la simetría en las interacciones débiles. En cambio, los informáticos estamos condenados a hablar de SOAP, COM, CORBA y otras siglas que suenan más a partidos políticos que a otra cosa. Pronuncie esta palabra en voz alta: “resolución”... ¿a quién demonios podría tentarle leer un capítulo sobre “resolución”?

### Grabación de datos con proveedores

Tampoco es para tanto: en inglés, se utiliza el término *resolution* para el proceso de grabación de los cambios realizados en un conjunto de datos cliente. Para ser más precisos, la resolución es un parte muy específica de la grabación, durante la cual un subcomponente, gestionado desde el proveedor y llamado *resolver* (*resolutor*<sup>29</sup>), genera automáticamente las sentencias SQL necesarias para la sincronización de la base de datos con los datos modificados por el usuario. Pero esa fase es tan importante que no me importa extender su nombre a todo el algoritmo.

Voy a mostrarle una lista muy simplificada de los pasos que ocurren durante la grabación de las actualizaciones. Dejaré fuera, principalmente, los eventos que se disparan en varios momentos del algoritmo, y que pueden modificar considerablemente el funcionamiento del mismo.

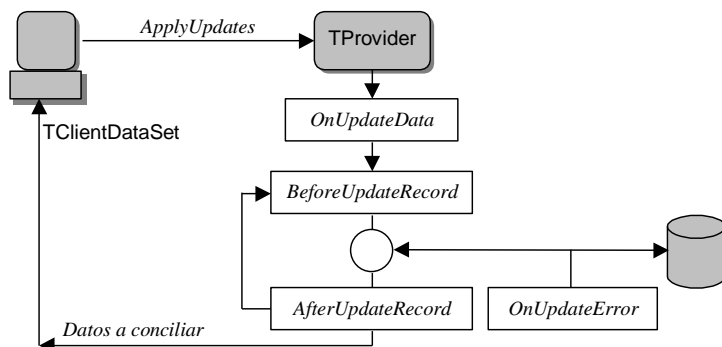
- 1 El usuario ha realizado una serie de cambios en la copia local almacenada en el conjunto de datos cliente. Esos datos se guardan dentro de la propiedad *Delta*. Para aplicar las modificaciones sobre la base de datos original, el usuario toma la iniciativa llamando al método *ApplyUpdates* de *TClientDataSet*.
- 2 *ApplyUpdates* envía el contenido de *Delta* al proveedor, llamando de forma remota o local, según corresponda, a uno de los métodos de *TDataSetProvider*.

---

<sup>29</sup> Ya sé que *resolutor* no existe en castellano. Bueno, ¿y qué?

- 3 El proveedor intenta activar una transacción en la base de datos a la que está indirectamente conectado; si ya existe una, la aprovecha sin más.
- 4 A partir de *Delta*, el proveedor crea un conjunto de datos cliente al vuelo, y recorre cada una de sus filas. La información contenida en cada registro depende del tipo de actualización efectuada: si es una inserción, aparecen los nuevos valores de los campos, si es un borrado, el valor de los campos del registro que desaparece. Si se trata de una modificación, aparecen los valores anteriores y posteriores de cada campo.
- 5 Para cada fila modificada, el proveedor intenta actualizar la base de datos. Si su propiedad *ResolveToDataSet* vale *True*, los cambios se efectúan mediante llamadas a los métodos *Locate*, *Edit*, *Post*, etcétera, del conjunto de datos de origen. Pero es más frecuente que el valor de esa propiedad sea *False*. En ese caso, un subcomponente, el resolutor, genera instrucciones SQL que se ejecutan sobre la base de datos a través de la interfaz *IProvider.Support* del conjunto de datos original.
- 6 Durante la fase anterior pueden producirse varios errores o, si la operación tiene éxito, puede que haya que releer el registro modificado o insertado, para reflejar correctamente las modificaciones adicionales producto de *triggers* y valores por omisión. Esta información se añade al vector *Delta* y se envía de vuelta al conjunto de datos cliente.
- 7 Finalmente, el conjunto de datos cliente recorre el vector recibido para actualizar su copia local. Si hay errores, estos deben ser mostrados al usuario, de una forma u otra. Y, para los registros que se han modificado exitosamente, se comprueba si alguno de los campos ha sido modificado posteriormente por un *trigger* o un valor por omisión. Esta fase se llama *reconciliation*, término que traduciremos como *conciliación*.

El siguiente gráfico muestra los eventos más importantes que dispara el proveedor durante la actualización. He dejado fuera intencionalmente los eventos más evidentes: *BeforeApplyUpdates* y *AfterApplyUpdates*, que se disparan tanto en el conjunto de datos clientes como en el proveedor.



## Un ejemplo muy simple de grabación

Para que no se asuste demasiado, vamos a mostrar un ejemplo muy sencillo de grabación. Será tan simple que utilizaremos como conjunto de datos original una tabla de Paradox, conectada a través del BDE.

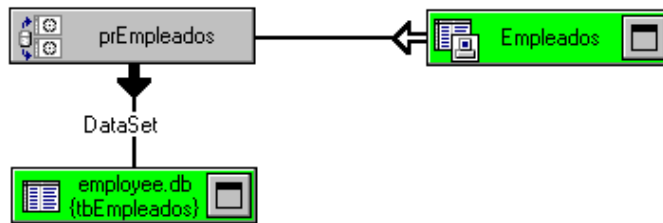
Cree una aplicación nueva, con una ventana principal y un módulo de datos. Añada en el módulo de datos un componente *TTable* de la página *BDE*. Cambie su nombre a *tbEmpleados*, la propiedad *DatabaseName* a *dbdemos*, y *TableName* a *employee.db*. Observe que ni siquiera vamos a traer un componente para la base de datos. Deje cerrada la tabla.

A continuación, traiga un *TDataSetProvider*, desde la página *Data Access*. Llámelo *prEmpleados*, y modifique su propiedad *DataSet* para que apunte a la tabla que hemos traído antes. Muy importante: cambie el valor de *ResolveToDataSet* a *True*.

### NOTA

La propiedad *ResolveToDataSet* solamente debe valer *True* cuando el soporte SQL del conjunto de datos original no existe o no es bueno, como sucede con Paradox y dBase.

Por último, añada un *TClientDataSet* al módulo. Cambie su nombre a *Empleados*, y asigne *prEmpleados* en su propiedad *ProviderName*. Esta vez, deje activo el conjunto de datos cliente. Como resultado, los componentes del módulo de datos quedarán relacionados de la siguiente manera:



Nos vamos entonces a la ventana principal, en la que pondremos componentes *TDataSource* y *TDBGrid*, y los enlazaremos al conjunto de datos cliente que se encuentra en el módulo de datos. Prepare también una barra de herramientas y una lista de acciones. Cree una nueva acción en esta última, y llámela *FicheroGuardar*, porque la utilizaremos para propagar los cambios que efectuemos en el conjunto de datos cliente a la tabla original de Paradox.

La respuesta al evento *OnUpdate* de la acción debe ser:

```

procedure TwndPrincipal.FicheroGuardarUpdate(Sender: TObject);
begin
    with modDatos.Empleados do

```

```

        TAction(Sender).Enabled := Modified or (ChangeCount > 0);
    end;

```

Al igual que hicimos al estudiar la actualización en ficheros de MyBase, comprobamos dos condiciones para saber si existen cambios:

- 1 Consultamos si la propiedad *Modified* es verdadera, para ver si hay cambios no grabados aún en el registro activo.
- 2 Preguntamos si el valor de *ChangeCount* es mayor que cero, para averiguar el número de registros modificados en *Delta*.

Donde si habrá diferencias respecto a MyBase es en la grabación de los cambios:

```

procedure TwndPrincipal.FicheroGuardarExecute(Sender: TObject);
begin
    if modDatos.Empleados.ApplyUpdates(0) > 0 then
        Abort;
end;

```

El prototipo del nuevo método, *ApplyUpdates*, es:

```

function TClientDataSet.ApplyUpdates(MaxErrors: Integer): Integer;

```

El valor de retorno es el número de errores encontrados durante la grabación. El número máximo de errores tolerables durante la grabación lo determina el valor del parámetro *MaxErrors*:

- Si indicamos -1, la operación tolera cualquier número de errores.
- Si indicamos 0, se detiene al producirse el primer error.
- Si indicamos  $n > 0$ , se admite ese número de errores antes de abortar todo el proceso.

¿Por qué hablamos de varios errores? ¿Tiene algún sentido permitir que alguien se equivoque varias veces antes de comunicárselo? Pues sí, tiene sentido. Piense en cómo funciona la detección de errores en el compilador de Delphi: éste intenta encontrar todos los errores posibles, para que usted pueda modificar la mayor cantidad de ellos en una sola revisión. Las primeras versiones de Turbo Pascal se detenían al encontrar el primer error, pero en cuanto evolucionó el producto se introdujo la detección múltiple, que ya implementaban otros compiladores, por su innegable utilidad.

Con las grabaciones en DataSnap sucede algo similar. En primer lugar, el conjunto de datos cliente puede acumular modificaciones en varios registros, y todas ellas son enviadas para su grabación en una misma operación. Cuando el proveedor se encuentra en un servidor remoto de capa intermedia, el envío de los datos consume un tiempo importante. Por lo tanto, es ventajoso detectar la mayor cantidad de fallos durante una sola operación. Por otra parte, no es tan complicado implementar esta característica como lo fue la detección de múltiples errores en los compiladores,



porque hay pocas relaciones mutuas entre los diferentes registros enviados desde el cliente al proveedor.

Más adelante veremos cómo funciona la recepción de estos múltiples mensajes de error, pero quiero adelantarle que una de las consecuencias es que *ApplyUpdates* no provoca excepciones, ni siquiera cuando se producen errores de grabación. Si queremos enterarnos de la existencia de esos errores debemos, al menos, crear un manejador para el evento *OnReconcileError* del conjunto de datos cliente:

```
procedure TmodDatos.EmpleadosReconcileError (
    DataSet: TCustomClientDataSet; E: EReconcileError;
    UpdateKind: TUpdateKind; var Action: TReconcileAction);
begin
    Application.ShowException(E);
end;
```

Es decir, nos limitamos a mostrar el mensaje que porta el objeto de excepción. Para este propósito podríamos haber utilizado cualquier otra función que muestre mensajes, pero decidí utilizar *ShowException* para que los errores de grabación se mostrasen en el mismo estilo que las excepciones internas. Por desgracia, *ShowException* debe aplicarse sobre el objeto global *Application*, que para las aplicaciones GUI es el declarado en unidad *Forms*, que hay que añadir a la cláusula **uses** del módulo de datos.

Debe tener en cuenta que, cuando se ejecuta el manejador de eventos anterior, aún no ha concluido la ejecución de *ApplyUpdates*. Observe que ejecuto el procedimiento *Abort* al terminar *ApplyUpdates* si se descubren errores. El motivo: cumplir con la regla de que un proceso fallido debe terminar con una excepción... aunque sea la excepción silenciosa, que no muestra mensajes.

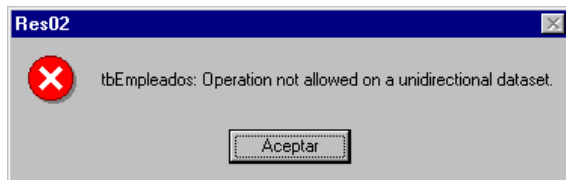
## Resolución SQL

Veamos ahora un ejemplo ligeramente más completo. La mayor complejidad consistirá en que utilizaremos DB Express como origen de nuestros datos, en vez de Paradox. Haga una copia del proyecto anterior en un nuevo directorio, añada un componente *TSQLConnection*, y conéctelo a la conocida base de datos de InterBase:

*C:\Archivos de programa\Archivos comunes\Borland Shared\Data\MastSql.gdb*

Sustituya entonces el componente *TTable* por un *TSQLTable*, pero conecte también este último a la tabla *EMPLOYEE* de la nueva base de datos. Vuelva a relacionar el proveedor con el *TSQLTable* y cree nuevamente los objetos de acceso a campos, tanto en la tabla como en el conjunto de datos cliente, porque cambian los tipos de algunos campos.

Las diferencias reales aparecen ahora. En primer lugar, nos será imposible dejar la propiedad *ResolveToDataSet*, del proveedor, con el valor *True*. Si intentamos grabar algún cambio con ese valor, recibiremos un mensaje de error:



El problema es que cuando *ResolveToDataSet* está activa, el proveedor utiliza directamente el conjunto de datos origen para la grabación. Por ejemplo, si le informan que el empleado número 2 ha sido despedido, intentará localizar ese registro en la tabla llamando a *Locate*. Como sabemos, DB Express no lo permitirá.

Por lo tanto, debemos cambiar *ResolveToDataSet* de nuevo a *False*, su valor inicial. Con este valor, el proveedor genera instrucciones SQL para cada registro modificado y las ejecuta sobre la base de datos a través de una interfaz auxiliar, *IProviderSupport*, implementada por *TSQLTable* y por todo conjunto de datos decente.

#### ADVERTENCIA

Incluso cuando un conjunto de datos SQL permite las llamadas a *Locate*, *Edit*, *Post* y demás, *ResolveToDataSet* debe seguir siendo *True*. Recuerde que el conjunto de datos de origen, cuando el proveedor recibe el paquete de grabación, debe encontrarse cerrado. Para poder grabar con la técnica de grabación directa, es necesario abrir el conjunto de datos, y recuperar registros desde el servidor SQL que son innecesarios en ese momento. Por el contrario, para realizar las actualizaciones indirectas con *IProviderSupport* sólo es necesario que la conexión a la base de datos esté activa. Dicho sea de paso, WebSnap padece este mismo problema, pues por el momento solamente da soporte para grabaciones directas.

Por último, el proveedor tiene una propiedad *UpdateMode*, al igual que los conjuntos de datos del BDE. Cambie su valor a *upWhereChanged*, y ahora le explicaré el motivo.

## La identificación de la tabla base

Hemos quedado en que el nuevo proveedor debe generar instrucciones SQL para actualizar registros. Sólo hay tres tipos de instrucciones para ese propósito: **insert**, **update** y **delete**. Para simplificar la explicación, nos concentraremos en la última de ellas, suponiendo que los únicos cambios consisten en registros eliminados. La instrucción **delete** de SQL estándar es muy sencilla:

```
delete from <NombreTabla>
where    <Condición>
```

Así que el proveedor, para generar la instrucción SQL correspondiente a un borrado, debe conocer el nombre de la tabla de la que eliminará el registro y, lo más importante, la condición que identifica al registro.

A primera vista, puede parecer muy sencillo obtener el nombre de la tabla. ¡Es que, en nuestro ejemplo, el conjunto de datos de origen es precisamente un componente basado en tablas! Sin embargo, piense en que podemos utilizar una consulta como origen de los datos. Incluso, la consulta podría corresponder a un encuentro entre varias tablas.

Por ejemplo, supongamos que queremos mostrar los datos de determinados productos que se encuentran en la tabla *PARTS* de la base de datos del ejemplo que estamos desarrollando. Cada producto almacena un código de proveedor, que hace referencia a un registro de la tabla *VENDORS*. ¿Qué hacemos si es necesario mostrar el nombre del proveedor para cada producto? Si estuviéramos trabajando con Paradox, crearíamos campos de referencia. Pero no es una buena idea para bases de datos SQL, y cuando se utilizan servidores remotos de capa intermedia, hay que descartarla completamente, a no ser que la tabla de referencia tenga muy pocas filas.

La mejor solución consiste en realizar un encuentro natural:

```
select p.*, v.VendorName
from PARTS p inner join VENDORS v
on p.VendorNo = v.VendorNo
```

Claro, aunque teóricamente este tipo de encuentros es perfectamente actualizable, en algunos sistemas como el BDE no lo es directamente. Sin embargo, cualquier persona se daría cuenta enseguida de que eliminar un registro de esa consulta “significa” eliminarlo de la tabla de productos; no de la de proveedores. Yo lo sé, usted lo sabe, pero ¿cómo se lo decimos al *TDataSetProvider*?

En realidad, no hace falta decírselo en la mayoría de los casos, pues el componente proveedor es capaz de averiguarlo por sí mismo. Más exactamente, con la ayuda de la interfaz *IProviderSupport* que implementa el conjunto de datos de origen. El proveedor puede llamar al siguiente método:

```
type
  IProviderSupport = interface
    // ...
    function PSGetTableName: string;
    // ...
end;
```

En muchos casos, el propio componente de consulta es capaz de determinar sin ayuda cuál es la tabla que hay que actualizar, realizando un sencillo análisis de la instrucción SQL asociada. Aunque en BDE, ADO y DB Express el código correspondiente se encuentra escondido dentro del controlador, tengo la sospecha de que se

utiliza el primer identificador que aparece a continuación de la primera palabra reservada **from** (puede haber más cláusulas **from** en subconsultas).

Si las cosas se ponen de color de hormiga y vemos que el *TDataSetProvider* usa la tabla equivocada, podemos corregirlo interceptando su evento *OnGetTableName*:

```
// ;;;Para nuestro ejemplo, es innecesario!!!
procedure TmodDatos.prEmpleadosGetTableName(Sender: TObject;
  DataSet: TDataSet; var TableName: string);
begin
  TableName := 'EMPLOYEE';
end;
```

Pero no se preocupe demasiado por este problema teórico porque, como he dicho, los conjuntos de datos pueden dar la respuesta acertada en la mayoría de las consultas. Es más probable que necesitemos el evento *OnGetTableName* cuando utilizamos un procedimiento almacenado como conjunto de datos de origen.

Aunque al principio de la sección aclaré que iba a limitarme de momento a los borrados, está claro que la identificación de la tabla a actualizar también es importante para grabar inserciones y actualizaciones.

## Configuración de los campos

Una vez que hemos identificado la tabla a actualizar, tenemos que saber cómo el proveedor genera la condición de búsqueda. Hay tres variantes posibles, y corresponden a los valores que puede tomar la propiedad *UpdateMode* del proveedor:

Valor	Significado
<i>upWhereAll</i>	Se incluyen todos los campos en la condición
<i>upWhereChanged</i>	Sólo la clave primaria y los campos modificados
<i>upWhereKeyOnly</i>	Sólo la clave primaria

Es decir: al proveedor llega un registro marcado para ser borrado, y podemos saber cuáles eran los valores originales de sus columnas. Si *UpdateMode* vale *upWhereAll*, la condición de búsqueda sería de la forma siguiente:

```
EmpNo = 2 and LastName = 'Nelson' and
FirstName = 'Roberto' and PhoneExt = '250' and ...
```

Todos esos campos se incluyen para evitar que eliminemos un registro desde un ordenador si se han realizado modificaciones sobre el mismo, desde otro ordenador, a partir del momento en que lo leímos por primera vez. Si, por ejemplo, leemos el registro anterior y alguien cambia el número de la extensión telefónica antes de que podamos releerlo, se produciría un error si intentamos borrarlo, porque la condición generada no lo encontraría.

Pero hay problemas a la vista. La tabla de nuestro ejemplo solamente tiene seis columnas, pero una tabla real tiene generalmente muchas más. No sólo eso, sino que los valores de sus campos pueden ocupar muchos caracteres, especialmente en las columnas de tipo **varchar**. ¿Se da cuenta de que utilizar *upWhereAll* podría consumir demasiado ancho de banda de la conexión con el servidor SQL?

Por consiguiente, es preferible casi siempre que *UpdateMode* tenga el valor *upWhereChanged* o *upWhereKeyOnly*. Con cualquiera de estos dos, la instrucción de eliminación debería ser:

```
delete from EMPLOYEE
where EmpNo = 2
```

Aunque para los borrados son equivalentes los dos valores mencionados, en el caso de modificaciones de registros existentes es preferible trabajar con *upWhereChanged*. De esta forma, el usuario puede saber si intenta modificar una columna que otro usuario acaba de retocar en otro puesto; algo que sería imposible con *upWhereKeyOnly*. Esto lo explicaremos más adelante.

¿Cómo puede el proveedor determinar qué campos forman parte de la clave primaria? Ya tropezamos con este problema en el capítulo anterior, al estudiar cómo releer registros individuales desde un conjunto de datos cliente. Vimos que, en ciertos casos, el propio conjunto de datos de origen puede echarle una mano al proveedor. Pero que lo más seguro era configurar la propiedad *ProviderFlags* de cada uno de los campos originales. Para mayor conveniencia, mostraré de nuevo las opciones de esa propiedad:

Opción	Significado
<i>pfInUpdate</i>	Indica si el campo es actualizable o no
<i>pfInWhere</i>	Indica si debemos utilizar el campo para localizar registros
<i>pfInKey</i>	Indica si el campo pertenece a la clave primaria
<i>pfHidden</i>	Cuando está activa, el campo no es visible desde el cliente

Lo principal a tener en cuenta es que las *ProviderFlags* las determinamos nosotros, no el componente SQL. Por omisión, todos los campos comienzan teniendo activas las opciones *pfInUpdate* y *pfInWhere*, y es el programador quien tiene que quitar o añadir opciones, de acuerdo a la forma en la quiera que se produzcan las actualizaciones.

Para terminar el ejemplo que hemos estado desarrollando, por lo tanto, sólo tenemos que buscar el campo *EmpNo* en el componente *tbEmpleados*, seleccionar su propiedad *ProviderFlags* en el Inspector de Objetos y activar la opción *pfInKey*. Si tuviéramos una tabla con una clave compuesta, tendríamos que activar la misma opción en todos los campos correspondientes. Con este paso, ya tenemos listo el ejemplo.

#### NOVEDAD

Delphi 6 propaga el valor de *ProviderFlags* a los campos del conjunto de datos cliente, algo que no se hacía en versiones anteriores. Esto es muy útil para determinar auto-

máticamente qué campos conforman la clave en la capa visual de una aplicación en múltiples capas.

## Más usos de *ProviderFlags*

Las opciones de *ProviderFlags* tienen otros usos igualmente interesantes. Por ejemplo, *pfInWhere* se utiliza para excluir las columnas que no queremos que aparezcan, en modo alguno, en una cláusula **where** de localización. Si el valor de *UpdateMode* en el proveedor es *upWhereKeyOnly*, no tiene mucho sentido modificarla, porque se utilizarán siempre los mismos campos en esa condición: los de la clave primaria. Supondremos entonces que se está utilizando uno de los otros dos valores.

¿Por qué querríamos excluir una columna de la condición de búsqueda? La explicación más frecuente es la existencia de columnas de tipo blob. No conozco un solo sistema SQL que permita comparar el contenido de dos valores de este tipo, por lo que es lógico excluir esas columnas de la cláusula **where**. Sin embargo, la explicación anterior ignora que el propio proveedor detecta esos tipos de campos y los descarta de la condición de búsqueda. Recuerde que *TField* declara una función con ese propósito en mente:

```
function TField.IsBlob: Boolean;
```

De cualquier manera, siempre es recomendable desactivar *pfInWhere* explícitamente para las columnas de tipo blob.

En realidad, hay otros dos motivos para modificar la mencionada opción. La primera tiene que ver con la representación de valores reales, y en realidad es un apaño para resolver un *bug* difícil de reproducir. Suponga que realizamos una actualización en un registro de una tabla con una columna de tipo flotante, y que *UpdateMode* está configurada con el valor *upWhereAll* o *upWhereChanged*. Puede suceder, debido a los problemas de representación exacta de los tipos flotantes, que la condición de búsqueda no encuentre el registro al existir alguna pequeña diferencia entre el valor real de la columna y el valor pasado en la instrucción SQL. Este problema lo encontré en una aplicación que utilizaba ADO Express y Midas sobre Access. Si se le presenta este tipo de problemas, intente solucionarlo quitando *pfInWhere* en el campo conflictivo.

El segundo motivo es más legítimo. ¿Recuerda el ejemplo que mencioné en la sección anterior sobre la actualización de encuentros naturales? La instrucción era la siguiente:

```
select p.*, v.VendorName
from PARTS p inner join VENDORS v
on p.VendorNo = v.VendorNo
```

La columna *VendorName* se trae solamente con fines informativos, y es conveniente desactivar quitar *pfInWhere*, pero también *pfInUpdate*, en el campo correspondiente, que se quedará con un conjunto vacío en su propiedad *ProviderFlags*.

La opción *pfInUpdate* debe también eliminarse en los campos de tipo autoincremental, como los que tienen el atributo **identity** en SQL Server, y en aquellos que obtienen siempre su valor por medio de *triggers*. Por ejemplo, una tienda en Internet puede definir una tabla de palabras claves para la búsqueda como la siguiente:

```
create table PALABRAS (
    IDPalabra integer not null primary key,
    Palabra varchar(30) not null,
    Normalizada varchar(30) not null unique
);
```

La columna *Normalizada* es actualizada mediante *triggers*, y debe contener una versión en mayúsculas y sin acentos del valor almacenado en *Palabra*. Es entonces evidente que *Normalizada* no debe ser modificada desde la interfaz del usuario, y que conviene quitar tanto *pfInUpdate* como *pfInWhere* del objeto de acceso asociado.

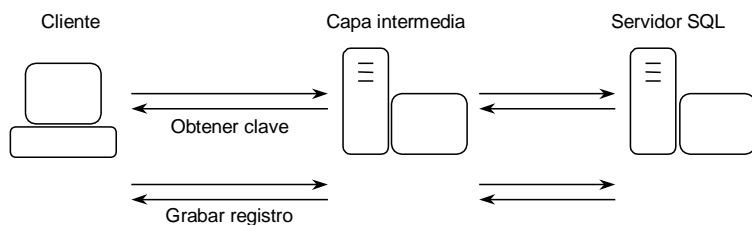
Por último, está la opción *pfHidden*. Se utiliza para que determinados campos no aparezcan en el lado cliente, aunque sí se incluyan en los paquetes enviados o recibidos por el proveedor. Puede servir para ocultar campos autoincrementales, en los casos en que no son necesarios en la capa visual.

## Asignación automática de claves

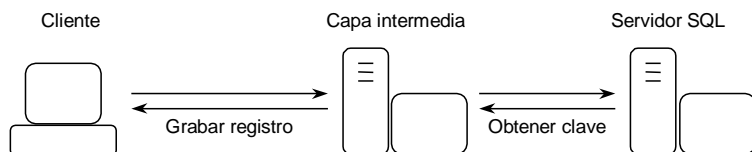
Con los conocimientos que acabamos de adquirir, podemos ya realizar la grabación de cualquier conjunto de datos simple, es decir, que no esté configurado en la relación maestro/detalles. Pero con una condición: que seamos nosotros quienes tecleemos los valores de todas sus columnas en el lado cliente de la aplicación.

Vamos a ver ahora otro caso: cuando la clave primaria debe asignarse de forma automática. Tomemos, como ejemplo, la tabla *CUSTOMER* de la base de datos *mastsql.gdb*. Su clave primaria es la columna *CustNo*, y aparentemente debemos asignar su valor usando una tabla de contadores, llamada *NEXTCUST* y ubicada en la misma base de datos. Hay un problema: en una base de datos bien diseñada, tendríamos un procedimiento almacenado que obtuviera un valor e incrementase el contador. Pero *mastsql* es una vieja chapuza ya sin remedio, así que tendremos que realizar toda la operación explícitamente.

Supongamos ahora que la aplicación que realiza el mantenimiento de esa tabla está dividida en dos capas físicas. ¿Cómo asignar valores a *CustNo* procedentes de la tabla *NEXTCUST*? Podríamos pedir desde el cliente el próximo valor al servidor SQL, y asignarlo a la columna *CustNo* en el conjunto de datos clientes durante el evento *OnNewRecord*, o quizás en *BeforePost*. Cuando estuviese listo el nuevo registro, lo enviaríamos al proveedor para su grabación en la base de datos SQL.



Pero esta técnica tiene un inconveniente grave: se realizan dos peticiones independientes desde el cliente final hacia el servidor de capa intermedia. Y eso consume tiempo, y ancho de banda. Lo ideal sería que el nuevo registro abandonase el lado cliente *sin* el valor definitivo de la clave, y que ésta se asignase durante la resolución, en el servidor de capa intermedia. Así solamente tendríamos un viaje de ida y vuelta a través de la red.



Vamos a iniciar entonces una nueva aplicación, para demostrar cómo hacer funcionar esta idea. Traiga un *TSQLConnection* al módulo de datos y conéctelo a la conocida base de datos de ejemplos. Añada un *TSQLTable*, conéctelo a la tabla *CUSTOMER* y bautícelo como *tbClientes*. Y cree los objetos de acceso a campo, recordando asignar la opción *pfInKey* en las *ProviderFlags* del campo *CustNo*. Tampoco estaría mal que asignase el valor *-1* en la propiedad *DefaultExpression* del mismo objeto de acceso a campos.

A continuación prepararemos el módulo para obtener valores automáticos para la clave primaria de la tabla de clientes. Traiga al módulo un *TSQLQuery*, cambie su nombre a *qrNextCust* y configúrelo con la siguiente instrucción SQL:

```

select NewCust
from   NEXTCUST
  
```

Declare entonces un método privado en la clase del módulo de datos, como éste:

```

function TmodDatos.ProximoValor: Integer;
begin
  SQLConnection1.ExecuteDirect(
    'update NEXTCUST set NEWCUST = NEWCUST + 1');
  qrNextCust.Open;
  try
    Result := qrNextCustNEWCUST.AsInteger - 1;
  finally
    qrNextCust.Close;
  end;
end;
  
```



**ADVERTENCIA**

Hay algo chocante en la implementación de *ProximoValor*. En primer lugar, estamos ejecutando dos instrucciones dependientes entre sí, ¡y no hay señales perceptibles de transacciones! Además, es muy raro incrementar primero el contador y recuperar su valor después. Ambos hechos serán explicados en la siguiente sección.

Ha llegado el momento de sumar un *TDataSetProvider* a la fiesta; lo llamaremos *prClientes*. Asíelo a la correspondiente consulta, y cambie el valor de *UpdateMode* para que sea *upWhereChanged*. Finalmente, y esto es *muy* importante: en su propiedad *Options* debe activar la ya conocida *poIncFieldProps* y la nueva *poPropagateChanges*. Un poco de paciencia, que ya explicaré el papel de la segunda opción.

Por primera vez, interceptaremos un evento del proveedor, *BeforeUpdateRecord*:

```
procedure TmodDatos.prClientesBeforeUpdateRecord(Sender: TObject;
  SourceDS: TDataSet; DeltaDS: TCustomClientDataSet;
  UpdateKind: TUpdateKind; var Applied: Boolean);
begin
  if UpdateKind = ukInsert then
    DeltaDS.FieldByName('CUSTNO').NewValue := ProximoValor;
end;
```

*BeforeUpdateRecord* se dispara cada vez que se va a generar la instrucción SQL de actualización de cada uno de los registros enviados al proveedor para su sincronización. El parámetro *UpdateKind* nos indica si se trata de una inserción, un borrado o una modificación; como ve, sólo nos interesan ahora las inserciones. En *SourceDS* viene un puntero al conjunto de datos que corresponde al registro actual. Esta información es imprescindible durante la resolución de relaciones maestro detalles, porque el mismo proveedor se ocupará de ambos conjuntos de datos.

Los datos del registro a insertar, sin embargo, vienen en el parámetro *DeltaDS*, que es un conjunto de datos cliente creado “al vuelo” por el proveedor, al inicio del algoritmo de resolución. Debe comprender que, al ser un componente dinámico, no existen campos persistentes definidos sobre *DeltaDS*. Por este motivo, utilizamos el método *FieldByName* para acceder a la información correspondiente a la columna *CustNo*. La otra novedad es que la clave que genera *ProximoValor* no se asigna donde podríamos pensar que es más lógico, en *Value*, o en *AsInteger*, sino en la propiedad *NewValue*:

```
DeltaDS.FieldByName('CUSTNO').NewValue := ProximoValor;
```

Esto es así porque durante la resolución los valores que nos interesan realmente se encuentran en las propiedades *OldValue* y *NewValue* de los campos de *DeltaDS*. Como en este caso se trata de una inserción, *OldValue* no tiene sentido alguno. Note también que no tenemos que llamar ni a *Edit*, para poner *DeltaDS* en modo de edición, ni a *Post*, para confirmar el valor que hemos asignado. Esta es otra característica especial del conjunto de datos utilizado durante la resolución.

Como comprenderá enseguida, hemos programado una especie de *trigger* en la capa intermedia de la aplicación. ¿Por qué, por amor de Buda, no nos dejamos de pampalinas y utilizamos un *trigger* de verdad? Por una parte, estamos moviendo tiempo de procesamiento desde el servidor SQL hacia el ordenador donde se ejecuta el servidor de capa intermedia; eso es un alivio, en la mayoría de los casos. Así, además, hacemos que nuestra aplicación dependa menos de las posibilidades del servidor SQL que estamos utilizando. Recuerde, por ejemplo, que en SQL Server los *triggers* siempre se disparan *después* de ejecutarse la operación.

Pero el motivo principal es que con un *trigger* tendríamos problemas, con toda seguridad, para releer el registro recién grabado y averiguar el valor de la clave primaria que le ha tocado en suerte. Con nuestro sistema, ese valor se obtiene de antemano y, lo más importante:

*“Cuando poPropagateChanges está activo, los valores que se asignen a los campos durante la resolución se propagan al conjunto de datos cliente durante la conciliación”*

En el ejemplo completo del CD, aprovecho para mostrar mi técnica favorita de actualización: utilizo un cuadro de diálogo modal para editar los registros individuales. De ese modo, puedo hacer que los cambios se envíen al proveedor cuando el usuario final pulsa el botón *Aceptar*. Y así evito tener que poner un comando para llamar explícitamente al método *ApplyUpdates*. Dígame a un usuario suyo que no puede olvidarse de pulsar un botón para que su aplicación funcione correctamente. Es seguro entonces que olvidará hacerlo, y además, le echará la culpa a usted.

Para concluir la parte visual del ejemplo, traiga un *TClientDataSet* al mismo módulo de datos, llámelo *Clientes*, conéctelo al proveedor *prClientes* a través de su propiedad *ProviderName*, cree sus objetos de acceso a campos y finalmente déjelo activo. No olvide tratar su evento *OnReconcileError* del mismo modo en que lo hicimos en el ejemplo anterior:

```
procedure TmodDatos.ClientesReconcileError(
  DataSet: TCustomClientDataSet; E: EReconcileError;
  UpdateKind: TUpdateKind; var Action: TReconcileAction);
begin
  Application.ShowException(E);
end;
```

Ponga una rejilla en la ventana principal y un *TDataSource*, y asócielos a *Clientes*. Cambie la propiedad *ReadOnly* de la rejilla a *True*, para que el usuario no pueda editar directamente sobre ella. Añada una lista de acciones, y traiga a la misma todas las acciones predefinidas de la categoría *DataSet*. Seleccione la acción *DataSetDelete1* y cree un manejador de eventos para su *OnExecute*:

```
procedure TwndPrincipal.DataSetDelete1Execute(Sender: TObject);
begin
  with modDatos.Clientes do
    begin
      Delete;
```

```

        if ApplyUpdates(0) > 0 then
        begin
            CancelUpdates;
            Abort;
        end;
    end;
end;
end;

```

La idea es mezclar en una misma operación el borrado en la caché local y la sincronización con la base de datos SQL. Por este motivo, si se produce un error durante la resolución, se deshace el único cambio que teníamos pendiente: en este caso, el borrado de la fila activa.

Para las altas y modificaciones crearemos una nueva ventana, a la que llamaremos *dlgCliente*, y que configuraremos del siguiente modo:

Observe que he desactivado el cuadro de edición del número de cliente. En la imagen se está creando un nuevo registro. Por ese motivo aparece un -1 en el cuadro de edición mencionado. No se preocupe, porque al grabar el registro recibiremos el valor real asignado.

Cuando el usuario pulsa cualquiera de los botones para cerrar el diálogo, debemos grabar o descartar los cambios, según el botón. Ya sabe que prefiero implementar ese comportamiento durante la respuesta al evento *OnCloseQuery* del formulario:

```

procedure TdlgCliente.FormCloseQuery(Sender: TObject;
    var CanClose: Boolean);
begin
    if ModalResult = mrOk then
    begin
        if modDatos.Clientes.ApplyUpdates(0) > 0 then Abort;
    end
    else
        modDatos.Clientes.CancelUpdates;
    end;

```

Finalmente, volvemos a la ventana principal y damos respuesta al evento *OnExecute*, esta vez de la acción *DataSetEdit1*:

```
procedure TwndPrincipal.DataSetEdit1Execute(Sender: TObject);
begin
  with TdlgCliente.Create(Self) do
    try
      modDatos.Clientes.Edit;
      ShowModal;
    finally
      Close;
    end;
  end;
end;
```

La respuesta a *OnExecute* de *DataSetInsert1* debe ser similar, pero sustituyendo la llamada a *Edit* por una llamada a *Insert* o *Append*; da lo mismo cualquiera de las dos.

## Transacciones y resolución

Aclaremos un par de dudas pendientes. ¿Por qué, en primer lugar, no me he preocupado por meter las dos instrucciones de *ProximoValor* dentro de una transacción? La respuesta es muy simple: cuando un proveedor va a iniciar la resolución comprueba si hemos iniciado explícitamente una transacción en el componente de conexión asociado al conjunto de datos de origen. La pregunta se basa en una llamada al método *PSInTransaction* de la interfaz *IProviderSupport*. Si no hay una transacción activa, el proveedor se encarga de iniciar una. Naturalmente, cuando termina la resolución el proveedor también concluye la transacción. Si todo ha ido bien, con un *Commit*; en caso contrario, con un *Rollback*. Es por eso que no me preocupó por tener dos instrucciones independientes lanzadas dentro de *ProximoValor*, porque sé con total seguridad que el proveedor las ejecutará siempre dentro de una transacción.

Esas eran las buenas noticias; ahora llegan las malas. ¿Se ha fijado en que primero he disparado y después es que he preguntado el inevitable “¿quién anda ahí?”? Como debe saber, la primera operación crea una versión tentativa del registro de la tabla de contadores en InterBase. Esto funciona a efectos prácticos como un bloqueo de escritura, e impide que nadie más lea el contador hasta que no termine la resolución... y con ella la transacción. Todo este ajetreo tiene como objetivo evitar los problemas que ya conoce cuando el aislamiento de las transacciones es *read committed*, no *repeatable read*.

¡Sí, no frunza las cejas! No es que el proveedor inicie siempre una transacción con ese pobre nivel de aislamiento. En general, la transacción se inicia con el nivel de aislamiento que tengamos configurado en la correspondiente propiedad del componente de conexión:

Componente de conexión	Propiedad para el nivel de aislamiento
<i>TDatabase</i>	<i>TransIsolation</i>
<i>TADOConnection</i>	<i>IsolationLevel</i>

Componente de conexión	Propiedad para el nivel de aislamiento
<i>TIBTransaction</i>	<i>Params</i>

¿Y qué pasa con DB Express? Es cierto que existe un parámetro para el nivel de aislamiento, que se especifica en la propiedad *Params* del *TSQLConnection*. Pero ese nivel se refiere a las transacciones implícitas. Recuerde que para las transacciones explícitas, el nivel de aislamiento se debe indicar ¡para cada llamada, en el parámetro *TransDesc*!

```
procedure TSQLConnection.StartTransaction(
    TransDesc: TTransactionDesc);
```

Esto es un disparate mayúsculo, que atenta contra las buenas reglas de diseño en la Programación Orientada a Objetos (pronuncie las mayúsculas). ¿No me cree? Pues mire la implementación, en el código fuente, que el componente *TCustomSQLDataSet* da al evento *PSStartTransaction* de la interfaz *IProviderSupport*:

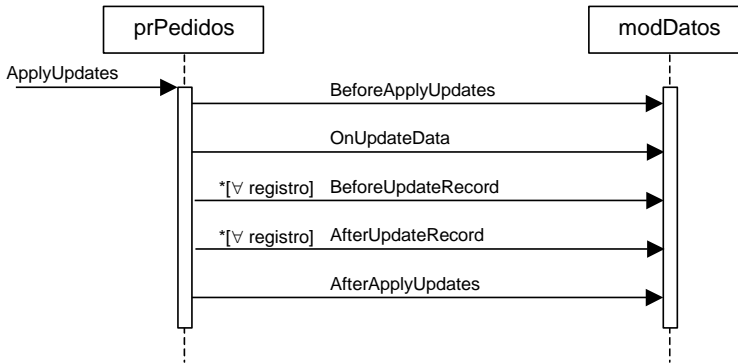
```
procedure TCustomSQLDataSet.PSStartTransaction;
var
    TransDesc: TTransactionDesc;
begin
    FillChar(TransDesc, SizeOf(TransDesc), 0);
    TransDesc.TransactionID := 1;
    FSQLConnection.StartTransaction(TransDesc);
end;
```

El método inicializa el descriptor de la transacción con ceros. Eso significa que el campo *IsolationLevel* del descriptor recibe el valor *xi/READCOMMITTED*. Para colmo, la ayuda en línea ha copiado equivocadamente la declaración del tipo enumerativo con los niveles de aislamiento de DB Express, y sugiere de forma errónea que el nivel utilizado por la transacción es *xi/DIRTYREAD*, que sería incluso peor.

Esto es un problema *mu*y grave que debe resolver Borland. Mientras tanto, tendremos que ocuparnos nosotros mismos de proteger con trucos nuestras transacciones, tal como he mostrado en *ProximoValor*. Pero una alternativa mejor es que nos adelantemos al proveedor e iniciemos la transacción que arrojará a la grabación, con el nivel deseado, como explicaré en la siguiente sección.

## Antes y después

Casi al principio del capítulo incluí un diagrama informal con el flujo de ejecución durante la ejecución. El siguiente diagrama de secuencias amplía los detalles sobre los eventos disparados por el proveedor:



Cuando el proveedor inicia la resolución y detecta que no hay una transacción activa, inicia una nueva, en el intervalo que media entre el disparo de *BeforeApplyUpdates* y *OnUpdateData*, y la cierra antes de llamar a *AfterApplyUpdates*. Por lo tanto, si queremos tomar la iniciativa para forzar el tipo de transacción que nos interesa, el evento que necesitamos es *BeforeApplyUpdates*.

¿Y para terminar la transacción? Está claro que hay que esperar hasta *AfterApplyUpdates*. No obstante, no me siento muy cómodo cuando tengo que pedir un recurso durante el disparo de un evento para liberarlo durante el disparo de otro. Si se produjese una excepción entre ambos, y no se llegara a ejecutar el segundo manejador, perderíamos el recurso.

Afortunadamente, el código que se encuentra entre los eventos mencionados está protegido a prueba de bombas. Como la resolución debe capturar todas las excepciones que se produzcan para reportarlas al cliente, se nos garantiza que, aunque surjan errores, que es muy probable, *siempre* podemos contar con el disparo de *AfterApplyUpdates*.

En este ejemplo, he interceptado tres eventos diferentes de un proveedor, al que he llamado *prEmpleados*, para forzar el uso del nivel de aislamiento superior de InterBase:

```

procedure TmodDatos.prEmpleadosBeforeApplyUpdates(Sender: TObject;
  var OwnerData: OleVariant);
begin
  TD.TransactionID := 1;
  // ;;;Esto era lo que queriamos !!!
  TD.IsolationLevel := xilREPEATABLEREAD;
  SQLConnection1.StartTransaction(TD);
end;

procedure TmodDatos.prEmpleadosAfterApplyUpdates(Sender: TObject;
  var OwnerData: OleVariant);
begin
  TD.TransactionID := 1;
  if SQLConnection1.InTransaction then
    SQLConnection1.Commit(TD);
end;
  
```

```

procedure TmodDatos.prEmpleadosUpdateError(Sender: TObject;
  DataSet: TCustomClientDataSet; E: EUpdateError;
  UpdateKind: TUpdateKind; var Response: TResolverResponse);
begin
  TD.TransactionID := 1;
  if SQLConnection1.InTransaction then
    SQLConnection1.Rollback(TD);
end;

```

La transacción se inicia al principio de la resolución. Pero observe que, si se produce un solo error, anulamos la transacción en la respuesta al evento *OnUpdateError*. Estoy asumiendo que la llamada a *ApplyUpdates* desde el cliente utiliza un cero como parámetro *MaxErrors*. O dicho en otras palabras, que solamente “toleramos” un error antes de detener la grabación. Por lo tanto, si al llegar a *AfterApplyUpdates* detectamos que hay una transacción activa todavía, la confirmamos, porque por fuerza todo ha ido bien.

### ADVERTENCIA

Teóricamente hay otra implementación posible, incluso más atractiva. Consiste en utilizar una variable *FErrorFlag* en el módulo, inicializarla a falso antes de comenzar la resolución, y activarla si detectamos un error. Entonces el evento final es el responsable de llamar a *Commit* o *Rollback*, de acuerdo al valor del indicador. Lamentablemente, las pruebas que hice en este sentido fallaron estrepitosamente: el mensaje de error que se entregaba al cliente aparecía “distorsionado”, y encontré varios problemas durante la conciliación.

## Identidades en SQL Server

Ya sabemos cómo asignar valores secuenciales a la clave primaria de una tabla de InterBase. Pero la técnica equivalente en SQL Server es diferente. Si extraemos los valores de la clave a partir de una tabla de contadores, podríamos hacer lo mismo que en InterBase: en el evento *BeforeUpdateRecord* pediríamos el próximo valor para asignarlo en la propiedad *NewValue* del campo correspondiente. Pero si utilizamos campos con el atributo **identity** se nos complica el asunto, porque hay que grabar primero el registro, sin conocer aún cuál será el valor que recibirá su clave, y sólo posteriormente podremos averiguar el valor asignado.

Este ejercicio asumirá que tenemos acceso a una base de datos de SQL Server 7 o 2000 a través de ADO Express, alias dbGo™. Cree una tabla sencilla, con una clave primaria que tenga el atributo **identity**, como la siguiente:

```

create table CLIENTES (
  IDCliente  integer    not null identity(1,1),
  Nombre     varchar(30) not null,

  primary key (IDCliente)
)
go

```

Prepare una nueva aplicación similar a la del ejemplo basado en InterBase, pero esta vez con un componente de conexión *TADOConnection*, y un *TADOQuery*, al que llamaremos *qrClientes*, y que configuraremos de acuerdo a la instrucción:

```
select *
from   CLIENTES
```

Cuando cree los objetos de acceso a campo de la consulta, debe activar el indicador *pfInKey* del campo *qrClientesIDCliente*. Pero también debe eliminar la opción *pfInUpdate*, porque los campos de identidad no son modificables en SQL Server.

Asocie un proveedor a la consulta anterior, llámelo *prClientes*, recuerde activar sus opciones *poIncFieldProps* y *poPropagateChanges*, y asigne en *UpdateMode* el valor *upWhereChanged*. Esta vez, en vez de interceptar el evento *BeforeUpdateRecord*, crearemos un manejador para *AfterUpdateRecord*:

```
procedure TmodDatos.prClientesAfterUpdateRecord(Sender: TObject;
  SourceDS: TDataSet; DeltaDS: TCustomClientDataSet;
  UpdateKind: TUpdateKind);
begin
  if UpdateKind = ukInsert then
    with DeltaDS.FieldByName('IDCliente') do
      begin
        ReadOnly := False;
        NewValue := UltimaIdentidad;
      end;
    end;
end;
```

El truco se basa en que SQL Server no necesita ayuda para asignar el valor de un campo identidad. Pero el *TClientDataSet*, al otro lado de la línea, necesita saber cuál fue el valor concreto asignado. Por eso modificamos *NewValue* en el campo correspondiente, pero *a posteriori*. El problema es que el campo *IDCliente* pertenece a la clase *TAutoIncField*, que Delphi crea siempre con el atributo *ReadOnly* activo. Por este motivo, primero modificamos el valor de *ReadOnly*, para después poder asignar en *NewValue* el valor de la última identidad.

Este se obtiene con la ayuda de un método auxiliar. Lo más fácil es añadir al módulo una nueva consulta, que llamaremos *qrIdentidad*, y que contendrá la siguiente sentencia SQL:

```
select @@identity
```

La pseudo variable de la instrucción anterior siempre devuelve el último valor de identidad que se ha asignado dentro de nuestra conexión, con total independencia de las tonterías que estén haciendo paralelamente el resto de las conexiones. La implementación del método auxiliar *UltimaIdentidad* es simple:

```
function TmodDatos.UltimaIdentidad: Integer;
begin
  qrIdentidad.Open;
```



```

    try
        Result := qrIdentidad.Fields[0].AsInteger;
    finally
        qrIdentidad.Close;
    end;
end;

```

La única precaución que debemos tomar es evitar triggers durante las inserciones que creen registros en otras tablas que utilicen identidades, porque entonces no podríamos conocer con certeza el último valor asignado por nuestra conexión.

## Tomando la iniciativa

El evento más importante de la resolución es *BeforeUpdateRecord*, porque nos permite tomar la iniciativa y especificar cómo queremos actualizar cada registro, saltándonos incluso el algoritmo de actualización por omisión que implementa el proveedor.

¿Por qué querríamos hacer tal cosa? Piense, por ejemplo, que está utilizando una base de datos de SQL Server 7, en la que todavía no podemos aprovechar las acciones referenciales. Nuestros registros de clientes actúan como maestros de los registros de la tabla de dirección: cada cliente tiene una o más direcciones asociadas, y cuando eliminamos un cliente queremos que desaparezcan también sus direcciones. En InterBase, Oracle y SQL Server 2000 declararíamos una relación de integridad referencial en la tabla de direcciones, pidiendo el borrado en cascada:

```

foreign key (IDCliente) references CLIENTES (IDCliente)
on delete cascade

```

Como sabe, en SQL Server 7 no existía esa posibilidad. Por supuesto, tenemos una primera solución, muy sencilla: cuando se dispara el evento *BeforeUpdateRecord* del proveedor asociado a una consulta de clientes, ejecutamos una instrucción SQL para eliminar las direcciones asociadas:

```

procedure TmodDatos.prClientesBeforeUpdateRecord(Sender: TObject;
    SourceDS: TDataSet; DeltaDS: TCustomClientDataSet;
    UpdateKind: TUpdateKind; var Applied: Boolean);
begin
    if UpdateKind = ukDelete then
        ADOConn.Execute('delete from DIRECCIONES where IDCliente = ' +
            VarToStr(DeltaDS.FieldByName('IDCliente').Value));
end;

```

Después de que se ejecute la eliminación de las direcciones, la actualización del registro sigue su curso, y es el proveedor el que borra el registro principal del cliente. Este es un sistema muy conservador, pero podemos ir más allá y crear un procedimiento almacenado con el objetivo de eliminar un cliente. Por ejemplo:

```

create procedure EliminarCliente @idcliente integer as
begin
    delete from DIRECCIONES
    where IDCliente = @idcliente

```

```

        delete from CLIENTES
        where IDCliente = @idcliente
    end
    go

```

Entonces podríamos dar una respuesta diferente dentro de *BeforeUpdateRecord*:

```

procedure TmodDatos.prClientesBeforeUpdateRecord(Sender: TObject;
    SourceDS: TDataSet; DeltaDS: TCustomClientDataSet;
    UpdateKind: TUpdateKind; var Applied: Boolean);
begin
    if UpdateKind = ukDelete then
        begin
            // El componente spEliminarCliente está asociado
            // al procedimiento almacenado EliminarCliente
            with spEliminarCliente.Parameters do
                ParamByName('@idcliente').Value :=
                    DeltaDS.FieldByName('IDCliente').Value;
            spEliminarCliente.ExecProc;
            Applied := True;           // ¡¡¡IMPORTANTE!!!
        end;
    end;
end;

```

Esta vez, hemos suplantado el algoritmo de borrado completamente, y lo indicamos modificando el valor del parámetro *Applied* al ejecutar la actualización.

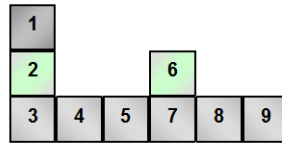
¿Qué ventajas e inconvenientes encontraremos si suplantamos la actualización de este modo? La mayor ventaja es la flexibilidad: con el nuevo sistema, el algoritmo real de eliminación de clientes reside en la base de datos, y es más sencillo corregirlo o ampliarlo que si estuviese codificado dentro de la aplicación escrita en Delphi. Además, podemos delegar su escritura y mantenimiento al equipo de desarrolladores en Transact SQL, y promover el paralelismo de los grupos de trabajo. El único inconveniente real, no teórico, se presentaría con tablas con muchas columnas. Un procedimiento almacenado de modificación o inserción tendría demasiados parámetros, y sería agobiante llamarlo desde Delphi. Claro, también podríamos establecer convenios con los nombres de parámetros y escribir algún método genérico en Delphi para el traspaso de parámetros, que aprovecharse la correspondencia que establezcamos entre el nombre del parámetro y el nombre de la columna.

## Actualizaciones maestro/detalles

¿Y es que hay algún problema especial con este tipo de actualizaciones que las menciono en una sección aparte? Si se trata de relaciones maestro/detalles en las que las claves se asignan manualmente por el usuario, no hay nada que temer. Pero cuando las tablas en juego utilizan claves de asignación automática, necesitamos conocer un par de tretas y tener a mano un par de aspirinas.

Veamos primero la teoría. Para complicar un poco las cosas, supongamos que hay un proveedor enlazado a una relación maestro/detalles en tres niveles. Podría tratarse de clientes, pedidos y líneas de detalles. En el siguiente diagrama, el cuadrado 1 corres-

pondería a un cliente; 2 y 5 serían dos de sus pedidos; 3, 4 y 5 serían las líneas del pedido 2, y los otros tres cuadrados serían las líneas del pedido 6:



Con esta configuración podríamos insertar todos esos registros en una misma operación, o eliminarlos de golpe, o actualizar todos o una parte de ellos. El orden que se seguiría es el siguiente:

Operación	Orden de recorrido
Insertión	1-2-3-4-5-6-7-8-9
Borrado y modificación	3-4-5-2-7-8-9-6-1

Cuando menciono el orden de eliminación, quiero decir realmente el orden que se seguiría si todos los registros se eliminan manualmente. En realidad, el proveedor tiene dos opciones que nos pueden ser de utilidad: *poCascadeDeletes* y *poCascadeUpdates*. Estas opciones se pueden activar cuando el servidor SQL soporta borrados o actualizaciones en cascada para *todas* las relaciones de integridad referencial que participan en el paquete de datos. No obstante, no es necesario activarlas, son totalmente opcionales. ¿Cuál es la diferencia entonces? Si *poCascadeDeletes*, por ejemplo, no está activa, entonces no podemos borrar un cliente en el *TClientDataSet* si tiene pedidos asociados. Tendríamos que eliminar primero todos los detalles, y luego todos los pedidos para poder entonces borrar el registro del cliente. Es cierto que podemos programar el borrado automáticamente utilizando eventos del conjunto de datos cliente. Pero si activamos la opción mencionada, podríamos eliminar directamente un cliente aunque tenga detalles a su cargo; en tal caso, los registros de detalles desaparecen por arte de magia. Ahora bien, esto significa que el proveedor solamente emitirá una instrucción **delete** para el registro maestro. Si la relación de integridad referencial no borra automáticamente los detalles, se violará la restricción y obtendremos una bonita excepción.

Este es el prototipo del evento *BeforeUpdateRecord* de un proveedor:

```

procedure TDataModule1.DataSetProvider1.BeforeUpdateRecord (
    Sender: TObject; SourceDS: TDataSet;
    DeltaDS: TCustomClientDataSet; UpdateKind: TUpdateKind;
    var Applied: Boolean);
begin
    // ...
end;

```

El parámetro *SourceDS* empieza a tener sentido cuando se trata de actualizaciones sobre relaciones maestro/detalles. Cada vez que se dispara, apunta al conjunto de

datos de donde procede el registro que se va a actualizar. Lo mismo sucede con su hermano gemelo, *AfterUpdateRecord*.

Imagine que las dos siguientes consultas están conectadas entre sí, como maestro y detalles:

```
select *
from   CLIENTES

select *
from   DIRECCIONES
where  IDCliente = :IDCliente
```

Ahora póngase mentalmente en el lado cliente de la aplicación, donde se utiliza un par de *TClientDataSets* para editar el contenido de las consultas. Cuando inserte un registro en el conjunto de datos de las direcciones, el campo *IDCliente* será inicializado automáticamente con el valor del campo con el mismo nombre en el registro activo de clientes.

## Alta de pedidos

Utilizaremos la famosa *mastsql.gdb* para mostrar cómo organizar la grabación de pedidos junto con sus líneas de detalles. Por desgracia, esta base de datos está muy mal diseñada, por lo que me concentraré en los detalles técnicos más bien que en los funcionales. Como hemos hecho hasta el momento, agruparé todos los componentes en un mismo módulo de datos, para evitar el uso de servidores de capa intermedia.

Una vez que haya configurado la conexión a la base de datos, añada una primera consulta DB Express, llámela *qrPedidos*, en plural, porque nos permitirá ver todos los pedidos existentes. La instrucción SQL debe ser:

```
select OrderNo, SaleDate, Company,
       LastName || ', ' || FirstName Employee,
       PaymentMethod, ItemsTotal
from   ORDERS od join CUSTOMER cu on od.CustNo = cu.CustNo
       left join EMPLOYEE em on od.EmpNo = em.EmpNo
order by SaleDate asc
```

No tengo intención de actualizar directamente esta consulta. Traiga, por lo tanto, un *TDataSetProvider*, cambie su nombre a *prPedidos*, enlázelo a *qrPedidos* y active dos de sus opciones: *poIncFieldProps* y *poReadOnly*. Finalmente, deje caer sobre el módulo un *TClientDataSet* y enlázelo al proveedor recién creado.

Ahora pondremos un par de consultas para traer *un solo registro* de la tabla de pedidos y todos sus detalles. He llamado *qrPedido*, en singular, al primer componente, y *qrDetalles* al segundo. Como debe imaginar, hay que añadir un *TDataSource* y enlazarlo a *qrPedido* por medio de su propiedad *DataSet*, para luego modificar la propiedad *DataSource* de *qrDetalles* de modo que apunte al nuevo componente. Estas son las instrucciones SQL asociadas a cada consulta:

```

select *
from ORDERS
where OrderNo = :OrderNo

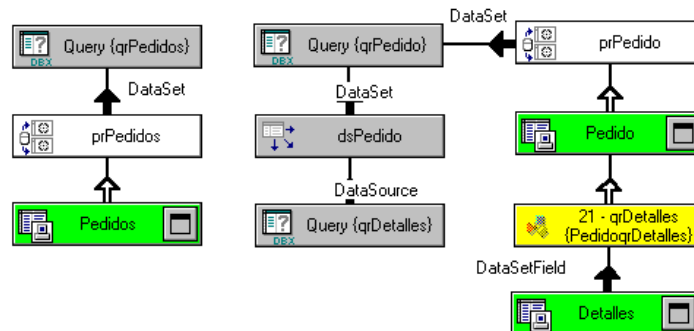
select *
from ITEMS
where OrderNo = :OrderNo
order by ItemNo asc

```

Este será el aspecto final del módulo de datos:



Observe que he añadido un *TSQLTable*, llamado *tbNextOrd*, y lo he asociado a la tabla *NEXTKEY*, que utilizaremos para asignar los números de pedidos. Y, naturalmente, he traído también los conjuntos de datos clientes *Pedido* y *Detalles*, donde se va a realizar la edición de los pedidos individuales. El siguiente diagrama muestra con más detalles las relaciones entre componentes.



Es el momento de configurar los campos de las consultas a actualizar, y de establecer los eventos de ayuda a la actualización. En primer lugar, tenemos que indicar cuáles campos forman parte de las claves primarias, para añadir *pInKey* en sus *ProviderFlags*:

- 1 En *qrPedido*, el campo *qrPedidoOrderNo*. Para este campo, asigne el valor `-1` en su propiedad *DefaultExpression*. La tabla *NEXTORD* proporcionará los números de pedidos automáticos.

- 2 En *qrDetalles*, los dos campos *qrDetallesOrderNo* y *qrDetallesItemNo*. El valor del primer campo hace referencia a un pedido existente, pero el valor de *ItemNo* debe asignarse automáticamente, aunque da lo mismo los valores que utilicemos, siempre que se garantice la unicidad.

Para pedir un nuevo número de pedido, añadimos estas declaraciones en la sección protegida de la clase del módulo de datos:

```
protected
    FUltimoPedido: Integer;
procedure NumeroPedido;
```

Y la implementación de *NumeroPedido* debe ser como la siguiente:

```
procedure TmodDatos.NumeroPedido;
begin
    Database.ExecuteDirect('update NEXTORD set NewKey = NewKey + 1');
    tbNextOrd.Open;
    try
        FUltimoPedido := tbNextOrdNEWKEY.AsInteger - 1;
    finally
        tbNextOrd.Close;
    end;
end;
```

Por último, debemos interceptar el disparo de *BeforeUpdateRecord*:

```
procedure TmodDatos.prPedidoBeforeUpdateRecord(Sender: TObject;
    SourceDS: TDataSet; DeltaDS: TCustomClientDataSet;
    UpdateKind: TUpdateKind; var Applied: Boolean);
var
    F: TField;
begin
    if UpdateKind = ukInsert then
        begin
            F := DeltaDS.FieldByName('ORDERNO');
            if SourceDS = qrPedido then
                begin
                    NumeroPedido;
                    F.NewValue := FUltimoPedido;
                end
            else // qrDetalles
                if F.NewValue <= 0 then
                    F.NewValue := FUltimoPedido;
            end;
        end;
```

Es aquí donde se encuentra la parte más importante para la actualización de relaciones maestro/detalles. Nos detendremos ahora, aunque el ejercicio está sin terminar, para explicar el funcionamiento de la técnica.

## Mantenimiento de la integridad referencial

En el método que acabamos de ver, se mezclan dos técnicas diferentes. Ya hemos hablado sobre la primera: la asignación automática de valores incrementales en una columna. Esto es lo que sucede en la tabla *ORDERS*, y el fragmento de código que le corresponde es:

```

if UpdateKind = ukInsert then
begin
  F := DeltaDS.FieldName('ORDERNO');
  if SourceDS = qrPedido then
  begin
    NumeroPedido;
    F.NewValue := FUltimoPedido;
  end
  // ...
end;

```

Recuerde que la opción *poPropagateChanges* del proveedor garantiza que el valor asignado a *NewValue* se propagará al conjunto de datos cliente durante la conciliación. No hay mayor misterio ahora.

Las novedades están en la actualización de las filas de detalles. No porque haya que asignarle valores secuenciales a *ItemNo*, pues lo resolveremos en el conjunto de datos clientes, con la ayuda de un campo de tipo agregado configurado de este modo:

Propiedad	Valor
<i>FieldName</i>	<i>Maximo</i>
<i>Expression</i>	<i>MAX(ITEMNO)</i>
<i>Active</i>	<i>True</i>

Este campo calcula automáticamente el valor máximo de la columna *ItemNo* de los detalles del pedido activo. Si el pedido no tiene detalles, el valor del campo estadístico *Maximo* será nulo. Para inicializar una nueva fila de detalles se utiliza el evento *OnNewRecord* del conjunto de datos cliente *Detalles*:

```

procedure TmodDatos.DetallesNewRecord(DataSet: TDataSet);
begin
  if VarIsNull(DetallesMaximo.Value) then
    DetallesITEMNO.Value := 1
  else
    DetallesITEMNO.Value := DetallesMaximo.Value + 1;
  DetallesQTY.Value := 1;
  DetallesDISCOUNT.Value := 0;
end;

```

Aunque no lo hacemos explícitamente, el valor de *OrderNo* en la fila de detalles se copia desde la columna correspondiente del registro de cabecera del pedido. ¿Recuerda que hemos asignado un -1 en la propiedad *DefaultExpression* de ese campo? Eso significa que, cuando creamos una nueva línea de detalles, su columna *OrderNo*

recibe un  $-1$ , si la cabecera de pedido es nueva también, o un valor positivo, si se trata de un pedido ya grabado. A continuación se asigna el valor de *ItemNo*. Observe que hay que comprobar por separado si el valor del campo calculado *Maximo* es nulo; en otro caso, se asigna el máximo *más* uno, y así queda resuelta la asignación de valores automáticos a *ItemNo*.

Nos queda solamente el problema que se presenta al insertar una línea de detalles en un pedido que también estamos creando. Pero eso es lo que estamos verificando en *BeforeUpdateRecord*, en el proveedor:

```
if F.NewValue <= 0 then
  F.NewValue := F.UltimoPedido;
```

Es decir, que si el valor del campo *OrderNo* en una fila de detalles es negativo, quiere decir que un poco antes se ha disparado la inserción del registro de cabecera. Debemos entonces tener en *F.UltimoPedido* el valor asignado para su clave primaria, y lo copiamos en el *NewValue* del *OrderNo* de la nueva fila de detalles.

## Buscar, listar y editar

¿Qué tal si terminamos el ejemplo de altas de pedidos? Seleccionamos la ventana principal, y colocamos en ella una rejilla de datos sólo lectura, con los controles habituales de navegación, para mostrar los datos del componente *Pedidos*. Añadimos también un par de acciones para editar o insertar pedidos. El evento *OnUpdate* de la acción de edición debe tener esta respuesta:

```
procedure TwndPrincipal.EditarUpdate(Sender: TObject);
begin
  TAction(Sender).Enabled := not modDatos.Pedidos.IsEmpty;
end;
```

Y la respuesta a *OnExecute* debe ser:

```
procedure TwndPrincipal.EditarExecute(Sender: TObject);
begin
  with modDatos.Pedido do begin
    Params.ParamByName('ORDERNO').Assign(modDatos.Pedidos.ORDERNO);
    Open;
    try
      with TdlgPedido.Create(Self) do
        try
          Edit;
          if ShowModal = mrOk then
            modDatos.Pedidos.Refresh;
          finally
            Free;
          end;
        finally
          Close;
        end;
      end;
    end;
  end;
end;
```



Se trata de una situación muy frecuente: si hubiéramos utilizado la misma consulta con detalles para el listado de pedidos, tendríamos que mover demasiados datos desde el servidor. Por lo tanto, en el listado se utiliza una consulta más “ligera”. Cuando queremos editar un registro, sin embargo, modificamos el parámetro de la consulta compleja para traer solamente *ese* registro. Abrimos la consulta, ejecutamos en forma modal el diálogo *TdlgPedido*, y si lo cerramos con el botón *Aceptar*, releemos el contenido de *Pedidos*.

#### NOTA

Hubiera sido preferible releer solamente el registro activo de *Pedidos*. Pero recordemos que la consulta asociada es un encuentro natural, y no existe una forma sencilla de releer solamente una fila.

La inserción es similar:

```
procedure TwndPrincipal.InsertarExecute(Sender: TObject);
begin
    with modDatos.Pedido do begin
        Params.ParamByName('ORDERNO').Value := -1;
        Open;
        try
            with TdlgPedido.Create(Self) do
                try
                    Append;
                    if ShowModal = mrOk then
                        modDatos.Pedidos.Refresh;
                    finally
                        Free;
                    end;
                finally
                    Close;
                end;
            end;
        end;
    end;
end;
```

En este caso, asignamos un valor “imposible” al parámetro de la consulta, para obtener un conjunto de datos vacío, pero que contenga la información estructural de la relación implicada. Hay casos en que es preferible obtener ese conjunto de datos vacío llamando a un viejo conocido, el método *CreateDataSet*:

```
procedure TwndPrincipal.InsertarExecute(Sender: TAction);
begin
    modDatos.Pedido.CreateDataSet;
    modDatos.Pedido.Append;
    // ... etcétera ...
end;
```

Lamentablemente, la presencia del campo agregado en el conjunto de datos de detalles provoca un error al llamar a *CreateDataSet*. Un *bug* más.

El aspecto del formulario de entrada de datos es el que sigue:

Línea	Producto	Cantidad	Descuento
1	900	2	0.00%
2	1.313	1	0.00%

No he querido complicar el ejemplo, y estoy obligando a teclear los códigos de clientes, empleados y productos. De todos modos, me he portado bien y he preparado un *TLabel* con algunas sugerencias para cuando ejecute la aplicación.

Lo más importante en este formulario es la respuesta al evento *OnCloseQuery*, donde intentamos grabar las modificaciones pendientes:

```

procedure TdlgPedido.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
begin
  with modDatos.Pedido do
    if ModalResult <> mrOk then
      CancelUpdates
    else if ApplyUpdates(0) > 0 then
      Abort;
end;

```

Si lo desea, puede mejorar el método pidiendo una confirmación al usuario cuando se cancela la ejecución y se han realizado cambios.

## Errores durante la resolución

En todos los ejemplos anteriores, el tratamiento de los errores ha sido muy primitivo... aunque si quiere que le sea sincero, es más que suficiente para la mayoría de las aplicaciones. No obstante, DataSnap nos permite mucho más control y es justo que presentemos, además de la verdad, *casi* todo el resto de la verdad.

```

procedure TmodDatos.DataSetProvider1UpdateError(Sender: TObject;
  DataSet: TCustomClientDataSet; E: EUpdateError;
  UpdateKind: TUpdateKind; var Response: TResolverResponse);
begin
  // ...
end;

```

Este evento, sin embargo, se diseñó pensando en un tipo de error muy concreto: el que se produce como resultado de la violación de un bloqueo optimista.

## Conciliación

Cuando termina la resolución, el proveedor envía el *Delta* modificado de vuelta al conjunto de datos cliente, con indicaciones sobre los errores encontrados y, opcionalmente, los valores modificados durante la resolución. Se inicia entonces el algoritmo de conciliación, que tiene dos objetivos importantes:

- 1 Si se produjeron errores, hay que notificárselo al cliente. En tal caso, es posible que el cliente corrija algunos registros.
- 2 Si no hay errores, la conciliación debe mezclar el contenido del nuevo *Delta* dentro del contenido de *Data*.

El segundo objetivo se cumple sin ayuda alguna por parte nuestra. Sólo tenemos que ocuparnos de notificar los errores y, opcionalmente, de ofrecer algún mecanismo de corrección. Sin embargo, es poco realista pensar que el usuario tendrá la suficiente sangre fría para poder resolver los conflictos de actualización concurrente. A un programador (¡a mí mismo!) le cuesta a veces comprender esos problemas sin pararse a meditar, lápiz y papel en mano.

La respuesta que hemos dado hasta ahora a los errores en la conciliación es, y seguirá siendo en la mayoría de nuestros ejemplos, la siguiente:

```
procedure TmodDatos.EmpleadosReconcileError (
    DataSet: TCustomClientDataSet; E: EReconcileError;
    UpdateKind: TUpdateKind; var Action: TReconcileAction);
begin
    Application.ShowException(E);
end;
```

Quizás lo más extraño sea que, a pesar de que me paso el tiempo advirtiendo sobre lo pecaminoso que es “enmascarar” una excepción, en este evento me limite a mostrar un mensaje de error en la pantalla. A estas alturas, ya debe imaginar la respuesta: es posible que se produzcan varios errores durante la resolución. Si lanzáramos una excepción al recibir el primer error, no podríamos procesar el siguiente.

Por otra parte, tampoco podemos dejar que la operación que inicia la grabación se vaya de rositas, con las manos limpias, si se produce algún contratiempo. Recuerde el principio básico de Delphi: si hemos llegado a la instrucción *X* es porque no se ha producido antes ninguna excepción. Es por este motivo que nos hemos acostumbrado a iniciar la grabación con instrucciones como la siguiente:

```
if modDatos.Empleados.ApplyUpdates(0) > 0 then
    Abort;
```

Con el código anterior, ¿cómo vería el usuario final los errores, si nos decidiéramos a aumentar el parámetro *MaxErrors* de *ApplyUpdates* a un valor positivo? Si se produjeran varios errores, en la pantalla aparecerían sucesivamente varios cuadros de mensajes. Pero podemos mejorar un poco esa presentación, con un poco de ingenio.

Le propongo iniciar una aplicación que permita editar la tabla *CUSTOMER* sobre una rejilla. Como la tabla mencionada forma parte de una relación de integridad referencial con la tabla de pedidos, será muy sencillo provocar errores en la resolución; bastará que borremos un cliente, porque casi todos ellos han realizado compras.

En el ejemplo del CD he puesto todos los componentes de acceso a datos en un módulo de datos, y he añadido las siguientes variables dentro de la clase del módulo:

```
private
    FErrorMessage: string;
    FErrorCount: Integer;
    // ...
```

A continuación he modificado el evento *OnReconcileError* del conjunto de datos *Clientes* en la forma que aparece a continuación:

```
procedure TmodDatos.ClientesReconcileError(
    DataSet: TCustomClientDataSet; E: EReconcileError;
    UpdateKind: TUpdateKind; var Action: TReconcileAction);
begin
    Inc(FErrorCount);
    // VarToStr le obligará a añadir la unidad Variants
    FErrorMessage := Format('%s'#13'%d- %s (cliente %s)',
        [FErrorMessage, FErrorCount, E.Message,
        VarToStr(DataSet.FieldName('CUSTNO').OldValue)]);
end;
```

En vez de mostrar prematuramente el error, almaceno el mensaje en un acumulador, y lo complemento con la información sobre el código de cliente original. Simulo además una lista numerada, llevando la cuenta sobre cuántos mensajes hemos recibido hasta el momento.

Para inicializar las dos nuevas variables, y para mostrar el mensaje acumulado, he definido un método público en el módulo de datos, para encapsular la grabación de las modificaciones hechas a la tabla de clientes:

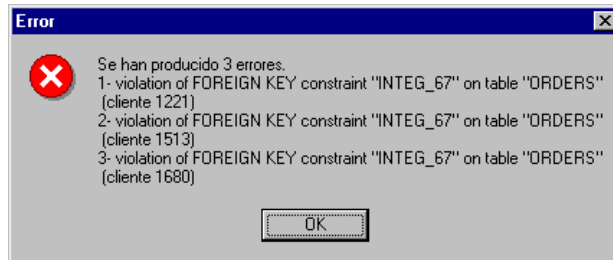
```
procedure TmodDatos.Guardar;
begin
    FErrorMessage := '';
    FErrorCount := 0;
    if Clientes.ApplyUpdates(-1) > 0 then
        begin
            FErrorMessage := Format(
                'Se han producido %d errores.%s',
                [FErrorCount, FErrorMessage]);
            MessageDlg(FErrorMessage, mtError, [mbOk], 0);
        end;
```

```

    Abort;
end;
end;

```

La siguiente imagen muestra un posible mensaje de error compuesto:



Para terminar con la conciliación, quiero que sepa que existe una plantilla de diálogo de conciliación, desde la lejana época de Delphi 3. Se encuentra en el Almacén de Objetos, en la página *Dialogs*, y el icono que la copia en el proyecto activo se llama *Reconcile Error Dialog*.

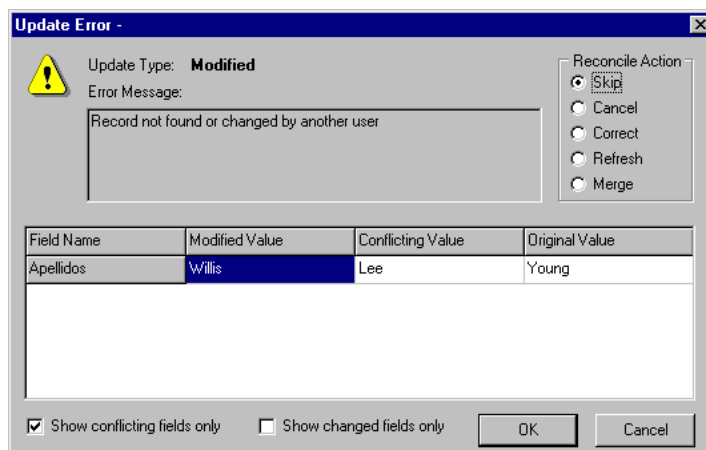
La persona que desarrolló la plantilla eliminó la variable global del formulario del código fuente de la unidad. Eso obliga a quitar el nuevo formulario de la lista de creación automática. La idea de este señor, o señora, era que para mostrar el diálogo tuviéramos que llamar a una función global que aparece declarada en la interfaz de la unidad:

```

procedure TmodDatos.EmpleadosReconcileError(
    DataSet: TCustomClientDataSet; E: EReconcileError;
    UpdateKind: TUpdateKind; var Action: TReconcileAction);
begin
    Action := HandleReconcileError(DataSet, UpdateKind, E);
end;

```

El aspecto del diálogo en tiempo de ejecución es el siguiente:



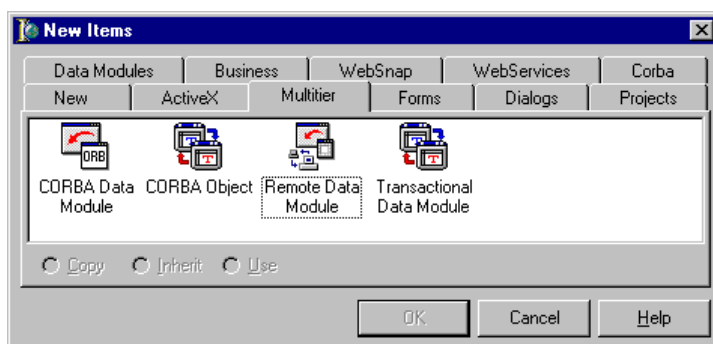
¿Qué acción de conciliación aplicaría usted en el caso que nuestro? Si puede responder en menos de 30 segundos, cierre el libro, baje al bar más cercano y bébase una jarra bien fría de cerveza, que se la ha ganado. Y dígame al de la barra que invito yo.

## Servidores de capa intermedia

EL GRAN MOMENTO HA LLEGADO: VAMOS A explicar cómo separar los componentes de acceso a SQL en un servidor de capa intermedia independiente. Los beneficios de esta separación son incontables. Podremos, por ejemplo, desarrollar aplicaciones que puedan acceder a sus datos a través de una conexión analógica a través de Internet. O incluso que decidan dinámicamente qué tipo de conexión deben utilizar en cada instante.

### Módulos remotos

Para poder utilizar un servidor remoto, necesitamos un objeto COM que resida en su interior. Las clases a las que pertenecen esos objetos se crean añadiendo *módulos de datos remotos* a una aplicación o DLL. El asistente necesario se encuentra dentro del Almacén de Objetos, en la página *Multitier*.



Lo más frecuente es que el proyecto donde residirá el módulo sea una aplicación ejecutable. En realidad, sería más conveniente que el proyecto generase un servicio de Windows NT/2K... pero el matrimonio entre los servicios creados en Delphi con la infraestructura de Borland para COM tiene sus más y sus menos. De todos modos, no es algo a descartar.

Ahora bien, en determinadas circunstancias se puede alojar el módulo remoto dentro de una DLL. Esto será posible cuando la conexión establecida por el cliente pase por alguno de los “sustitutos” de DCOM proporcionados por Borland: cuando la cone-

ción se haga mediante los componentes *TWebConnection* o *TSocketConnection*. En ambos casos, en el lado servidor habrá una aplicación o incluso otra DLL que será la encargada de cargar la DLL que contiene nuestro módulo remoto. Por otra parte, también puede crear su servidor remoto por medio de un módulo “transaccional”, listo para ser instalado en COM+ o en el entorno MTS. En este caso, es obligatorio que el proyecto activo sea una DLL.

Una vez que invocamos el asistente para módulos remotos, aparece esta ventana:



Tenemos, en primer lugar, que suministrar el nombre de la clase COM. Recuerde que el nombre que teclee se concatenará al final del nombre del proyecto para obtener el identificador de programa. Por lo tanto, mientras el nombre de su proyecto sea suficientemente identificativo, no se caerá el cielo sobre los galos si utiliza algún nombre tonto para la clase, como el de la imagen.

Si el módulo se aloja dentro de un ejecutable, es recomendable utilizar *Multiple Instance*, para que un mismo proceso sirva múltiples instancias de la clase COM. Así se ahorran recursos, pero si la clase COM está mal programada hay peligro de que un cliente pueda estropear la memoria global del proceso. En tal extraño caso, podría intentarlo inicialmente con *Single Instance*... pero poniendo entre los planes a corto plazo la migración al modelo más eficiente.

Al terminar de ejecutar el asistente, se añade una nueva unidad al proyecto activo, en la que se define la siguiente clase:

```
type
  TDatabase = class(TRemoteDataModule, IDatabase)
    // ...
  protected
    class procedure UpdateRegistry(Register: Boolean;
      const ClassID, ProgID: string); override;
    // ...
  end;
```

Como verá, hay una clase base *TRemoteDataModule* predefinida por Delphi, que implementa los métodos de lo que parece ser una nueva interfaz, *IDatabase*; más adelante veremos qué métodos contiene *IDatabase*. Hay también un solo método que se redefine, *UpdateRegistry*. Esta es su implementación:

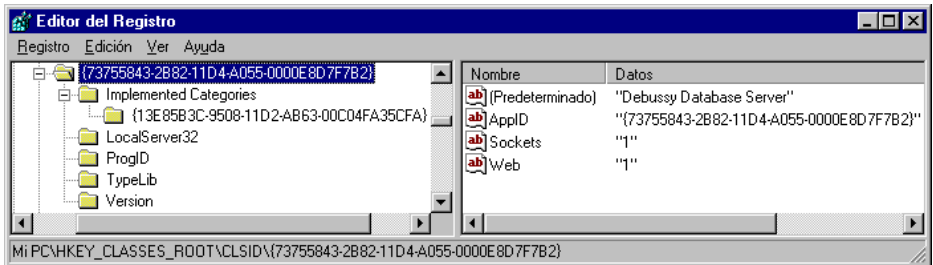


```

class procedure TDatabase.UpdateRegistry(Register: Boolean;
const ClassID, ProgID: string);
begin
  if Register then
  begin
    inherited UpdateRegistry(Register, ClassID, ProgID);
    EnableSocketTransport(ClassID);
    EnableWebTransport(ClassID);
  end
  else
  begin
    DisableSocketTransport(ClassID);
    DisableWebTransport(ClassID);
    inherited UpdateRegistry(Register, ClassID, ProgID);
  end;
end;
end;

```

Note, en primer lugar, que se trata de un método de clase *virtual*; no hace falta que exista una instancia del módulo para poder ejecutarlo. Este *UpdateRegistry* en particular, es introducido en *TComponent*. Pero todas las clases de Delphi que soportan una fábrica de clase tienen un método similar, que es ejecutado por la correspondiente fábrica de clases durante la instalación y desinstalación del servidor. La implementación de este método por parte de la clase *TRemoteDataModule* graba en el registro, además de las claves básicas que necesita cualquier servidor COM, una clave que identifica a esta clase como un servidor de capa intermedia para DataSnap. Observe en la siguiente imagen la clave que cuelga bajo *Implemented Categories*:



El GUID que menciono no es un valor generado para este servidor concreto, sino una constante definida por Borland para identificar la categoría *Borland DataSnap Application Servers*, también definida por los susodichos. La declaración se encuentra en la unidad *MidConst*:

```

const
  CATID_MIDASAppServer: TGUID =
    '{13E85B3C-9508-11D2-AB63-00C04FA35CFA}';

```

Gracias a esta categoría, Borland puede recorrer el registro para identificar los servidores de capa intermedia instalados en un ordenador.

Más adelante estudiaremos qué significan las llamadas a los métodos *EnableSocketTransport*, *EnableWebTransport* y sus dos hermanos gemelos.

## El modelo de concurrencia

Si no me ha hecho caso y ha elegido *Single Instance*, está muy claro el modelo de concurrencia que va a utilizar su servidor. Como cada objeto va a disponer de todo un proceso, y de su hilo principal, para sí, tendrá toda la simultaneidad que pueda desear. Sólo que no podrá servir demasiados clientes con un único ordenador, y que cada cliente tendrá que esperar a que arranque el proceso antes de poder echar a andar.

Por lo tanto, supongamos que el modelo de creación sea *Multiple Instance*. Si el modelo de concurrencia elegido es *Apartment*, será el propio servidor el que creará un hilo separado para cada objeto que cree, y cada hilo contará con su propia bomba de mensajes. Estas labores son implementadas por la fábrica de clases que el asistente asigna a la nueva clase COM: *TComponentFactory*.

En cambio, si elige *Free* como modelo de concurrencia, será el propio sistema operativo el encargado de administrar los hilos cada vez que llegue una petición a uno de los objetos creados. Normalmente, esto se hará eficientemente, almacenando los hilos disponibles en una caché del sistema. Recuerde que puede haber muchos clientes conectados simultáneamente, pero si las peticiones se responden en poco tiempo, puede que hagan falta muy pocos hilos para atender a esos clientes.

¿Entonces qué, apartamentos o el modelo libre? Mi experiencia me aconseja comenzar con el modelo de apartamento, porque siempre será mejor tener un hilo dedicado que tener que esperar a que se libere uno. Naturalmente, el que gana con este compromiso es el cliente, pero el servidor es quien lo sufre. Siempre que he utilizado servidores de capa intermedia en aplicaciones reales, se han dado las condiciones para que ningún servidor atienda más de veinte o treinta clientes, y no he encontrado problemas que no se resuelvan añadiendo memoria RAM. Cuando llegue al límite de las posibilidades del servidor, puede entonces probar el modelo *Free*.

¿Y qué hay de la sincronización interna dentro del servidor? Casi todas las explicaciones sobre los modelos de concurrencia, como sabrá, se obsesionan con la explicación de los peligros de hacer llamadas a un mismo objeto desde dos hilos clientes diferentes. Es cierto que si esto sucede, el modelo *Apartment* nos permite despreocuparnos, porque todas esas llamadas son atendidas secuencialmente gracias a la cola de mensajes dedicada. Entonces nos asustan advirtiéndonos sobre la necesidad, en el modelo *Free*, de utilizar secciones críticas incluso para los datos de instancia de cada objeto. Pero no se preocupe, porque el problema no se presentará si su aplicación cliente llama al servidor desde un solo hilo, y si no se comparte explícitamente, mecanismos de *pooling* aparte, un mismo objeto servidor desde dos clientes. En tres palabras: puede dormir tranquilo.

## La interfaz *IAppServer*

Cada vez que creamos un módulo remoto, Delphi define un nuevo tipo de interfaz; si el proyecto no tiene una biblioteca de tipos, crea una nueva y se la asocia. La interfaz descende siempre de *LAppServer*, cuya declaración es:

```

type
  IAppServer = interface(IDispatch)
    ['{1AEFCC20-7A24-11D2-98B0-C69BEB4B5B6D}']
    function AS_ApplyUpdates(const ProviderName: WideString;
      Delta: OleVariant; MaxErrors: Integer;
      out ErrorCount: Integer;
      var OwnerData: OleVariant): OleVariant; safecall;
    function AS_GetRecords(const ProviderName: WideString;
      Count: Integer; out RecsOut: Integer; Options: Integer;
      const CommandText: WideString; var Params: OleVariant;
      var OwnerData: OleVariant): OleVariant; safecall;
    function AS_DataRequest(const ProviderName: WideString;
      Data: OleVariant): OleVariant; safecall;
    function AS_GetProviderNames: OleVariant; safecall;
    function AS_GetParams(const ProviderName: WideString;
      var OwnerData: OleVariant): OleVariant; safecall;
    function AS_RowRequest(const ProviderName: WideString;
      Row: OleVariant; RequestType: Integer;
      var OwnerData: OleVariant): OleVariant; safecall;
    procedure AS_Execute(const ProviderName: WideString;
      const CommandText: WideString; var Params: OleVariant;
      var OwnerData: OleVariant); safecall;
  end;

```

La interfaz derivada de *LAppServer* es la que utilizará el cliente posteriormente para traer datos desde el servidor de capa intermedia, y para enviar las modificaciones durante la resolución. Habrá notado que, excepto *AS\_GetProviderNames*, todos restantes métodos de la interfaz tienen como primer parámetro el nombre de un proveedor. La mayoría de ellos se implementan, dentro de *TRemoteDataModule*, localizando un componente proveedor y delegando en él la ejecución. Esta es, por ejemplo, la implementación que recibe *AS\_ApplyUpdates*:

```

function TRemoteDataModule.AS_ApplyUpdates(
  const ProviderName: WideString; Delta: OleVariant;
  MaxErrors: Integer; out ErrorCount: Integer;
  var OwnerData: OleVariant): OleVariant;
begin
  Lock;
  try
    Result := Providers[ProviderName].ApplyUpdates(
      Delta, MaxErrors, ErrorCount, OwnerData);
  finally
    UnLock;
  end;
end;

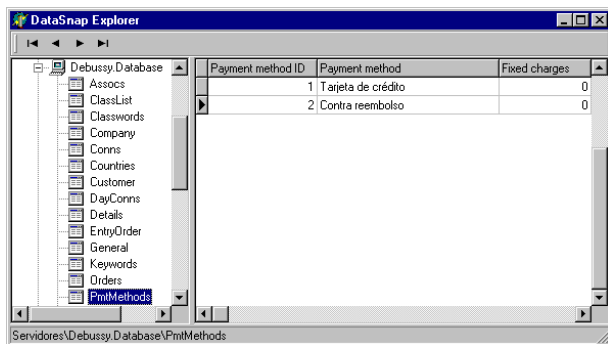
```

Esto nos interesa mucho. Ya he explicado antes algunos trucos que se logran pidiendo directamente registros al proveedor. Si necesitamos pedir algo a un proveedor que se encuentre en un módulo remoto, es bueno saber que podemos utilizar la in-

terfaz *IAppServer* para acceder indirectamente a la mayoría de los métodos interesantes del proveedor. Por ejemplo, la siguiente función recibe el identificador de clase de un servidor de capa intermedia, crea una instancia local de la clase, y utiliza el método *AS\_GetProvidersNames* para recuperar la lista de nombres de proveedores soportados por el servidor:

```
procedure GetProviders(AClassID: TGUID; ProvList: TStrings);
var
  I: Integer;
  AppServer: IAppServer;
  Providers: Variant;
begin
  ProvList.BeginUpdate;
  try
    ProvList.Clear;
    AppServer := CreateComObject(AClassID) as IAppServer;
    Providers := AppServer.AS_GetProviderNames;
    if VarIsArray(Providers) then
      for I := VarArrayLowBound(Providers, 1) to
        VarArrayHighBound(Providers, 1) do
        ProvList.Add(Providers[I]);
    finally
      ProvList.EndUpdate;
    end;
  end;
```

La aplicación *DataSnap Explorer*, del CD, muestra cómo recuperar los nombres de los servidores de DataSnap instalados localmente, y los proveedores asociados:



Como puede ver, incluso puede traer los primeros 25 registros devueltos por el proveedor, para mostrarlos en una rejilla.

## Exportación de proveedores

Ya tenemos un módulo de datos remoto dentro de un servidor COM, ¿qué hacemos ahora? Nuestra tarea será traer componentes *TDataSetProvider* al módulo, uno por cada consulta que queramos “publicar”... y por supuesto, traer también esas consultas. Exactamente igual que hemos hecho en los capítulos anteriores. Normalmente, la propiedad *Exported* de los proveedores vale *True*. si la cambiamos a *False*, el provee-

dor no será visible fuera del módulo. Las aplicaciones clientes trabajarán indirectamente con los proveedores exportados a través de la interfaz *LAppServer* del módulo que los contiene.

### UN POCO DE HISTORIA

Antes de la aparición de Delphi 5, la interfaz remota utilizada por Midas se llamaba *IDataBroker*. Para “exportar” un proveedor había que crear una propiedad de tipo *IProvider* en la interfaz derivada de *IDataBroker*, se trataba de un segundo tipo de interfaz remota, que era la que realmente usaban los clientes para recuperar datos y enviar actualizaciones. Por fortuna, la creación e implementación de la interfaz se realizaban por medio de un comando de menú, en el Entorno de Desarrollo. Para obtener un *IProvider* y comenzar a trabajar, en consecuencia, el cliente debía primero pedirlo al puntero *IDataBroker* que obtenía mediante una conexión remota, y sólo después tenía acceso a sus datos. Resultado: una llamada adicional. Además, COM se veía obligado a crear un *proxy* para cada proveedor en el lado cliente, y eso aumentaba el consumo de recursos. Por estas razones, la aparición de *LAppServer* en Delphi 5 fue un adelanto notable.

Voy a ahora a plantear dos preguntas diferentes, porque la respuesta será la misma. ¿Cómo obtiene el módulo remoto los nombres de proveedores exportables? Cuando se ejecuta un método de *LAppServer* que menciona un nombre de proveedor, ¿cómo localiza el módulo remoto el componente deseado? Aunque parezca lo más natural, el módulo no recurre a su propiedad *Components*, que agrupa en una lista todos los componentes de los que el módulo es propietario. Eso sería ineficiente; tenga en cuenta que por cada proveedor debe haber al menos un conjunto de datos. Si el proveedor se asocia a una relación maestro/detalles, aumenta el número de componentes que no son proveedores. Y no tiene mucho sentido buscar en una lista de 100 elementos de la que sólo 30 son candidatos aceptables.

En consecuencia, *TRemoteDataModule* implementa internamente una lista de proveedores. Y preste atención, porque esto es importante: para el mantenimiento de la lista, la clase del módulo ofrece dos métodos públicos.

```
procedure TRemoteDataModule.RegisterProvider(
    Value: TCustomProvider);
procedure TRemoteDataModule.UnRegisterProvider(
    Value: TCustomProvider);
```

Cuando se crea un proveedor, desde el constructor del componente se comprueba si su propietario es un módulo remoto para, en caso afirmativo, ejecutar su método *RegisterProvider*. Durante la destrucción del proveedor, se llama a *UnRegisterProvider*.

¿Por qué es interesante conocer la existencia de estos métodos? Un problema que encontrará cuando desarrolle servidores de capa intermedia *reales*, es que el espacio físico, dentro del módulo, para colocar los componentes de acceso a datos y los proveedores se acaba con relativa rapidez, y hay que utilizar otros módulos para poder seguir añadiendo proveedores. Podemos hacer que estos módulos de desbordamiento sean también módulos remotos, pero eso requeriría que cada cliente estableciese dos conexiones por servidor, todo un desperdicio. Veremos, más adelante, que

Delphi 6 ofrece una solución, por medio del componente *TSharedConnection*. Pero es más sencillo utilizar *RegisterProvider* para registrar los proveedores de los módulos secundarios como si pertenecieran al módulo remoto principal. Es cierto también que con esta técnica aumentamos el tamaño de la lista de proveedores, y si abusamos, se ralentizará la localización de los proveedores.

## Tipos de conexiones

Volvemos al lado del cliente. Suponemos que ya tenemos un servidor de capa intermedia instalado y disponible en algún punto de alguna red. Para simplificar, por supuesto, comenzaremos con el caso más sencillo: el servidor reside en el mismo ordenador que el cliente. Sea como sea, necesitamos un componente que realice la conexión con el servidor, y los encontraremos en la página *DataSnap*:



En comparación con Delphi 5, falta el componente *TOLEnterpriseConnection*, que ya estaba oculto en la Paleta de esa versión. Ahora ha desaparecido definitivamente. He aquí un pequeño resumen sobre el uso de cada uno de los componentes *DataSnap*:

Componente	Propósito
<i>TDCOMConnection</i>	Conexiones locales o remotas que usan DCOM directamente
<i>TSocketConnection</i>	Conexión simulada por medio de TCP/IP
<i>TSimpleObjectBroker</i>	Permite elegir aleatoriamente un ordenador como servidor
<i>TWebConnection</i>	Conexión simulada por HTTP, a través de Internet
<i>TConnectionBroker</i>	Permite cambiar el tipo de conexión en tiempo de ejecución
<i>TSharedConnection</i>	Permite acceder a módulos “hijos” remotos
<i>TLocalConnection</i>	Simula un servidor remoto dentro de la propia aplicación
<i>TCorbaConnection</i>	Conexión a un módulo CORBA remoto

Es también posible que en algún momento necesite un componente, *TSoapConnection*, que se encuentra en la página *WebServices*:

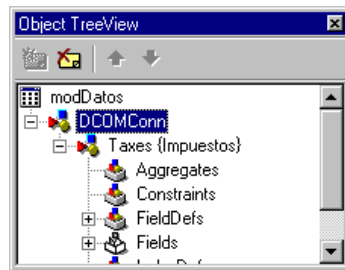


Este último componente establece la conexión con un servidor remoto implementado como un servicio de Internet, usando el protocolo SOAP.

## Conexiones DCOM

No hay misterio alguno en el uso de *TDCOMConnection*. Lo primero es determinar en qué ordenador se encuentra el servidor, para asignar su nombre en la propiedad *ComputerName*. A continuación, hay que asignar algo en *ServerGUID* o en *ServerName*. Y es aquí donde podemos encontrar alguna dificultad, porque DCOM no nos echará una mano para listar los servidores de capa intermedia registrados en un ordenador remoto.

¿La solución? Registre la biblioteca de tipos del servidor en el ordenador donde está programando. Si realmente va a utilizar DCOM para las conexiones remotas y quiere añadir métodos a la interfaz remota, estará obligado de todos modos a registrar la biblioteca de tipos para que el sistema operativo pueda encargarse automáticamente del *marshaling*. Recuerde que registrar la biblioteca no es lo mismo que registrar el servidor completo. La biblioteca reside en un fichero de extensión *tlb*. Puede recurrir al programa *regsvr.exe* de Borland, o incluso podría incluir el fichero binario como un recurso dentro de una DLL vacía, y registrar la DLL sin más.



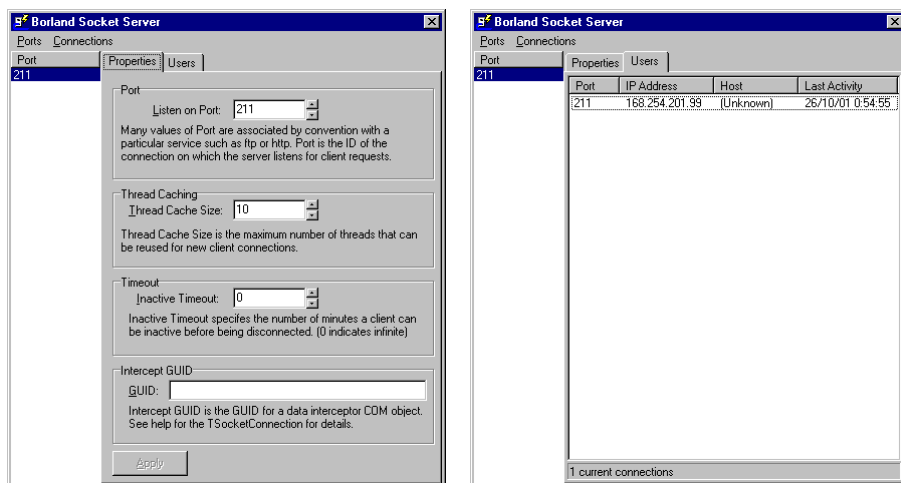
El resto es extremadamente simple. Traiga un *TClientDataSet* y modifique su propiedad *RemoteServer* para que apunte a *DCOMConnection1*. A continuación, despliegue la lista asociada a su propiedad *ProviderNames*. Verá entonces todos los nombres de proveedores exportados desde el servidor. A partir de ese momento podrá trabajar como he explicado en los capítulos anteriores: podrá crear componentes de acceso a campos, recuperar la lista de parámetros del conjunto de datos, añadir verificaciones a los campos o al conjunto de datos, abrir el conjunto de datos y navegar sobre él, y si ha realizado cambios, llamar a *ApplyUpdates* para iniciar el algoritmo de resolución.

## Marshaling a través de zócalos

¿Cuántas redes conoce usted que estén bien configuradas? ¡La suya, por supuesto! Bueno, y la mía, porque está hablando conmigo, ¿no? Bromas aparte, es muy difícil encontrar una red de mediano tamaño que satisfaga los requerimientos de DCOM. Casi nunca hay un dominio donde puedan validarse los usuarios y sus contraseñas. O, siendo optimistas sobre las habilidades del administrador, tropezamos con algún cortafuegos que entorpece el acceso remoto de los usuarios. He leído artículos, en revistas de informática supuestamente serias, que recomiendan a los administradores

de red que detengan el servicio RPC para cortar el paso a los intrusos. Suponer que un cliente remoto podrá acceder al servidor de capa intermedia usando sólo DCOM es ser muy ingenuos.

Borland tuvo una idea inteligente: sustituir la capa de comunicación entre ordenadores con alguna simulación de COM. Las siguientes imágenes pertenecen a una aplicación, *scktsrvr.exe*, que encontrará en el directorio *bin* de Delphi:

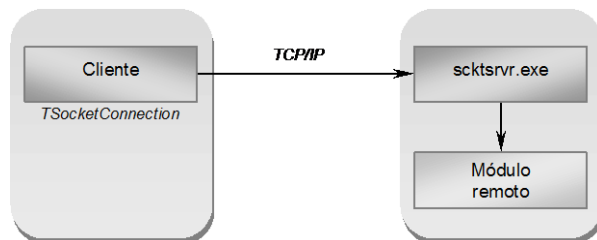


La aplicación mencionada debe ejecutarse en el ordenador donde está registrado el servidor de capa intermedia, y debe estar ejecutándose en el momento en que un cliente solicite una conexión al servidor. Por este motivo, aunque podemos ejecutar el *Socket Server* como una aplicación “más”, es preferible instalarla como un servicio de Win32. Para registrarla como servicio, nos bastaría ejecutar el siguiente comando:

```
scktsrvr.exe /install /silent
```

Muestro la opción *silent* para que conozca cómo “silenciar” el mensaje de instalación exitosa si desea instalar el servicio junto con su aplicación. Si necesita desinstalar el servicio, debe utilizar el parámetro *uninstall*.

El siguiente diagrama representa el funcionamiento de una conexión por zócalos:





Las peticiones que envía el cliente a través del componente *TSocketConnection* tienen la misma estructura que las peticiones COM que se envían a través de *TDCOMConnection...* pero físicamente consisten en escrituras sobre un zócalo, o *socket*. Al otro extremo del zócalo se encuentra *sktsrvr.exe*; el servicio TCP/IP implementado por esta aplicación recibe las peticiones y actúa como un *stub* de COM, instanciando los módulos remotos en forma local, ejecutando los métodos solicitados por el cliente, y finalmente empaquetando los resultados, de vuelta al cliente. Por este motivo es posible programar el servidor remoto como una DLL. Es más, si *sktsrvr* está ejecutándose como un servicio Win32, no hará falta que alguien inicie una sesión en el servidor para poder utilizar el módulo remoto. El efecto es similar a desarrollar el módulo COM dentro de una aplicación de servicio.

## Configuración de *TSocketConnection*

La configuración, en el lado cliente, de una conexión por zócalos es muy parecida a la de *TDCOMConnection*, aunque existen diferencias lógicas. En primer lugar, hay dos formas alternativas de señalar el ordenador que actuará como servidor:

```
property TSocketConnection.Address: string;  
property TSocketConnection.Host: string;
```

En *Address* podemos escribir la dirección IP del servidor, mientras que *Host* admite un nombre simbólico, que debe ser traducido entonces a una dirección; previsiblemente, ambas propiedades son excluyentes. A diferencia de lo que sucede con las conexiones DCOM, no se deben dejar vacías las dos, y si queremos señalar al ordenador local debemos utilizar el propio nombre del ordenador, el ya conocido *localhost* (mi opción preferida) o la dirección del bucle local: *127.0.0.1*. Muy importante: el cliente debe tener instalado WinSock2. Esta versión del software de sistema para los zócalos se introdujo en Windows 98.

El siguiente paso es indicar el puerto que está siendo atendido por el servidor. Borland ha escogido que por omisión se utilice el puerto 211. Pero la elección final depende del administrador de la red, y de las posibilidades del cortafuegos. Si decide usar otro puerto, debe primero realizar el cambio en el servidor, mediante su cuadro de diálogo de propiedades. El puerto utilizado por *sktsrvr* se almacena en el registro del ordenador remoto, en la siguiente clave:

```
[HKEY_LOCAL_MACHINE\Software\Borland\Socket Server\211]  
"Port"="211"
```

Tenga cuidado, porque si cambia el puerto, cambia también el nombre de la última clave, además del valor asociado a *Port*.

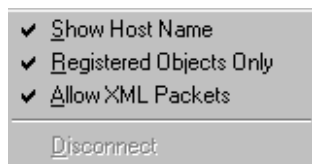
Y ahora viene una sorpresa: como regalo, *sktsrvr* puede escudriñar el registro del ordenador donde se ejecuta, para ofrecernos la lista de servidores DataSnap allí registrados. Recuerde que DCOM no era tan generoso con nosotros. La lista aparecerá cuando despleguemos el editor de la propiedad *ServerName*. Así evitamos registrar la

biblioteca de tipos del servidor en el cliente, o tener que apuntar en un trozo de papel el identificador de clase del servidor. Tanta potencia puede, sin embargo, convertirse en un arma de doble filo. Si un intruso quiere acceder a un servidor de capa intermedia registrado en un ordenador con acceso a Internet, tiene primero que adivinar su identificador de clase. ¿Probabilidades?, ¡cero! En cambio, si *socktsrvr* está activo, al *hacker* le bastaría conocer la IP del servidor, y obtendría los CLSIDs de todos los módulos remotos registrados por DataSnap.

¿Qué se puede hacer en tal caso? En primer lugar, incluir algún sistema de validación de clientes. Puede consistir en conceder el acceso al servicio 211 solamente a ciertas direcciones IP. O puede consistir en un sistema de identificación con claves y contraseñas, implementados por el servidor, o incluso por otra capa de software. Otra cosa sería impedir que determinados servidores se puedan instanciar a través de TCP/IP. ¿Recuerda la implementación del método *UpdateRegistry*, en el módulo remoto? Repito el código:

```
class procedure TDatabase.UpdateRegistry(Register: Boolean;
  const ClassID, ProgID: string);
begin
  if Register then begin
    inherited UpdateRegistry(Register, ClassID, ProgID);
    EnableSocketTransport(ClassID);
    EnableWebTransport(ClassID);
  end
  else begin
    DisableSocketTransport(ClassID);
    DisableWebTransport(ClassID);
    inherited UpdateRegistry(Register, ClassID, ProgID);
  end;
end;
```

Si quitamos la llamada a *EnableSocketTransport*, o la sustituimos por *DisableSocketTransport*, el servidor deja un aviso en el registro de Windows para que *socktsrvr* no intente su activación. No obstante, es necesario también activar la opción *Registered Objects Only* en el menú del servidor de zócalos.



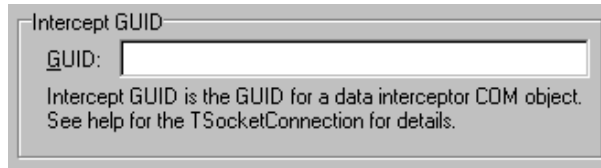
Para terminar, hay otra propiedad interesante en *TSocketConnection*. Se llama *SupportCallbacks*, es de tipo lógico, y se encuentra activa por omisión. En ese estado, el componente puede recibir eventos enviados desde el servidor COM. Las primeras versiones de *socktsrvr* no permitían el uso de eventos, y sólo fue posible su implementación a partir de la aparición de WinSock 2.

## Interceptores

La característica más potente de las conexiones por zócalos es que podemos crear *interceptores de paquetes*. Se trata de objetos COM que filtran todos los paquetes de datos que se envían entre sí *TSocketConnection* y el servicio implementado por *socksrrvr*. Hay que instalar el servidor que hará de interceptor y registrarlo, tanto en el servidor como en el cliente, y advertir a los dos extremos que deben utilizar la misma clase. En el componente *TSocketConnection* hay que modificar una de estas dos propiedades:

```
property TStreamedConnection.InterceptGUID: string;  
property TStreamedConnection.InterceptName: string;
```

En el extremo servidor hay que teclear el GUID (¡sí, teclearlo!) en el diálogo de propiedades de *socksrrvr*. También se puede realizar la asociación directamente, en el Registro de Windows.



Las clases de interceptores deben implementar la interfaz *IDataIntercept*. Esta es su declaración, extraída de la unidad *SConnect*:

```
type  
  IDataIntercept = interface  
    ['{B249776B-E429-11D1-AAA4-00C04FA35CFA}']  
    procedure DataIn(const Data: IDataBlock); stdcall;  
    procedure DataOut(const Data: IDataBlock); stdcall;  
end;
```

En el método *DataOut* recibimos un bloque de datos con el formato original que utiliza *DataSnap* para la comunicación remota. Debemos entonces realizar la transformación de esos datos. Podríamos cifrar el envío, por ejemplo. Si un *hacker* logra interceptar nuestro envío, no estaría en condiciones de entender su contenido. O podríamos comprimir los datos, para ganar velocidad; la compresión es también una forma de cifrado, si el intruso no conoce el algoritmo, claro. El método *DataIn* es el responsable de restaurar el bloque de datos a su formato original.

En ambos métodos, el acceso a los datos se realiza a través de la interfaz *IDataBlock*; así no nos atamos a una representación física de su estructura. En esta declaración he eliminado los métodos de implementación de las propiedades, para simplificarla:

```
type  
  IDataBlock = interface (IUnknown)  
    ['{CA6564C2-4683-11D1-88D4-00A0248E5091}']  
    procedure Clear; stdcall;  
    function Write(const Buf; Count: Integer): Integer; stdcall;
```

```

function Read(var Buf; Count: Integer): Integer; stdcall;
procedure IgnoreStream; stdcall;
function InitData(Data: Pointer; DataLen: Integer;
    CheckLen: Boolean): Integer; stdcall;
property BytesReserved: Integer;
property Memory: Pointer;
property Signature: Integer;
property Size: Integer;
property Stream: TStream;
end;

```

Delphi incluye entre sus ejemplos un proyecto que genera un servidor de filtrado de paquetes, *intrept.dpr*, dentro del directorio *Midas*. En el ejemplo se utiliza la unidad *ZLib*, que también se distribuye con Delphi, para comprimir paquetes. Si quiere probar fuerzas, puede crear su propio interceptor para registrar en un fichero el tamaño de cada envío de datos; es una herramienta muy útil, porque le permitirá medir cuán bien o mal se está comportando su servidor de capa intermedia.

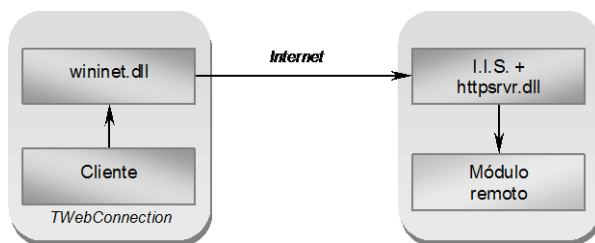
Es muy importante que se utilice la clase *TPacketInterceptFactory* para crear la fábrica de clases. Esta clase redefine el método *UpdateRegistry* para indicar en el registro que la clase implementa la categoría de interceptores de DataSnap. Gracias a esta marca, el editor de la propiedad *InterceptName* puede localizar todas las clases registradas que pueden desempeñar esta función.

#### ADVERTENCIA

La interfaz *IDataIntercept* se ha definido internamente, dentro de una unidad Delphi, y sin una biblioteca de tipos; como el interceptor se instancia localmente, preferiblemente dentro de una DLL, no hace falta *marshaling*. Esto significa que no podemos utilizar los asistentes ActiveX de Delphi para crear la clase COM. De todos modos, se trata de una interfaz con muy pocos métodos, y no cuesta trabajo escribir la clase a mano, usando *TComObject* como clase base.

## Conexiones a través de Internet

A pesar de las pocas exigencias de las conexiones con *TSocketConnection*, muchos sistemas no están en condiciones de aprovecharlas. El principal obstáculo sigue siendo la presencia de cortafuegos. Por este motivo, Borland introdujo, en Delphi 5, otro sistema de *marshaling*, esta vez basado en el protocolo HTTP:



Hay mil formas de materializar físicamente una red con estas características, pero voy a usar un ejemplo muy concreto de configuración, en aras de la claridad. Suponga-

mos que su empresa tiene una tienda en Internet. Los clientes acceden a la tienda a través de una página Web, y solamente necesitan un navegador para poder comprar. Es decir, el software que mueve la tienda es una aplicación para Internet como las que estudiaremos más adelante. El usuario para acceder a la tienda teclea una URL como la siguiente:

`http://www.classiqueCentral.com`

La página que aparece es generada por un ordenador que contiene los siguientes ingredientes:

- 1 Internet Information Server (preferiblemente la versión 5 o superior).
- 2 Su aplicación, una DLL que es ejecutada por I.I.S.
- 3 Un servidor de bases de datos. Lo normal sería mover este servidor a otro ordenador, pero voy a suponer que mi cliente es más pobre que las ratas pobres. La DLL de la tienda accede a una de sus bases de datos y la modifica.

Su empresa se ocupa de la gestión posventa de la tienda, para lo que necesita una aplicación diferente a la que utilizan los clientes para comprar. Esta segunda aplicación podría consistir también en una aplicación para Internet, y los empleados llevarían la cuenta de los pedidos, envíos, pagos y cancelaciones a través de una página HTML dinámica... ¡me entran escalofríos cuando me pongo en el lugar de un empleado! Porque la interacción con HTML, a pesar de estar de moda, es una verdadera porquería: no hay árboles, no hay arrastrar y soltar, ni barras de herramientas flotantes. Y los problemas de compatibilidad entre navegadores mueven al desarrollador a desarrollar las interfaces visuales más sencillas posibles.

Por consenso general, en su empresa claman por una aplicación de interfaz gráfica tradicional. Si el servidor de Internet reside en la misma empresa, y existe una red interna local para acceder a él, no hay necesidad de seguir la explicación. Con el ritmo actual de las conexiones DSL y por cable, puede que en poco tiempo sea ésta la situación más frecuente. Supongamos, por el contrario, que el servidor está ubicado en las instalaciones de un proveedor de alojamiento externo. ¿Hay un administrador de red competente en la empresa proveedora? De ser afirmativa la respuesta, pídale que abra el puerto 211 para las conexiones al servidor. Instale en esa máquina un servidor de capa intermedia, el programa *sktsrvr*, y desarrolle una aplicación cliente que obtenga sus datos a través de un *TSocketConnection*.

Pero no. Su proveedor no quiere ni oír hablar de otro puerto que no sea el puerto 80, que es donde se reciben las peticiones HTTP; probablemente es que *no conoce* otro puerto que el 80. Cambiamos nuestros planes, en consecuencia: en vez de copiar *sktsrvr*, instalamos una DLL que también distribuye Borland en el directorio *bin* de Delphi: *httpsrvr.dll*. Se trata de una aplicación ISAPI, que se carga dentro del proceso del Internet Information Server. Y en la aplicación cliente utilizamos un *TWebConnection* para las conexiones.

Cada vez que la aplicación cliente necesita acceder al servidor, utiliza el *TWebConnection* como *proxy* que imita la interfaz del módulo remoto. La conexión, a su vez, envía los datos de la petición al servidor, utilizando una DLL que acompaña a Internet Explorer desde su versión 4: *wininet.dll*. Dicho en otras palabras, *TWebConnection* actúa como un cliente HTTP, y se aprovecha del código de soporte del cliente HTTP más popular en Windows, que es el Internet Explorer.

En el lado servidor, las peticiones son atendidas por Internet Information Server, y si hemos configurado correctamente el componente de conexión, éste pasa el balón a *httpssrvr.dll*, que hace el papel de *stub*: crea los módulos COM y realiza llamadas a sus métodos en representación nuestra. Los datos obtenidos son vueltos a empaquetar en formato HTML y son enviados de vuelta al cliente.

## Configuración de *TWebConnection*

Traiga un componente *TWebConnection* a un módulo de datos, en una nueva aplicación. Al igual que con los componentes de conexión que ya hemos visto, hay que comenzar por indicar la dirección del servidor. Si se trata de una conexión Web, hay que usar su propiedad *URL*, que por omisión tiene como valor:

```
http://server.company.com/scripts/httpssrvr.dll
```

Voy a suponer que hay algún servidor HTTP instalado en su ordenador. En este caso, debería cambiar el valor de *URL* a algo como:

```
http://localhost/scripts/httpssrvr.dll
```

Debe entonces comprobar que su servidor local define un directorio virtual llamado *scripts*, y que se han concedido permisos de ejecución de aplicaciones sobre este directorio. En el capítulo 35 explicaremos cómo hacerlo, pero estas opciones vienen configuradas por omisión para casi todos los servidores HTTP. Localice el directorio físico que corresponde al *scripts* virtual, y copie en su interior el fichero *httpssrvr.dll*, desde el directorio *bin* de Delphi.

Segundo paso: seleccione la propiedad *ServerName*, y despliegue la lista de servidores registrados. El *stub* de las conexiones Web nos presta el mismo servicio que *sockssrvr*: puede entregarnos la lista de servidores registrados en un ordenador remoto. Por este motivo, también existe una función *EnableWebTransport* que debe llamarse desde el método de registro del módulo remoto para habilitar este tipo de conexión.

A partir de este momento, sin embargo, encontraremos algunas propiedades adicionales para lidiar con las peculiaridades del protocolo HTTP. Por ejemplo, puede que algunos ordenadores no tengan acceso directo al servidor HTTP, sino que tengan que utilizar un *proxy*, en el sentido que tiene esta palabra en Internet. En tal caso, tenemos una propiedad *Proxy* para asignar los nombres o direcciones de uno o más *proxies*, separados por puntos y comas.

Es también posible que el directorio virtual remoto donde está ubicada la DLL que actúa como *stub*, tenga el acceso protegido por contraseña. Es una forma más de evitar que algún extraño haga uso del servidor de capa intermedia. Si esto sucediese, *TWebConnection* cuenta con las propiedades *UserName* y *Password* para que indiquemos el nombre de usuario y su contraseña. Más adelante veremos cómo hacer para que el usuario teclee estos valores al establecer la conexión.

## La caché de módulos

Hay un beneficio importante al realizar las conexiones a través de *TWebConnection*, y consiste en que *httpsrvr.dll* implementa para nosotros una caché de módulos remotos. En parte, se trata de una necesidad impuesta por las características de HTTP. En este protocolo cada petición es independiente de las demás; no se puede mantener una conexión activa de forma indefinida. Cuando *TWebConnection* ha recibido la respuesta a una de sus peticiones, se pierde el enlace con el módulo que la contestó. Si destruyésemos entonces ese módulo, el coste de volver a crearlo sería excesivo; recuerde que el módulo debe ir acompañado de al menos una conexión a una base de datos.

Como no hemos estudiado todavía el funcionamiento de HTTP, no puedo dar una explicación detallada de cómo funciona la caché de módulos de *httpsrvr*, por lo que nos limitaremos a una descripción funcional. En primer lugar, para activar el mecanismo hay que añadir algunas claves en el registro mediante el siguiente procedimiento:

```
procedure RegisterPooled(
  const ClassID: string;
  Max, Timeout: Integer;
  Singleton: Boolean = False);
```

El lugar idónea para hacerlo es la implementación de *UpdateRegistry* en el módulo remoto. Por ejemplo:

```
class procedure TDatabase.UpdateRegistry(Register: Boolean;
  const ClassID, ProgID: string);
begin
  if Register then begin
    inherited UpdateRegistry(Register, ClassID, ProgID);
    EnableSocketTransport(ClassID);
    EnableWebTransport(ClassID);
    RegisterPooled(ClassID, 16, 30, False);
  end
  else begin
    DisableSocketTransport(ClassID);
    DisableWebTransport(ClassID);
    UnregisterPooled(ClassID);
    inherited UpdateRegistry(Register, ClassID, ProgID);
  end;
end;
```

Veamos ahora los parámetros de *RegisterPooled*. El primero, *ClassID*, no necesita presentación, pues es el identificador de clase del módulo, y sirve para saber en qué rama

del registro hay que almacenar los restantes parámetros. *Max* indica el número máximo de módulos que pueden ser almacenados en la caché. Este número depende del mayor número de peticiones concurrentes que deba atender el servidor, y está muy ligado al tiempo que se tarde en satisfacer cada una de ellas. Mientras más corto sea, menos necesidad tendremos de módulos en paralelo. He pedido 16 instancias en el ejemplo anterior; eso no quiere decir que se pueda atender a un máximo de 16 usuarios. Sólo que si se da la coincidencia de que diecisiete usuarios piden datos simultáneamente, uno de ellos tendrá que esperar.

El parámetro *Timeout* indica qué tiempo puede permanecer un módulo inactivo en la caché, expresado en minutos. Supongamos que llega una ráfaga de peticiones inusual, y que el servidor se ve obligado a crear los 16 módulos del ejemplo. Luego, sobreviene la calma. ¿Qué sentido tiene mantener esos 16 módulos, con sus respectivas conexiones a la base de datos, en total ociosidad? Si transcurren 30 minutos desde la última petición atendida, se puede destruir un módulo. De todos modos, la búsqueda de módulos ociosos se dispara cada 6 minutos, en la versión actual de *httpsrvr.dll*.

Por último, si activa *Singleton* se utilizará un modelo muy curioso: solamente se creará un módulo para todas las peticiones. Si el modelo de concurrencia del servidor COM es *Apartment*, eso significa que todas las peticiones serán serializadas mediante una cola de mensajes. Malo, ¿verdad? Y si el modelo es *Free*, puede que más de un hilo acceda simultáneamente a las variables internas del módulo. Demasiado trabajo. No se lo recomiendo.

#### **Y NO SE PERMITEN EVENTOS...**

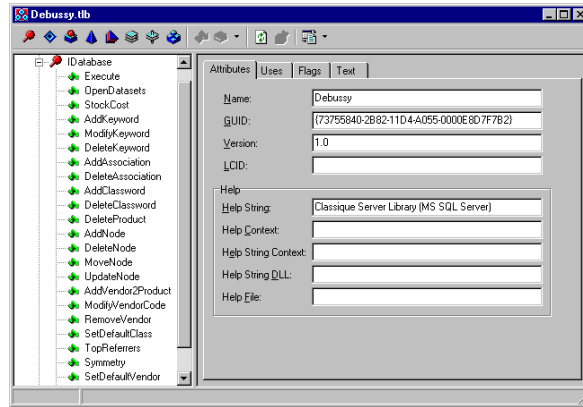
Esas eran las buenas noticias. Ahora las malas: HTTP es un protocolo en el que la iniciativa corre siempre a cargo del cliente. Esto implica que el servidor no puede disparar eventos, porque tendría que tomar la iniciativa en la conexión. Se trata de una limitación compartida con el famoso modelo SOAP, el último grito de la moda, al menos cuando utiliza HTTP para la transmisión.

## **Extendiendo la interfaz del servidor**

Rebobinemos. Cada vez que creamos un módulo de datos remoto, Delphi define una nueva interfaz derivada de *IAppServer*. Sin embargo, podríamos habernos limitado a implementar la interfaz existente. ¿Por qué tomarnos el trabajo de definir un nuevo tipo de interfaz? En parte, porque consiste en un elemento de identidad adicional para el servidor remoto. Pero la razón principal es que así podemos añadir nuevos métodos remotos al servidor.

Creo que no le costará mucho esfuerzo la creación de los métodos en sí: hay una biblioteca de tipos asociada al proyecto. Solamente tiene que ejecutar el comando de menú *View | Type Library* y definir el prototipo del método en IDL o utilizando el editor gráfico. La pregunta es, más bien, ¿para qué crear nuevos métodos?





El motivo más frecuente es poder ejecutar procedimientos almacenados en el servidor. Es verdad que se puede resolver igual a través de un proveedor que tenga la opción *po.AllowCommandText*. Se le asocia un *TClientDataSet*, se pone la llamada al procedimiento, parámetros incluidos, en su *CommandText* y se ejecuta su método *Execute*. Si tenemos más deseos de trabajar, podemos crear una lista de parámetros explícita, sobre todo para recibir parámetros de salida. O incluso podemos llamar directamente al método *AS\_Execute* de la interfaz *IAppServer*; en la próxima sección explicaré cómo.

A pesar de lo anterior, prefiero siempre crear un método remoto en esos casos. Es muy fácil equivocarse al teclear el nombre del procedimiento, o en el orden de los parámetros, o incluso en el tipo. Es más, considero que es malo tener un canal tan genérico de comunicación con el servidor, porque en ningún lugar queda constancia de que el cliente usa tal o más cual procedimiento del servidor. Para rematar, si ejecutamos directamente procedimientos del servidor nos estamos atando a las manías sintácticas de un dialecto particular del SQL.

Este fragmento de código corresponde a una aplicación “real”, y es la definición en IDL de algunos de los métodos definidos para un servidor de capa intermedia; en concreto, los métodos de gestión de un árbol de categorías:

```
[ uuid(73755841-2B82-11D4-A055-0000E8D7F7B2), version(1.0),
  helpstring("Dispatch interface for Database Object"),
  dual, oleautomation ]
interface IDatabase: IAppServer {
    /* ... */
    [id(0x0000000C)]
    HRESULT STDMETHODCALLTYPE AddNode(
        [in] long AParentID, [in] long AKeywordID,
        [out, retval] long * ATreeNodeID );
    [id(0x0000000D)]
    HRESULT STDMETHODCALLTYPE DeleteNode(
        [in] long ATreeNodeID );
    [id(0x0000000E)]
    HRESULT STDMETHODCALLTYPE MoveNode(
        [in] long ATreeNodeID, [in] long AParentID );
    [id(0x0000000F)]
```

```

HRESULT stdcall UpdateNode(
    [in] long ATreeNodeID, [in] BSTR ANewName );
/* ... */
};

```

La implementación de *AddNode* es la siguiente (la aplicación usa ADO Express):

```

function TDatabase.AddNode(AParentID, AKeywordID: Integer): Integer;
begin
    with spAddNode.Parameters do
        begin
            ParamByName('@parentID').Value := AParentID;
            ParamByName('@keywordID').Value := AKeywordID;
            Execute(spAddNode);
            Result := ParamByName('@treeNodeID').Value;
        end;
    end;

```

Preste mucha atención a la implementación de mi método *Execute*:

```

procedure TDatabase.Execute(AProc: TADOStoredProc);
var
    DoTrans: Boolean;
begin
    DoTrans := not Data.InTransaction;
    if DoTrans then Data.BeginTrans;
    try
        AProc.ExecProc;
        if DoTrans then Data.CommitTrans;
    except
        if DoTrans then Data.RollbackTrans;
        raise;
    end;
end;

```

Primero compruebo que no haya una transacción activa; si no la hay, encierro la llamada al procedimiento dentro de una nueva transacción. Esto es algo que hay que recordar cuando se llama a un método remoto. Recuerde que durante la resolución, el proveedor inicia una transacción automáticamente. En cambio, ahora tenemos que hacerlo explícitamente. Para poder reutilizar *AddNode* dentro de alguna fase de la resolución, si se diese el caso, tengo en cuenta los dos escenarios, dentro del método interno *Execute*.

Debo advertirle que aquí se enfrentará a un compromiso. Si añade muchos métodos adicionales a la interfaz del servidor, estará dejando un camino muy bien marcado a la persona que tenga que crear la aplicación cliente, ¡que puede ser usted mismo! Pero eso significará también añadir trabajo al desarrollador del servidor de capa intermedia. Y lo peor vendrá cuando el programador de la capa cliente detecte, o crea detectar, la necesidad de nuevos métodos en el servidor. Como en todo, la solución es usar el sentido común...

## Utilizando la interfaz del servidor

Después de añadir métodos a la interfaz remota, tenemos que saber cómo ejecutar los métodos que acabamos de implementar en el servidor de capa intermedia. Pero esto es relativamente fácil, si tenemos un poco de cuidado. Debemos obtener primero el puntero a la interfaz *LAppServer* que crea el componente de conexión remota. Existen dos propiedades con este propósito:

```
function TCustomRemoteServer.GetServer: IAppServer;
property TCustomRemoteServer.AppServer: Variant;
```

Ya sabe: mediante *GetServer* tiene acceso directo a la *v-table* del proxy que representa al servidor en el lado cliente. Si utiliza *GetServer*, podrá verificar en tiempo de compilación tanto el nombre de los métodos que llame, como el número y el tipo de cada uno de los parámetros. En cambio, con *AppServer* trabajará con la interfaz *IDispatch*, la misma que emplean los intérpretes que soportan automatización OLE.

Lamentablemente, las cosas no son tan sencillas. Los métodos que hemos añadido pertenecen a la interfaz *IMyServer*, no a *LAppServer*. Es cierto que podemos llamar a *QueryInterface* para, partiendo de *LAppServer*, recuperar cualquier interfaz implementada por el objeto remoto o, lo que es igual, podemos utilizar el operador **as**. El primer paso consistiría en copiar la unidad con sufijo *\_tlb* que se genera en paralelo con la biblioteca de tipos del servidor, e incluirla en el proyecto cliente; sería incluso preferible que el proyecto cliente haga referencia directamente a la unidad original. Entonces, si estamos utilizando DCOM para conectarnos al servidor, podemos añadir una función al módulo de datos que contiene el componente de conexión, como muestro a continuación:

```
function TmodDatos.Servidor: IMyServer;
begin
    // ¡SOLO CON TDCOMConnection!
    Result := DCOMConnection1.GetServer as IMyServer;
end;
```

Espero que haya leído el comentario: sólo podemos hacer la conversión entre interfaces si la conexión se realiza sobre DCOM. Si la hacemos a través de *sockets* o de HTTP, recibiremos un mensaje de error, ¡porque el software de *marshaling* sólo se ocupa de transportar la interfaz de automatización! Pero en tal caso, podemos evitar también el trabajo con el tipo *Variant*. Observe este método alternativo:

```
function TmodDatos.Servidor: IMyServerDisp;
begin
    // Otra versión diferente (no se puede usar overload)
    Result := IMyServerDisp(SocketConnection1.GetServer);
end;
```

En este caso, estamos recurriendo a la **dispinterface** que se declara al traducir la biblioteca de tipos del servidor, y que sí podemos utilizar, independientemente del tipo de *marshaling* que estemos empleando. No es tan rápido como ejecutar el método

a través de la *v-table*, pero a diferencia de la ejecución a través de variantes, no hay que traducir primero una cadena con el nombre del método (y sus parámetros) a los números internos, o **dispid**, que acepta el método genérico *Invoke*. De esta forma, además, el compilador puede verificar si estamos llamando a un método existente, si los parámetros coinciden en número y tipo, etc.

## Alguien llama a mi puerta

Cuando se trabaja con aplicaciones divididas en capa, y es un servidor remoto el único que tiene acceso al servidor SQL, ¿debemos seguir utilizando el sistema de seguridad que ofrece la base de datos? Sé que mi respuesta traerá polémica, pero la respuesta es: indudablemente NO. Maticemos. Si va a existir un módulo remoto por cada cliente, si no se van a “reciclar” los módulos, no hay problema alguno en utilizar los nombres y permisos de la base de datos SQL. Pero es muy poco probable que se presente la situación anterior. En primer lugar, si las conexiones se realizan a través de *TWebConnection*, es simplemente imposible. En segundo lugar, incluso cuando todas las capas de la aplicación están ubicadas en una red rápida de área local, es casi siempre una tontería dejar que cada cliente tenga una conexión separada al servidor SQL, porque convertiríamos lo convertiríamos en el cuello de botella del sistema.

Si reutilizamos los módulos remotos, en cambio, tendríamos problemas cuando dos usuarios diferentes utilizaran un mismo módulo de forma consecutiva. Habría que cerrar la conexión a la base de datos al terminar cada petición. Y perderíamos una de las ventajas de la caché de módulo. Por este motivo, en las aplicaciones multicapas el acceso SQL suele realizarse a través de una cuenta SQL reservada para este único propósito.

... lo que no quiere decir que tengamos que renunciar a toda forma de autenticación. Si los permisos deben concederse mediante la lógica de la aplicación, debemos estar seguros que cada cuál es quien dice ser. Necesitamos dos cosas:

- 1 Hay que diseñar un método remoto que verifique si una combinación usuario más contraseña es aceptable o no.
- 2 Sería conveniente disponer de algún evento o mecanismo implícito para que se solicitasen dichos datos al usuario que intenta establecer una conexión.

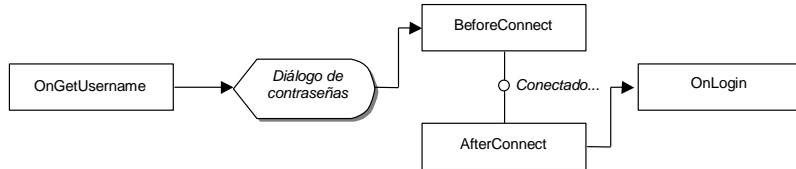
El método remoto del primer punto puede ser implementado por el mismo servidor de capa intermedia, pero también podría pertenecer a una capa separada que se ocupase de los temas de seguridad. El método podría tener un prototipo similar al siguiente:

```
function VerificarUsuario(const User, Password: string): Boolean;
```

Como ve, es preferible que *VerificarUsuario* devuelva un valor lógico, y que sea el cliente el que lance una excepción si hay problemas. La alternativa sería dejar que la excepción fuese lanzada por el servidor. No habría problemas con propagarla al

cliente, si se utiliza el convenio **safecall** de llamadas. Pero tendríamos entonces problemas con el idioma del mensaje de error.

En cuanto al segundo punto, todos los componentes de conexión remota implementan varios eventos que pueden sernos de gran ayuda. Esta es la secuencia de acontecimientos que se producen cuando se intenta activar una conexión remota:



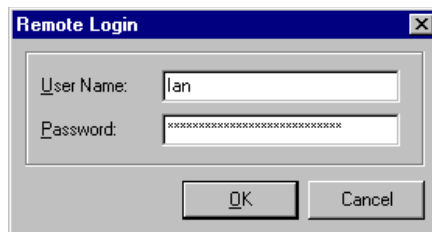
El primer evento que se dispara es *OnGetUsername*, y su único propósito es que podamos inicializar, si queremos, el nombre de usuario que se le propondrá al usuario. Por ejemplo, en el siguiente método se obtiene el nombre del usuario activo en Windows:

```

procedure TmodDatos.DCOMConnection1GetUsername (
    Sender: TObject; var Username: string);
var
    Buffer: array [0..255] of Char;
    Size: Cardinal;
begin
    StrPCopy(Buffer, Username);
    Size := SizeOf(Buffer);
    GetUserName(Buffer, Size);
    UserName := StrPas(Buffer);
end;
  
```

Usted puede, por supuesto, inicializar el nombre del usuario como le dé la gana. Por ejemplo, podría también almacenar el nombre utilizado en la última conexión en el registro de Windows para recuperarlo en este método.

Lo que sucede a continuación depende de dos condiciones: si *LoginPrompt* vale *True* y el proyecto ha incluido la unidad *DBLogDlg*, se muestra el diálogo predefinido de la VCL para pedir el nombre y contraseña del usuario:



Si el usuario cancela el diálogo, se produce una excepción que aborta el proceso. Si pulsa *OK*, el nombre y la contraseña se guardan en variables locales. Y si *LoginPrompt*

es falso, o no hemos incluido la unidad *DBLogDlg*, esas mismas variables locales se quedan inicializadas con cadenas vacías.

En cualquier caso, el siguiente paso consiste en activar realmente la conexión, disparando los inevitables eventos *Before* y *After*. Lo más interesante es esto: ¡el nombre de usuario y la contraseña no intervienen para nada en este proceso! Piense un instante, ¿para qué iban a servir?

Entonces, si *LoginPrompt* está activo, se dispara el evento *OnLogin*. Usted pensará que a buenas horas... Pero es que necesitaremos que la conexión se haya establecido para poder autenticar al usuario:

```
procedure TmodDatos.DCOMConnection1Login(
  Sender: TObject; Username, Password: string);
begin
  with TDCOMConnection(Sender) do
    if not (GetServer as IMyServer).VerificarUsuario(
      UserName, Password) then begin
        Connected := False;
        raise Exception.Create(';Largo de aquí!');
      end;
  end;
end;
```

Es aquí donde está la explicación: Delphi asume que el servicio de autenticación se ha implementado dentro del mismo servidor remoto, y sabe que no puede utilizarlo hasta que no se haya activado el servidor. Imagino que no estará muy convencido de que todo esto tenga sentido. Espere un poco, por favor...

Debo aclarar que el mecanismo anterior es *completamente independiente* del uso que se le da a las propiedades *UserName* y *Password* de *TWebConnection*. Como comprenderá, estas propiedades deben estar asignadas *antes* de establecer la conexión, lo que significa que tendrá que crear su propio diálogo de identificación, y asignar manualmente lo que teclee el usuario en las correspondientes propiedades de la conexión. Si lo desea, podrá utilizar posteriormente el evento *OnLogin* para realizar una segunda verificación, ya a nivel de la aplicación.

## Seguridad por medio de *tokens*

Lo que voy a explicar en esta sección es fruto de mi propia cosecha. Así que pido perdón si se me escapa algún fallo. Para empezar, le confieso que el sistema de seguridad de las conexiones remotas no resuelve mucho en su estado original. Nada impide a un *hacker* programador crear su propia aplicación que acceda al servidor remoto y se salte la verificación del evento *OnLogin*. En realidad, es muy sencillo “craquear”<sup>30</sup> incluso nuestra propia aplicación, porque se puede localizar la implementación del evento con mucha facilidad. Y basta un salto al final del método para desactivar la protección.

---

<sup>30</sup> ¡Perdóname, Cervantes!

Por supuesto, a partir de este punto hay muchas maneras de mejorar lo existente, y voy a explicarle una muy sencilla, que no consume demasiados recursos ni tiempo de programación. Consiste en los siguientes pasos:

- 1 Debemos ampliar el método remoto de autenticación, de manera que éste entregue al cliente un *token* o ficha (o vale, o resguardo, como prefiera).
- 2 Como en los parques de atracciones, cada vez que el cliente realice una petición al servidor deberá presentar su acreditación.
- 3 El servidor dispondrá de una lista de acreditaciones extendidas, para ver si el cliente ha pagado, o si está intentando montarse en la montaña rusa por toda su cara.

El *token* debe ser un valor aleatorio que sea difícil de adivinar por parte de un tramposo. Mi sugerencia es usar un número aleatorio de al menos 64 o 128 bits. Los GUIDs no nos valdrían, porque 48 de sus 128 bits serían predecibles; me refiero al número MAC del adaptador de red del servidor.

Lo que puede parecer más complicado es lograr que el cliente siempre presente su acreditación, pero aquí podemos aprovechar una característica ya conocida de la interfaz *LAppServer*: la posibilidad de pasar un valor en el parámetro *OwnerData* de casi todos sus métodos. Podríamos forzar a todos los conjuntos de datos clientes de nuestra aplicación para que tengan manejadores compartidos para sus eventos *BeforeGetRecords*, *BeforeApplyUpdates*, *BeforeRowRequest* y *BeforeExecute*, y en esos métodos se copiaría el *token* recibido para enviarlo al servidor. En el servidor compartiríamos los mismos eventos, pero esta vez en los proveedores.

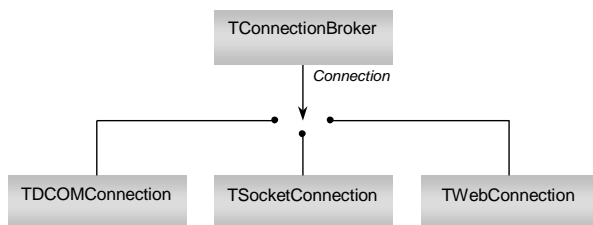
#### ADVERTENCIA

Solamente habría que tener cuidado si estamos utilizando *OwnerData* para implementar la carga incremental de registros. Pero son detalles que se pueden arreglar.

## Cambio de conexión

Yo desarrollo normalmente en una red local, en la que tengo a mi disposición un servidor de dominio, varios servidores de bases de datos, e incluso un ordenador con Internet Information Services, para hacer pruebas con Internet. Pero a veces debo llevarme los proyectos en un portátil, y trabajar en solitario, ya sea en un aeropuerto o en la habitación del hotel. A veces he llegado al atrevimiento de sacar el portátil a la piscina: había un grupo de suecas haciendo *topless*, ¡qué demonios! El condenado aparato se calentaba como una cafetera y se reiniciaba a cada rato, pero el espectáculo merecía la pena.

¿Cómo hacer para que, en algunos casos, una aplicación se conecte a su servidor a través de DCOM, o a través de zócalos, o usando HTTP? En los viejos tiempos, tenía que editar directamente el *dflm* del módulo de datos, en el bloc de notas, antes de abrir el fichero. A partir de Delphi 6, puedo utilizar el componente *TConnectionBroker*, y así me queda más tiempo para mirar a las suecas.



*TConnectionBroker* funciona igual que la mayoría de los analistas que he conocido: que por sí mismos, no saben mover una línea de código, pero a pesar de todo se venden como si lo supieran. Y eso sí, cuentan con un ejército de programadores para cuando tienen que demostrarlo... Quiero decir, que *ConnectionBroker* implementa los métodos abstractos de la clase *TCustomRemoteServer* delegándolos en otro componente de conexión, al que hacer referencia por medio de su propiedad *Connection*.

¿Cómo utilizarlo? Cuando comience a desarrollar la aplicación cliente, arroje al módulo de datos todos los tipos de conexión que vaya a emplear: DCOM, *sockets*, Web; si su cliente es masoquista, pruebe también con *TCorbaConnection*. Configúrelas todas lo mejor que pueda... pero déjelas cerradas. Asigne entonces un jefe al proyecto... perdón, traiga un *TConnectionBroker*. Todos los *TClientDataSet* que utilice a partir de este momento, deben apuntar a este último componente. Asigne entonces la conexión DCOM en la propiedad *Connection* de *ConnectionBroker1*, y active entonces este último componente. Recomiendo que utilice *TDCOMConnection* como conexión real durante el desarrollo para que no tenga que cargar siempre con un servidor HTTP activo para las pruebas.

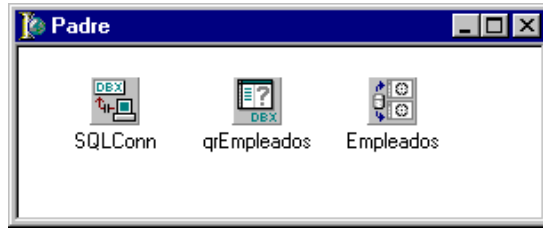
¿Y en tiempo de ejecución? Deberá entonces usar algún artificio de configuración externa, como un fichero *ini* o el registro de Windows, para que el programa, al iniciarse, sepa con qué tipo de conexión deberá trabajar. Solamente tendrá que cambiar una propiedad: *Connection*, en el controlador de conexiones.

## Conexiones compartidas

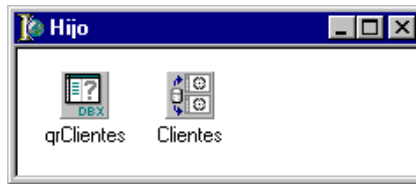
Y ahora nos ocuparemos de un problema congestivo: en las aplicaciones reales, como decía al inicio del capítulo, se debe configurar un número elevado de proveedores y sus conjuntos de datos asociados en los módulos remotos. Siempre llega el momento en que no cabe un alfiler más en la superficie del módulo. ¿Qué se puede hacer entonces?

¡Crear un segundo módulo, por supuesto! El truco consistirá en evitar la creación de una segunda conexión independiente. Vamos a iniciar una aplicación servidora, en la crearemos primero un módulo remoto con el nombre de *Padre*. Deje caer en él un componente *TSQLConnection*, y conéctelo a la base de datos de siempre, a la *mastsql*. Luego traiga una consulta o una tabla, da lo mismo, y conéctela a la tabla que le plazca; yo utilizaré la de *Empleados*. No olvidé acoplarle un *TDataSetProvider*.





Imaginemos que se nos ha agotado el espacio para seguir poniendo componentes. Entonces creamos, dentro del mismo proyecto, un segundo módulo de datos al que llamaremos *Hijo*. Y en el prepararemos las condiciones para publicar datos de clientes:



Un momento, ¿dónde está la conexión? Bueno, he recurrido a un truco sucio: primero, he añadido la unidad del módulo padre en la lista **uses** del hijo. Y he forzado *qrClientes* para que apunte a la conexión del padre. Así puedo comprobar las consultas en tiempo de diseño, configurar campos, etc. Naturalmente, en tiempo de ejecución no valdrá el sistema que siempre usa Delphi para resolver las referencias entre módulos. Por este motivo he redefinido el método *Loaded* en el módulo hijo:

```
procedure THijo.Loaded;
begin
    inherited Loaded;
    qrClientes.SQLConnection := nil;
end;
```

Así elimino, en tiempo de ejecución, la referencia establecida en tiempo de diseño. Claro, tenemos que preparar algún método para poder asignarle una conexión a la consulta. Lo que he hecho es añadir una propiedad *SQLConnection* a la interfaz *IHijo*, utilizando el editor de la biblioteca de tipos:

```
[ uuid(735150A7-03D3-11D6-8943-00C026263861), version(1.0),
  helpstring("Dispatch interface for Hijo Object"),
  dual, oleautomation ]
interface IHijo: IAppServer {
    [ propget, id(0x00000001) ]
    HRESULT _stdcall SQLConnection( [out, retval] long * Value );
    [ propput, id(0x00000001) ]
    HRESULT _stdcall SQLConnection( [in] long Value );
};
```

¿Ha visto que la propiedad es de tipo entero? Es que no podemos utilizar directamente un puntero a un componente, porque no se podría realizar el *marshaling* de

*IHijo*. Pero solamente utilizaremos *SQLConnection* localmente, así que no hay peligro en realizar conversiones de tipos tan descaradas. Esta es la implementación de los métodos de acceso de *SQLConnection*:

```
function THijo.Get_SQLConnection: Integer;
begin
    Result := Integer(qrClientes.SQLConnection);
end;

procedure THijo.Set_SQLConnection(Value: Integer);
begin
    qrClientes.SQLConnection := TSQLConnection(Value);
end;
```

Esto, sin embargo, no ha hecho más que empezar. Añada la siguiente declaración en la sección de interfaz de la unidad del módulo hijo:

```
var
    FabricaHijo: TComponentFactory;
```

Luego, modifique la sección de inicialización de la misma unidad del siguiente modo:

```
initialization
    FabricaHijo := TComponentFactory.Create(ComServer, THijo,
        Class_Hijo, ciInternal, tmApartment);
end.
```

Hay dos cambios: en primer lugar, el objeto que actúa como fábrica de clases para los módulos hijos se guarda en una variable, nuestra *FabricaHijo*, para ser utilizada más adelante dentro del servidor, de forma interna. Observe además que hemos modificado el valor del penúltimo parámetro del constructor de la fábrica, para que solamente se atiendan peticiones internas de creación. Enseguida veremos por qué.

Regresamos al editor de la biblioteca de tipos, pero esta vez modificaremos la interfaz *IPadre*, añadiéndole una propiedad de sólo lectura. El nombre que le demos da igual, pero yo la llamaré *Hijo*. La nueva propiedad deberá devolver un puntero de interfaz de tipo *IHijo*:

```
[ propget, id(0x00000001) ]
HRESULT _stdcall Hijo( [out, retval] IHijo ** Value );
```

Esta es su implementación:

```
function TPadre.Get_Hijo: IHijo;
begin
    Result := FabricaHijo.CreateComObject(nil) as IHijo;
    ASSERT(Result.SQLConnection = 0);
    Result.SQLConnection := Integer(SQLConn);
end;
```

Como ve, para esto nos habíamos quedado con el puntero a la fábrica de clases del hijo. La aserción comprueba que el truco con *Loaded* funcione correctamente, elimi-

nando la referencia inicial, y a continuación se asigna el componente de conexión correcto por medio de la propiedad de lectura y escritura que creamos en el hijo. Y ya está listo el servidor.

Del cliente, que será muy sencillo, solamente mostraré el módulo de datos:



Para acceder a los proveedores situados en el módulo principal, se utiliza un componente de conexión “normal”; en este caso, un *TDCOMConnection*. El conjunto de datos *Empleado* obtiene su proveedor a través de esta primera conexión.

Para acceder al segundo módulo, hay que traer un *TSharedConnection* al módulo. Primero hay que asignar su propiedad *ParentConnection*, para que apunte al componente de conexión asociado al módulo padre. Y finalmente, hay que asignar en *ChildName* el nombre de la propiedad, dentro de la interfaz del módulo padre, que apunta al módulo hijo. En este ejemplo, la propiedad se llama *Hijo*.

## Las conexiones locales

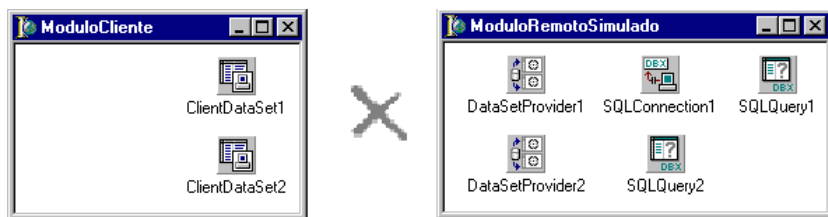
Acabamos de conocer una mutación interesante de *TCustomerRemoteServer*, y ahora voy a presentarle otra más: *TLocalConnection*. Este componente existe gracias a DB Express, y resuelve un problema peliagudo que se arrastró moribundo hasta la aparición de Delphi 6.

¿Cómo hemos utilizado los proveedores en las aplicaciones de los tres capítulos anteriores? Como no quería adelantar el uso de aplicaciones en tres capas, poníamos todos los componentes en una misma capa: tanto los conjuntos de datos de origen, casi siempre de DB Express, y los proveedores asociados, como los conjuntos de datos clientes. ¿Resultado? Una mezcla difícil de mantener; por suerte para nosotros, se trataba de aplicaciones muy pequeñas.

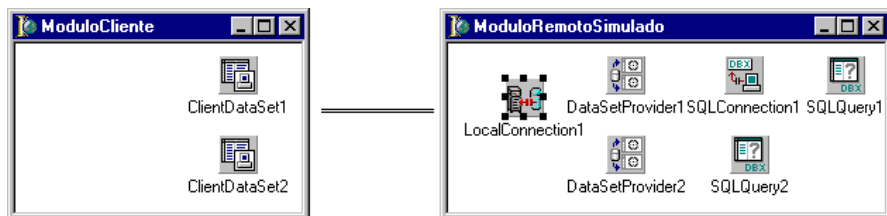
En las versiones anteriores de Delphi, todas las interfaces de acceso que se incluían proporcionaban medios para la navegación bidireccional y la actualización sobre conjunto de datos, al estilo RAD. Si alguien quería trabajar con conjuntos de datos clientes, no había ningún cargo de conciencia en obligarle a dividir su aplicación en capas. Pero llega entonces Delphi 6 e introduce DB Express: si quiere trabajar al modo RAD, tendrá que utilizar *TClientDataSet* y proveedores. ¿Debe Borland obli-

garle a dividir su aplicación en capas? Aunque hemos visto que esa división no es costosa, en términos de eficiencia, si el servidor se plantea como una DLL dentro del proceso, no sería de recibo forzar una forma de trabajo incluso para aplicaciones pequeñas.

Ahora bien, *TClientDataSet* tiene serias dificultades para conectarse con un *TDataSetProvider* que se encuentre en otro módulo de datos. Si hemos sido organizados y tenemos los proveedores y los componentes SQL en un módulo aparte, simulando un módulo remoto, nos encontraremos con que los conjuntos de datos clientes no tienen acceso a los nombres de proveedores:



Lo que tenemos que hacer, en ese caso, es traer un *TLocalConnection* al módulo que contiene los proveedores:



*TLocalConnection* se limita a implementar la interfaz *LAppServer* internamente, como si él mismo fuese un módulo remoto. Los conjuntos de datos cliente que se conecten a un *TLocalConnection*, mediante sus propiedades *Remote.Server*, podrán seleccionar cualquiera de los proveedores que se encuentren en el mismo módulo de datos que la conexión local. Y si necesitamos más de un módulo SQL, no hay problemas: bastará con añadir otro *TLocalConnection*.

## Balance de carga

Para terminar, veremos cómo Delphi nos facilita los primeros pasos en la implantación de una técnica conocida como *balance de carga*. Pero antes debo plantearle una pregunta: ¿merece la pena dividir una aplicación en capas físicas, es decir, separar un ordenador para el servidor de capa intermedia, para ganar en velocidad? Pues sí, merece la pena, pero hay que saber qué en qué circunstancias. Porque hay un motivo muy poderoso que desaconseja la separación: cada vez que se añade una capa de software, se pierde algo de tiempo en la comunicación entre capas. Por lo tanto, lo

que necesitamos es saber qué factores pueden contrarrestar esta inevitable pérdida de eficiencia o incluso sobrepasarla.

Tenemos, en primer lugar, los factores relacionados con la lógica de la aplicación. Si logramos mover parte de esa lógica a la capa intermedia, estaríamos liberando de trabajo al servidor SQL (que como las madres, sólo puede haber uno)... a la vez que seguimos manteniendo algo de centralización en el diseño. Porque el otro extremo sería mover todo ese procesamiento al lado cliente, a la capa de presentación, y eso sería muy malo. ¿Cuánta lógica están “acaparando” nuestros ejemplos de capa intermedia? Seamos sinceros: muy poca. Es cierto que se trata de ejemplos pequeños. En aplicaciones grandes podríamos fracturar la aplicación en módulos: separar, por ejemplo, las tablas de empleados y permisos a otra base de datos, y tratar la seguridad mediante un segundo sistema SQL. Pero la parte principal de la aplicación seguirá siendo controlada por un solitario servidor SQL.

¿Quiere decir que hemos estudiado DataSnap, o Midas, en vano? ¡De ningún modo! Hemos logrado unos cuantos objetivos importantes: trabajar con conjuntos de datos desconectados, tener más control en la forma en que se realizan las actualizaciones... Todo eso sin renunciar a la programación RAD, que es el sacrificio impuesto a los programadores de Java. Pero tenga en cuenta que esta forma de trabajo puede incorporarse a la propia interfaz de acceso. De hecho, Oracle ya ha dado varios pasos importantes en esa dirección...

Lo que quiero que comprenda es que los beneficios físicos siguen siendo muy importantes en los sistemas multicapas. Por ejemplo, ¿se ha parado a pensar cuán costoso es para un servidor atender una conexión de red? Imagine un pobre servidor SQL, luchando para optimizar las consultas estúpidas que recibe... y a la vez dedicando RAM e interrupciones a escuchar los quejidos de cien usuarios. ¿Qué tal si esos cien usuarios se conectan a diez servidores de capa intermedia, más baratos que el servidor SQL, y así este último sólo atienda a diez conexiones? Podemos ir más allá: se puede ubicar al servidor SQL con la batería de servidores de capa intermedia dentro de una superred de 1Gb o incluso de 10Gb, con fibra óptica. Los usuarios finales estarían en uno o más segmentos de una red más barata, de 10/100Mb. Podríamos dividir a los usuarios en grupos: los gilitontos de recursos humanos, que se conecten al servidor Verde, los de contabilidad al Rojo, y los de atención al público, al Azul. La aplicación cliente, que sería la misma para todos ellos, en nuestro ejemplo, tendría simplemente que buscar en el registro cuál es la dirección IP que se le ha asignado al configurar la instalación. Este sería un sistema de *balance estático*.

El paso siguiente en complejidad sería permitir que un cliente escogiese su servidor de acuerdo al tráfico que haya en cada momento. En el momento de establecer la conexión, el cliente utilizaría *algún* algoritmo, en este momento *abstracto*, para decidir el otro extremo de la conexión. Pues bien, Delphi nos echa una mano para esto. En primer lugar, todos los componentes de conexión remota, derivados del componente *TDispatchConnection*, tienen la siguiente propiedad:

```
property TDispatchConnection.ObjectBroker: TCustomObjectBroker;
```

Por su parte, *TCustomObjectBroker* es una clase componente abstracta (reliquia de la era anterior a los tipos de interfaz) que tiene los siguiente métodos virtuales:

```
function TCustomObjectBroker.GetComputerForGUID (
    GUID: TGUID): string; virtual; abstract;
function TCustomObjectBroker.GetComputerForProgID (
    const ProgID): string; virtual; abstract;
function TCustomObjectBroker.GetPortForComputer (
    const ComputerName: string): Integer; virtual; abstract;
```

En otras palabras, por medio de *ObjectBroker* podemos asociar a una conexión remota una *caja negra* que ayudará a escoger dinámicamente un nombre de ordenador y, si hace falta, un puerto TCP/IP para establecer la conexión por zócalos. La única condición necesaria para que el componente de conexión utilice el agente es que las propiedades que normalmente indican la dirección del servidor (*ComputerName*, *Host*, *Address* y *URL*) se encuentren vacías.

El único componente concreto, derivado de *TCustomObjectBroker*, que Delphi ofrece actualmente es *TSimpleObjectBroker*, y su comportamiento hace honor al nombre. Tiene una propiedad, de tipo colección, llamada *Servers*. En ella se almacenan elementos con las siguientes propiedades:

```
property TServerItem.HasFailed: Boolean;      // Pública
property TServerItem.ComputerName: string;  // Publicada
property TServerItem.Port: Integer;          // Publicada
property TServerItem.Enabled: Boolean;       // Publicada
```

Podemos teclear en *Servers*, en tiempo de diseño, el nombre de varios servidores. También tenemos métodos *LoadFromStream* y *SaveFromStream* para guardar y restaurar la lista de servidores a partir de un flujo de datos.

¿Cómo funciona el algoritmo de selección de servidor de *TSimpleObjectBroker*? Depende del valor de su propiedad *LoadBalanced*. Si vale *False*, que es el valor por omisión, se toma el nombre del primer servidor activo (*Enabled*) que no haya fallado antes su conexión (*HasFailed*). Esto no vale de mucho, excepto para ayudar en un sistema de balance estático. La aplicación del usuario final debe contar con algún diálogo de configuración en el que podríamos mover al tope de la lista el servidor asignado al departamento donde trabaja el usuario. ¿La ventaja? Pues que si falla el primer servidor, *TSimpleObjectBroker* no se da por vencido e intenta utilizar el siguiente servidor de la lista. Todo esto sin que el usuario se entere.

Si por el contrario, se activa *LoadBalanced*, la elección del servidor tiene lugar al azar. Se supone que, teniendo un número de mediano a grande de usuarios, este algoritmo repartiría la carga uniformemente entre los servidores. No hace falta que llame a *Randomize*, pues ya lo hace el propio componente. Puede parecer una tontería, pero ¿es tan “terrible” este algoritmo? Resulta que no, que incluso tiene una característica

muy buena: no hay que enviar ni recibir nada de la red para determinar el servidor que utilizaremos.

**CONSEJO**

Si quiere un algoritmo mejor, puede crear su propia clase descendiente de *TCustomObjectBroker*. Puede implementar un segundo servicio en la red, que se comunicaría con los usuarios y los servidores. Estos últimos enviarían un aviso al servicio cada vez que se conectase o desconectase un usuario. Así tendríamos una cuenta fiable de la carga de cada servidor, y podríamos proponer una dirección con mayor acierto. De este modo lograríamos otro objetivo importante: que la lista de servidores se manejase en un puesto central, lo que facilitaría la inclusión o las bajas por mantenimiento de los servidores de capa intermedia.





# 6

## Internet

---

- **El protocolo HTTP**
- **Introducción a HTML**
- **Fundamentos de JavaScript**
- **WebBroker**
- **Mantenimiento del estado**
- **Páginas Activas en el Servidor**
- **WebSnap: conceptos básicos**
- **WebSnap: conjuntos de datos**
- **Servicios Web**

# Parte



## El protocolo HTTP

**C**RISTÓBAL COLÓN FUE UN BUEN SEÑOR QUE un buen día se lanzó a la mar pensando en llegar al Japón. En vez de eso, se dio de narices con América. La historia del protocolo HTTP es similar: no fue diseñado para el uso que actualmente le damos. HTTP se comporta de maravillas cuando se trata de poner las fotos de mi sobrina pequeña en una página Web para asombro de la Humanidad. Pero funciona de pena cuando queremos montar una aplicación de comercio electrónico medianamente decente. Así todo, como diría Galileo, se mueve...

Aunque HTTP es un protocolo fácil de explicar y de entender, muchos programadores tienen una idea bastante difusa sobre su funcionamiento. Y podéis incluirme en ese grupo. Es incluso posible escribir aplicaciones para Internet sin dominar estos detalles. Pero saber no ocupa espacio, y creo que una vez que se entiende mejor qué es lo que sucede detrás del telón, la programación se simplifica enormemente, en vez de complicarse. Este será un capítulo con mucha teoría y poco código.

### Protocolo y lenguaje

Es necesario ser lo suficientemente precisos con las palabras que empleamos. HTTP es una sigla muy parecida a HTML, porque tiene cuatro letras y también comienza con HT. ¿Qué parentesco hay entre ambas?

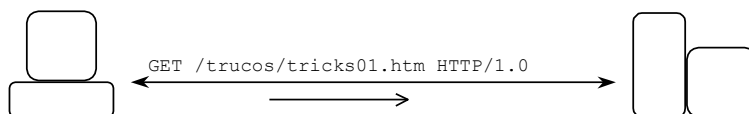
HTTP quiere decir *Hypertext Transfer Protocol*, es decir, protocolo de transferencia de hipertextos. Sea lo que sea un “hipertexto”. Un protocolo debe especificar todos los detalles imaginables acerca de cómo tiene lugar el intercambio entre dos aplicaciones. Yo digo “buenos días”, usted responde “hola, ¿en qué puedo ayudarle?” y es entonces que le pido mi diario favorito con sus suplementos. En este ejemplo las dos aplicaciones somos el quiosquero y yo; en el caso de HTTP son el navegador de Internet, en el lado cliente, y un servidor HTTP, que evidentemente se sitúa en el lado del servidor.

Por otra parte, un quiosco de prensa puede proporcionarnos libros, revistas, incluso chicles, aunque lo más frecuente es que pidamos el periódico. Lo mismo sucede con el protocolo HTTP: lo más probable es que el navegador pida el contenido de un fichero de texto con el formato de una página expresado en el lenguaje HTML (Hy-

*pertext Marking Language*, de modo que las dos letras HT sí indicaban un parentesco importante). Pero es también probable que a un servidor HTTP le pidamos un fichero gráfico, un archivo de ondas o incluso una descripción virtual de un mundo tridimensional. Todo es posible.

## El modelo de interacción en la Web

El protocolo HTTP es un caso particular de arquitectura cliente/servidor. Normalmente utiliza TCP/IP como protocolo de transporte, pero no es una obligación impuesta por su especificación. Para establecer la comunicación, la aplicación cliente (generalmente un navegador, o *browser*) pide al servidor, ubicado en una dirección conocida por el cliente, que abra un conducto para lectura y escritura en su puerto 80. TCP/IP utiliza los números de puertos para permitir que dos ordenadores puedan establecer varios canales de transmisión simultáneos entre sí; cada canal se debe poder identificar unívocamente por el par de direcciones del cliente y del servidor, y por la dirección del puerto especificado.



Entonces comienza la fiesta. La iniciativa la toma el cliente, que envía una “petición” (*request*) al servidor. La petición es simplemente un texto, dividido en líneas y con una sintaxis muy simple que estudiaremos luego. Vamos a suponer que el cliente ha “pedido” al servidor que le envíe un documento, escrito en lenguaje HTML, con un truco para Delphi.



Si el servidor encuentra el documento y decide que el cliente puede leerlo, envía una respuesta, también formato texto. La respuesta consiste en una cabecera con algunos campos obligatorios y otros opcionales; al final de esa cabecera, se concatena entonces el código fuente del documento HTML solicitado. Si no existe el documento, o si el cliente no tiene los permisos requeridos, se envía también una respuesta en modo texto. El cliente distinguiría esta situación de la anterior gracias a que el servidor incluye en la primera línea de la respuesta un código numérico de estado. Como se ve en el diagrama anterior, 200 significa “viento en popa a toda vela”. Y todos hemos recibido en algún momento el temido código 404: no encontrado.

En el caso más simple, el cliente cerraría la conexión al recibir la respuesta. Analizaría sintácticamente el documento HTML recibido, porque lo más probable es que tenga

que dibujarlo en pantalla. Y puede que encuentre referencias a imágenes situadas en el servidor. Con los navegadores más antiguos, esto significaría repetir el viaje de ida y vuelta tantas veces como imágenes, y otros recursos similares, haya. En los navegadores más recientes se ha introducido una pequeña mejora, que permite que el servidor descargue los recursos asociados a un documento en una misma conexión, para ahorrar tiempo.

## Localización de recursos

Para solicitar un “documento” al servidor, el cliente debe indicar qué es lo que desea mediante una cadena conocida como URL: *Uniform Resource Locator*. Analizaremos ahora la sintaxis de una URL “absoluta”, que no necesita referencias de clase alguna para identificar un recurso en Internet.

La primera parte de una URL tiene el propósito de indicar qué protocolo vamos a utilizar. Consiste simplemente en el nombre del protocolo, seguido de dos puntos. Por ejemplo:

Protocolo	Su uso
<i>http:</i>	La forma más sencilla de HTTP, que estamos examinando.
<i>https:</i>	Protocolo HTTP cifrado. La <i>s</i> significa <i>secure</i> , seguro.
<i>ftp:</i>	<i>File Transfer Protocol</i> : una forma eficiente de enviar y recibir ficheros.
<i>file:</i>	Un fichero local de este ordenador.
<i>javascript:</i>	Pseudo protocolo que llama al intérprete de JavaScript.
<i>mailto:</i>	Pseudo protocolo que inicia un cliente de SMTP (correo electrónico).

Los dos últimos “protocolos” mencionados no son protocolos reales, sino un ejemplo de cómo los navegadores modernos aprovechan la sintaxis de las URLs para otros propósitos.

A continuación debe venir un nombre que identifique el equipo donde se encuentra el recurso que buscamos. Como HTTP suele implementarse sobre el protocolo de transporte TCP/IP, ese nombre simbólico suele “resolverse”, o traducirse, en una dirección IP típica. Tomemos como ejemplo la siguiente URL:

```
http://www.marteens.com/trucos/tricks01.htm
```

El nombre del ordenador o nodo es, como puede imaginar, *www.marteens.com*. ¿Elemental? No tanto. Voy a adelantarme un poco, para mostrar una etiqueta de enlace de HTML:

```
<a href="www.marteens.com">Trucos (incorrectos)</a>
```

El valor del atributo `HREF` está incorrectamente escrito, al menos si lo que pretendíamos era acceder a mi página Web. El problema es que faltan las dos barras inclinadas iniciales, para indicar que se trata de una URL *absoluta*. Sin esas barras, cualquier na-

vegador debe interpretar que *www.marteens.com* debe combinarse con la URL que se está mostrando en ese preciso momento dentro del navegador. Por ahora, sólo hace falta que tenga conciencia de la necesidad de las dos barras; más adelante volveremos a mencionar las URLs relativas y sus reglas de interpretación.

Como decía, el nombre del servidor se extrae a partir de las dos barras inclinadas, hasta llegar a la primera barra simple. Nombres típicos serían:

```
www.borland.com
community.borland.com
localhost
christine
```

He incluido los dos primeros nombres para mostrar un error muy frecuente: las tres uves dobles de las direcciones en Internet no forman parte del nombre de dominio. Cuando una empresa o un particular alquila un nombre de dominio, está realmente alquilando la palabra que va inmediatamente antes del sufijo (el “punto com”, que es el más usual)... acompañada del propio sufijo, por supuesto. De ahí en adelante, el propietario tiene la libertad de crear todos los *subdominios* que le plazca, dentro de su dominio principal. Esos subdominios podrían significar, al nivel físico, ordenadores o direcciones IP diferentes, aunque no es necesario.

*Christine* es el nombre de mi actual servidor Windows 2000, que tiene instalado el Internet Information Server como servidor HTTP. Como *Christine*, dentro de mi red local, es un nombre que se traduce en una dirección IP, puedo utilizarlo sin problema dentro de una URL. Mi estación de trabajo actual también tiene instalado su propio servidor HTTP, y se llama *Naroa*. Es con este servidor local con el que prefiero desarrollar los ejemplos para el libro. Ahora bien, si utilizara el nombre *Naroa* para las URLs de los ejemplos, usted tendría que modificar ese nombre antes de poder hacer algo con ellos. Por eso uso el nombre alternativo *localhost*, que es un convenio universal de TCP/IP para referirse al ordenador local.

Voy a repetir la URL que mencioné antes:

```
http://www.marteens.com/trucos/tricks01.htm
```

A partir del nombre del servidor, el resto de la URL describe un directorio *virtual* dentro de la máquina, y un documento HTML situado dentro de este directorio:

```
trucos/tricks01.htm
```

En este caso, he utilizado un solo subdirectorio, *trucos*, pero podríamos tener más niveles. Quiero enfatizar que se trata de subdirectorios virtuales, que son gestionados mediante el software del servidor HTTP que estemos utilizando, como veremos en breve.

Pero una URL no tiene porqué finalizar con un nombre de fichero físico. El otro tipo frecuente de URL termina mencionando un módulo ejecutable o incluso una DLL. Por ejemplo:

```
http://www.ClassiqueCentral.com/formfill/formfill.exe/
list?filename="fftest.xml"&maxrows=300
```

Esta URL, en realidad, debería estar escrita en una sola línea, pero he tenido que romperla por las limitaciones de espacio. En este caso, estamos haciendo referencia a una aplicación llamada *formfill.exe* que se encuentra dentro de un subdirectorío virtual, también de nombre *formfill*, en un servidor llamado *www.ClassiqueCentral.com*. El resto de la URL, a partir del nombre de aplicación, puede ser interpretado como parámetros que se le pasan a la misma. La idea es que el servidor HTTP debe ejecutar remotamente la aplicación mencionada para que ésta genere una página HTML dinámica, en base a los parámetros que reciba. Este tipo de aplicaciones recibe el nombre de *Common Gateway Interface*, o CGI, y es el área principal del desarrollo para Internet en la que utilizaremos nuestro querido Delphi.

### MÁS SINTAXIS

Para simplificar la explicación, he omitido dos posibilidades: que el servidor no esté escuchando en el puerto 80 estándar. Y que la URL contenga también un nombre de usuario y una contraseña. Esta última sintaxis es muy utilizada con las conexiones FTP.

## Varios tipos de peticiones

¿Vemos más detalles sobre el formato de las peticiones? He aquí una muy sencilla, generada con la ayuda de Internet Explorer 5.5:

```
GET /trucos/trick01.htm HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
Accept-Language: es
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows 98)
Host: localhost
Connection: Keep-Alive
```

El primer identificador de la primera línea es el *comando HTTP*, que identifica el tipo de petición. A continuación se incluye la parte de la URL que sigue tras el nombre del servidor. No se incluye el nombre del servidor y el protocolo porque esos datos ya se han utilizado para establecer la conexión TCP/IP. Por último, al final de la línea se indica la versión de HTTP que comprende el cliente.

El comando HTTP más usual es *GET*, con el que le decimos al servidor que nos envíe un documento; o una página generada dinámicamente, en el caso de las aplicaciones CGI. Pero no todos los clientes necesitan siempre un documento completo. Por ejemplo, la Web está llena de “robots”, o aplicaciones automáticas, que localizan servidores y piden sus páginas principales. Estos robots, en los casos más benignos,

están interesados en conocer el tipo de información que ofrece cada servidor, para lo cual necesitan solamente la cabecera HTML de la página principal, donde normalmente se guardan palabras claves a modo de orientación. Mi página Web contiene en estos momentos la siguiente cabecera:

```
<HEAD>
<title>Calling Dr. Martens...</title>
<meta NAME="keywords"
      CONTENT="Delphi, C++ Builder, Client/Server, SQL,
      Object Oriented Programming, OOP, Kylix, Pascal,
      Databases, COM, DCOM">
<meta NAME="DESCRIPTION"
      CONTENT="All about Delphi & C++ Builder programming,
      tricks and tips.">
</HEAD>
```

Si la petición que envía el robot a mi petición utilizara el comando `GET`, el servidor tendría que enviar el documento completo. Pero si el comando enviado es `HEAD`, y el servidor es capaz de interpretarlo, solamente se respondería con la parte de la cabecera del documento HTML.

## Espiando las peticiones

Antes de seguir con las peticiones, ¿cómo es que he obtenido el texto interno de la petición que he utilizado antes como ejemplo? He hecho trampa, simulando un servidor HTTP dentro de mi ordenador, y espiando las peticiones locales generadas por mi navegador.

Cree una nueva aplicación, y añada en el formulario principal un componente *TRichEdit*, de la página *Win32*, y un *TServerSocket*, de la página *Internet* de la Paleta de Componentes. El *server socket*, o *zócalo servidor*, es quien va a recibir las peticiones, como si fuera un verdadero servidor HTTP, y va a copiar el texto recibido en el editor de texto enriquecido. Para almacenar el texto de las peticiones podríamos haber usado también un simple *TMemo*, pero su capacidad se habría agotado más rápidamente en Windows 98/ME.

Hay que cambiar solamente dos propiedades en el zócalo. Primero, la propiedad *Port*, en la que asignaremos el valor *80*, el puerto estándar utilizado por HTTP. Cuando la aplicación esté ejecutándose, todas las peticiones realizadas al puerto 80 del ordenador serán atendidas por nosotros. Es muy importante, por lo tanto, que si tenemos algún servidor HTTP activo en ese mismo ordenador, lo desactivemos temporalmente antes de ejecutar la aplicación. Podríamos también asignar otro puerto libre a nuestro falso servidor, con la condición de que especificásemos ese puerto en la URL cuando enviemos la petición. La segunda propiedad a modificar en el zócalo es *Active*, que debemos poner a *True*, para que se inicie automáticamente el proceso de escucha en tiempo de ejecución.

Por último, tenemos que interceptar el evento *OnClientRead* del zócalo:



```

procedure TwndPrincipal.ServerSocket1ClientRead(Sender: TObject;
    Socket: TCustomWinSocket);
begin
    RichEdit1.Lines.Add(Socket.ReceiveText);
    Socket.SendText('HTTP/1.0 404 Not found'#13#10);
    Socket.Close;
end;

```

Cuando un cliente se conecta a esta aplicación, *ServerSocket1* crea un objeto interno dentro del propio servidor para que atienda en exclusiva al nuevo cliente. El zócalo original queda entonces libre para seguir a la escucha de nuevas peticiones. Es por ese motivo que el evento anterior tiene un parámetro llamado *Socket*, que no es en modo alguno el *ServerSocket1* original.

*OnClientRead* se dispara cuando se produce un envío de datos por iniciativa del cliente. Al recibir ese aviso, el falso servidor examina el texto enviado desde el cliente echando un vistazo en la propiedad *ReceiveText* del parámetro *Socket* del evento. Esto es posible con HTTP porque todo el contenido intercambiado son cadenas de caracteres legibles. Si se tratase de contenido binario, tendríamos que utilizar otros métodos y propiedades, como *ReceiveBuf* y *ReceiveLength*.

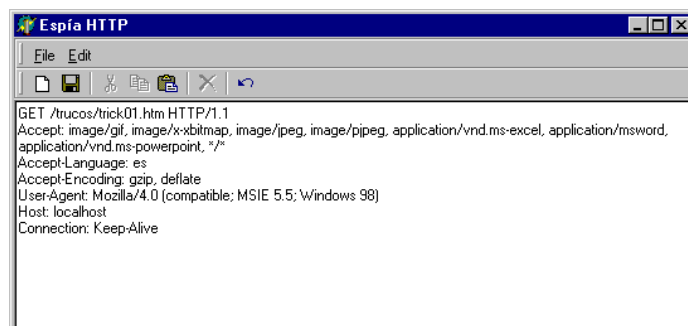
En este caso, el texto enviado desde el cliente se añade al editor de texto enriquecido, y el servidor envía la supuesta respuesta al otro lado de la conexión. Digo “supuesta” porque, pregunte el cliente lo que pregunte, estamos dándole con la puerta en las narices:

```
HTTP/1.0 404 Not found
```

La cadena anterior es la respuesta estándar para advertirle al cliente que el documento que desea no existe. Más adelante veremos otras respuestas posibles.

Para realizar la prueba, ejecute la aplicación y déjela activa. Lance entonces el navegador que esté utilizando regularmente, no importa si Internet Explorer o Navigator. En la barra de direcciones, teclee la siguiente URL:

```
http://localhost/trucos/tricks01.htm
```



Por supuesto, lo que es realmente importante es el nombre del servidor, *localhost* en este ejemplo. Si todo va bien, deberá ver en el editor de texto del servidor el contenido de la petición lanzada desde el navegador.

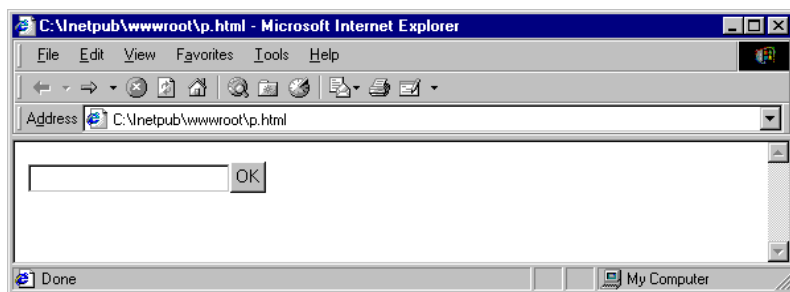
## El comando POST

Después de *GET*, el comando más popular en las peticiones HTTP es *POST*. Esta palabra significa, más o menos, *enviar* en inglés. ¿Qué es lo que se envía, y para qué? Para explicarlo, vamos a tener que adelantarnos un poco en el contenido.

Cree un fichero con extensión *htm* ó *html*, y ábralo desde Delphi con el comando *File|Open*, asegurándose de seleccionar el filtro de extensiones correspondiente. Teclee entonces el siguiente texto dentro del nuevo fichero:

```
<html>
<body>
  <form method=post action="http://localhost/p.htm">
    <input type=text name="edit1"><input type=submit value="OK">
  </form>
</body>
</html>
```

Abra el Explorador de Windows y vaya al directorio donde ha guardado el fichero. Haga doble clic sobre el mismo, para abrirlo con su navegador de Internet predeterminado:



No es el momento de explicar todos los detalles del código HTML empleado, pero es muy sencillo. Hemos creado una página con un formulario para que el usuario teclee algunos datos. Cuando el usuario pulse el botón de aceptar del formulario, esos datos se enviarán a una aplicación CGI, cuya URL se debe indicar en el atributo *ACTION* de la etiqueta *<FORM>*. En este caso, la URL es falsa, y la vamos a interceptar con la ayuda de nuestro espía. Antes de realizar el experimento observe un detalle muy importante: el atributo *METHOD* de *<FORM>* es igual a *POST*. Luego veremos otras posibilidades.

Cuando tecleamos algo y pulsamos el botón dentro del navegador, nuestro servidor espía recibe una petición similar a la siguiente:

```

POST /p.htm HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
Accept-Language: es
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows 98)
Host: localhost
Content-Length: 21
Connection: Keep-Alive
Cache-Control: no-cache

edit1=Prueba+de+datos

```

Las diferencias son pocas, respecto al primer ejemplo del comando `GET`: evidentemente, cambia la primera línea, hay unas cuantas líneas más con parámetros, especialmente la que comienza con *Content-Length*, y al final de la petición hay una línea con el texto que hayamos tecleado en el formulario.

## Formularios

He tenido que adelantar la presentación de los formularios porque no es posible efectuar una petición de tipo `POST` desde la barra de direcciones de un navegador. Pero aprovecharemos esa presentación para explicar también la estrecha relación existentes entre formularios y aplicaciones CGI.

Hay tres formas básicas de enviar una petición HTTP. La primera es la más directa:

### 1 Escribir la URL en la barra de direcciones del navegador.

En realidad, cualquier secuencia de navegación por Internet comienza por una petición enviada de esta forma, o por su simulación. Ya hemos visto que la petición generada utiliza el comando `GET`.

Las otras dos técnicas están vinculadas al uso de código HTML devuelto por otras peticiones; en el ejemplo de la sección anterior, la petición original utilizó el protocolo *file*: para abrir un fichero local. En concreto:

### 2 Pulsar sobre un enlace HTML.

Este ejemplo muestra un enlace simple en HTML:

```
<a href="http://www.marteens.com">Mi página</a>
```

Puede efectuar la prueba para que verifique que la petición generada de esta manera utiliza también el comando `GET`.

### 3 Enviar los datos de un formulario.

Un formulario se define, dentro de un fichero HTML, encerrando entre las etiquetas `<FORM>` y `</FORM>` las definiciones de una serie de *controles HTML*, con los cuales

puede interactuar el usuario final. La etiqueta de apertura de formulario permite una serie de atributos, pero ahora nos interesan dos de ellos: la *acción* y el *método*.

La acción es una URL, y esto que voy a decir es crucial:

*“No tiene sentido que la acción de un formulario sea una URL que haga referencia a un documento estático”*

En otras palabras, la URL de acción de un formulario debe referirse a alguna entidad ejecutable: una aplicación CGI, un módulo ISAPI, un fichero ASP o PHP, o algo similar. Quiero aclarar que *sí* es posible utilizar un documento estático en ese atributo, como hicimos en el ejemplo de la sección anterior. Pero reconozcamos que se trataba de una trampa preparada por nuestro espía.

¿Por qué esta restricción? Porque el objetivo de un formulario es realizar una petición *añadiéndole parámetros*. Los nombres y valores de los parámetros se extraen a partir de los controles encerrados entre las etiquetas de inicio y fin del formulario. Un documento estático, ¿para qué podría querer parámetros? Para nada. En cambio, cualquier generador dinámico de páginas medianamente flexible *necesita* parámetros para su funcionamiento.

¿Cómo se pasan los parámetros? Depende del valor asociado al atributo `METHOD` de `<FORM>`. Los únicos dos valores aceptados son `GET` y `POST`. Si utilizamos `GET`, los parámetros se añadirán a la URL especificada en el atributo `ACTION`. Suponga que habilitamos un formulario para que el usuario teclee su nombre y apellidos y pueda registrarse en una base de datos ubicada en el servidor. Una posible petición enviada desde el formulario sería:

```
http://www.marteens.com/registro.exe?nombre=Ian&apellido=Marteens
```

Pero hay un límite para la longitud de una URL, y es una, aunque no la más importante, de las razones para la existencia del método `POST`. Con éste, la URL de la acción no se modifica, y los parámetros se pasan dentro del propio cuerpo de la petición, como vimos en el ejemplo de la sección anterior.

En el siguiente capítulo, estudiaremos más detalles sobre el uso de formularios en páginas HTML.

## La actualización de una página

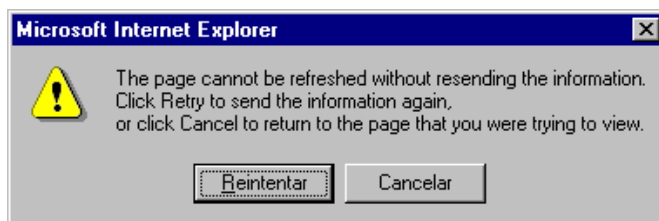
Sin embargo, la razón más importante para la existencia del método `POST` es otra muy diferente. Como sabemos, los navegadores permiten que el usuario *actualice* la página que está viendo en un momento dado. Internet Explorer nos ofrece la tecla de función `F5`, y creo que Navigator utiliza `CTRL+R`. El motivo más frecuente que lleva a un usuario a pulsar una de estas teclas es que una página tarde demasiado en mostrarse,

y es casi imposible de evitar, dada la poca fiabilidad de las actuales conexiones a Internet.

Hagamos un experimento mental con el formulario que permite grabar nuestro nombre y apellido en un servidor remoto. Tecleamos esa información, pulsamos el botón de *Enviar*, y el formulario lanza la siguiente URL al servidor:

`http://www.marteens.com/registro.exe?nombre=Ian&apellido=Marteens`

Suponemos que *registro.exe* sea una aplicación ejecutable con dos funciones: primero extraer los parámetros de la petición y grabar un registro en una base de datos. A continuación, generar una página HTML con un mensaje del tipo: “muchas gracias por cedernos su alma inmortal”. ¿Qué pasaría si un usuario se registrase pero no recibiese a tiempo la respuesta confirmatoria? Pues que perdería los nervios y pulsaría F5 o CTRL+R. Pero eso volvería a enviar los datos del formulario al servidor, y se repetiría la grabación. Para un registro de clientes, esto no sería una catástrofe, pero para la página de pago de una tienda en Internet sí lo sería, porque el importe se cargaría dos veces en la cuenta del cliente.



Eso sucede porque he asumido que el método utilizado por el formulario es `GET`. Si hubiéramos recurrido a `POST`, el navegador nos mostraría el “popular” mensaje de error del cuadro de diálogo mostrado en la imagen anterior. Las páginas obtenidas mediante una petición `POST` deben ser tratadas especialmente por el navegador cuando el usuario pide su actualización. Claro, nada impide que el usuario se ponga cabezón y pulse *Reintentar*. Esta protección es solamente una primera barrera, pero un usuario dispuesto a todo se la puede saltar a la torera.

Resumiendo: el comando `POST` fue diseñado originalmente para ser utilizado en peticiones a recursos ejecutables que implicasen modificaciones de alguna forma en el servidor. Como este tipo de acciones requiere parámetros por lo general, este comando los pasa en el propio cuerpo de la petición, en vez del mecanismo más elemental, que consiste en pasarlos en la propia URL. Por último, los navegadores deben tratar de forma diferente la actualización de una página recibida mediante una petición `POST` y una `GET`.

### ADVERTENCIA

Cuando estemos metidos de lleno en el desarrollo de aplicaciones para Internet, veremos que el método `POST` se utiliza preferentemente con un objetivo espurio: ocultar los parámetros de las peticiones de la vista de los usuarios. Y que, irónicamente,

existen técnicas mejores para evitar acciones duplicadas si el usuario decide actualizar una página desde su navegador.

## Codificación de datos

Aunque volveremos a encontrarnos con ellos, creo importante mencionar ahora la existencia de un tercer atributo en la etiqueta de inicio de un formulario: el atributo `ENCTYPE`, o tipo de codificación. Por codificación se entiende el sistema utilizado para representar parámetros y valores en una petición.

Normalmente, no es necesario preocuparse por la codificación de formularios. Cuando hablo de codificación en este contexto no estoy hablando de hacer ilegible el contenido de los parámetros en aras de la seguridad. Me refiero, en cambio, a que la información que enviamos al servidor no siempre será directamente representable mediante caracteres legibles. Por ejemplo, podemos enviar un fichero al servidor mediante un formulario, quizás con una foto, o con música en algún formato comprimido como MP3. Si a la acción más frecuente de descargar ficheros desde un servidor se le llama en inglés *download*, a esta acción inversa se le conoce como *upload*.

En estos casos, el tipo de codificación requerido se llama *multipart/form-data*. Si quiere tener una idea acerca de cómo funciona este mecanismo, realice el experimento de espiar las peticiones con el siguiente formulario:

```
<form method=post
      enctype="multipart/form-data"
      action="http://localhost/p.htm">
  Nombre del fichero: <input type=file name=fichero><br>
  <input type=submit value="OK">
</form>
```

Tenga en cuenta que el método tiene que ser obligatoriamente `POST`. Observe además que la etiqueta `<INPUT>`, que crea un control de edición HTML, es ahora de tipo `FILE`, y el control que aparece en la página tiene el siguiente aspecto “compuesto”:



Si lleva a cabo la prueba con el espía verá que, aparentemente, la aplicación que simula el funcionamiento de un servidor HTTP no recibe correctamente toda la información. Se debe a que siempre cerramos la conexión después de recibir el primer paquete de datos desde el cliente. Pero en este nuevo ejemplo el cliente puede enviar varios paquetes con datos; en concreto, el contenido del fichero que elijamos viaja en un segundo envío.

Para dar una solución real a este problema, tendríamos que analizar cada petición recibida, para detectar aquellas que utilicen la codificación *multipart* y no cerrar inmediatamente el zócalo de conexión. Pero le sugiero hacer algo más sencillo y que nos valdrá por el momento: eliminar la instrucción de cierre del zócalo.

```

procedure TwndPrincipal.ServerSocket1ClientRead(Sender: TObject;
    Socket: TCustomWinSocket);
begin
    RichEdit1.Lines.Add(Socket.ReceiveText);
end;

```

Ahora tendremos un problema en el cliente, porque el navegador se quedará esperando indefinidamente la respuesta. Pero no es algo que me quite el sueño, porque lo importante es obtener y reunir todos los fragmentos de la petición:

```

POST /p.htm HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
Accept-Language: es
Content-Type: multipart/form-data;
    boundary=-----7d1f02d64
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows 98)
Host: localhost
Content-Length: 314
Connection: Keep-Alive
Cache-Control: no-cache

-----7d1f02d64
Content-Disposition: form-data; name="fichero";
filename="C:\Marteens\Classique\Bin\Mozart.ini"
Content-Type: application/octet-stream

[Mozart]
ComputerName=http://localhost/scripts/httpsrvr.dll
ServerName=Debussy.Database

-----7d1f02d64--

```

Observe que el navegador define, en la cabecera de la petición, un separador para las partes más simples que conforman la petición completa. Generé esta petición enviando un pequeño fichero llamado *mozart.ini* desde mi ordenador. Como se trata de un fichero de texto, su contenido puede identificarse inmediatamente. He tenido que retocar un poco el texto, para no tener que cortar las líneas en sitios arbitrarios.

Me he metido en este berenjenal del formato de codificación por dos motivos:

- 1 Delphi 5 no ofrecía soporte directo para la codificación *multipart*. Para poder implementar el envío de ficheros (*upload*) había que programar bastante.
- 2 Personal Web Server, el servidor HTTP que suele emplearse para desarrollar aplicaciones de Internet en ordenadores con Windows 98/ME, tiene problemas para manejar la carga de ficheros de más de 4KB de tamaño. Debe tener en cuenta esta limitación si desea trabajar con este recurso.

#### NOTA

El formato de codificación por omisión se llama *application/x-www-form-urlencoded*. Es un nombre que solamente podría haber sido inventado por un alemán o por un burocrata. El tercer formato actualmente reconocido es *text/plain*, y es utilizado por algu-

nos navegadores para enviar el contenido de un formulario como un mensaje de correo electrónico.

## Las cabeceras HTTP

Como habrá observado, a partir de la segunda línea de una petición se incluye una serie de “parámetros” comunes. El verdadero nombre para esta información es el de *cabeceras HTTP*, o *HTTP headers*. Algunos de los parámetros de cabeceras se repetirán también en el formato de las respuestas HTTP, pero otros son exclusivos de las peticiones o de las respuestas.

Mencionaré a continuación algunas de las cabeceras de peticiones con significados no demasiado evidentes:

Cabecera	Significado
<i>Accept</i>	Los tipos de documentos que el navegador “comprende”
<i>Accept-Encoding</i>	El servidor podría comprimir el documento antes de enviarlo
<i>User-Agent</i>	La “marca” del navegador: si es Internet Explorer, Navigator, etc
<i>Connection</i>	<i>Keep-Alive</i> optimiza el envío de imágenes asociadas a una página
<i>Referer</i>	La posible página desde la cual se ha generado esta petición

La cabecera *Referer* solamente se genera cuando, desde una página obtenida mediante *http* o *https*, pulsamos sobre un enlace o enviamos el contenido de un formulario. Así podemos distinguir esos dos casos, de las peticiones generadas al teclear una URL en la barra de direcciones del navegador.

## El resultado de una petición

Para terminar nuestro rápido repaso sobre el protocolo HTTP, veremos qué tipos de respuestas puede provocar una petición. Lo más normal es que el servidor devuelva el documento pedido, sea virtual o no, o que nos diga que no tiene ni la más remota idea sobre lo que estamos pidiendo. En el primero de los casos, el código de respuesta es el 200, y en el segundo, el odiado 404. El valor numérico se incluye en la primera línea de la respuesta, y a continuación debe ir un texto con la descripción del error, como en estos dos ejemplos:

```
HTTP/1.1 200 OK
HTTP/1.1 404 Not found
```

Hay más valores, sin embargo. Los más populares son:

- 1 Si se produce un error al procesar la petición y el servidor puede recuperarse lo suficiente para enviar una respuesta, ésta debería ser *500 Internal Server Error*. En la práctica, la respuesta 500 se produce al ejecutar aplicaciones CGI o módulos ISAPI que terminan con una excepción.



- 2 El servidor puede redirigir una petición a otra URL, quizás dentro del propio dominio, o fuera de él. La respuesta sería *301 Moved Permanently*, y el navegador debería reaccionar realizando la petición a la nueva URL, que se envía dentro de una de las cabeceras de la respuesta. Veremos que la respuesta 301 tendrá aplicaciones insospechadas en el desarrollo para Internet.
- 3 Si el recurso solicitado requiere autorización, normalmente especificando el usuario y su contraseña, el servidor debe enviar *401 Unauthorized*. El cliente que recibe la respuesta 401 debería mostrar al usuario un cuadro de diálogo para que se identifique y repita la petición.
- 4 Si queremos denegar definitivamente el acceso al recurso, podemos responder con un *403 Forbidden*. En ese caso, el navegador no debería insistir.

Cuando la respuesta es 200, el contenido del documento se añade a continuación de los campos de la cabecera. El campo *Content-Type* indica el tipo del documento enviado, y *Content-Length* su longitud.

¿Existe alguna forma sencilla de espiar las respuestas completas de un servidor HTTP, al igual que hicimos con las peticiones? Sí, y no es mirar el código fuente de las páginas recibidas, porque así no vemos la cabecera. Hay que compilar y ejecutar uno de los ejemplos que acompañan a Delphi, que se encuentra en el siguiente subdirectorio:

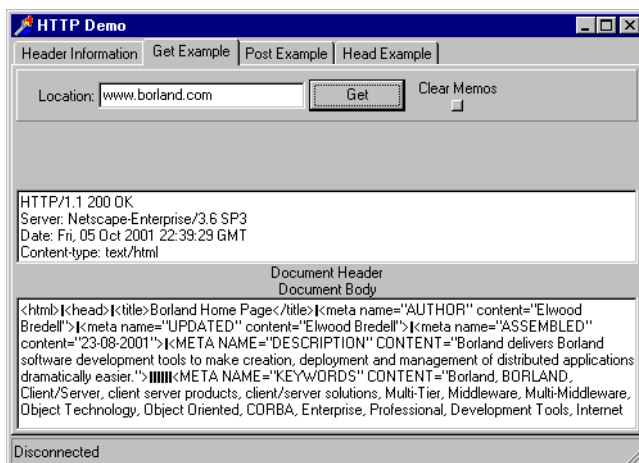
```
Demos\FastNet\Http\HttpDemo.dpr
```

Debo aclarar que el ejemplo no ha sido programado por Borland, o eso quiero creer, porque su interfaz es espantosa. ¿Ve esa protuberancia debajo de un texto que dice *Clear memos*? Aunque no lo parezca, se trata de un botón. Si quiere divertirse, cambie el tamaño de la ventana<sup>31</sup>.

En la siguiente imagen se muestra el resultado de una petición a la página principal de Borland. Observe la cabecera, en el panel superior, y los parámetros recibidos con la respuesta. En particular, note que la página Web de Borland utiliza como motor el servidor HTTP de Netscape.

---

<sup>31</sup> Una de mis actuales aficiones es la música por ordenador. Hay un famoso y prestigioso programa secuenciador alemán, que es el más utilizado en Europa. Lo menciono porque tiene la peor interfaz de usuario que conozco. No es que funcione mal, pero se salta todos los convenios de interfaz a los que estamos acostumbrados. Claro, es una aplicación con versiones para Windows y MAC...



## Servidores HTTP

Antes de empezar a desarrollar aplicaciones, es imprescindible que conozcamos unos cuantos detalles sobre la configuración y funcionamiento de los servidores HTTP con los que vamos a trabajar. Delphi nos permite desarrollar varios tipos de aplicaciones o de módulos para aplicaciones: por ejemplo, podemos crear objetos ActiveX con características especiales para ser utilizados desde páginas ASP. Y solamente podemos utilizar páginas ASP con los servidores HTTP de Microsoft. Pero el tipo más común de proyecto para Internet creado con Delphi son las aplicaciones CGI, y sus variantes más eficientes basadas en DLLs. Para este modelo concreto de desarrollo, podemos utilizar cualquiera de los siguientes tipos de servidores:

- 1 Internet Information Server, de Microsoft
- 2 Personal Web Server, también de Microsoft
- 3 Netscape Server
- 4 El servidor HTTP de O'Reilly & Associates
- 5 Apache para Windows (sólo a partir de Delphi 6)

Además de estos, podríamos también usar cualquier otro servidor que permita el uso de aplicaciones CGI. Lo que sí debemos exigir es que el servidor se ejecute bajo alguna versión de Windows... porque el propio Delphi no genera código para otros sistemas operativos.

Hasta aquí los hechos. ¿Cuál es el *mejor* de estos servidores? Es difícil dar una respuesta objetiva, y no sería honesto presentarla como verdad absoluta. Pero sí puedo responder a una variante de esa pregunta: ¿cuál servidor *recomiendo*? Sé que mi respuesta no va a gustar a todos, pero de acuerdo con mi experiencia de desarrollo *real*, mi favorito es Internet Information Server (IIS) versión 5, ejecutándose sobre Windows 2000 Server. Es rápido, y cada nueva versión introduce mejoras en este

sentido. Su estrecha integración con el sistema operativo, aunque sea tramposa, influye mucho en este aspecto.

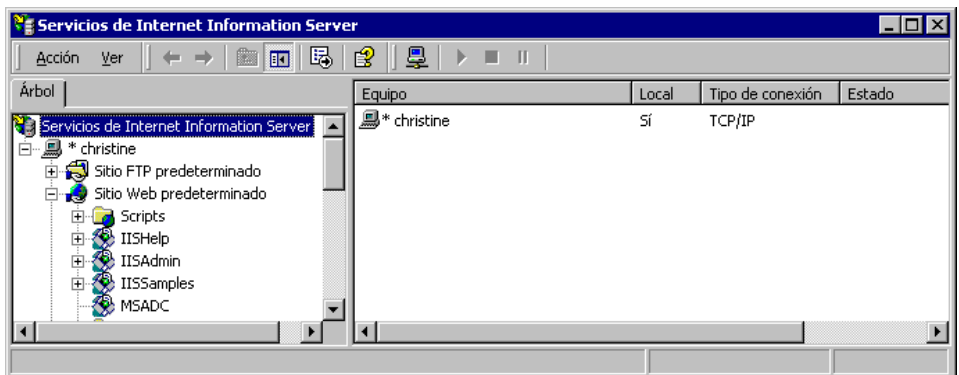
En manos de un técnico calificado, IIS es seguro: el incidente del virus RedCode afectó principalmente a sistemas mal configurados. La alerta se produjo con tiempo suficiente para que los responsables de servidores Web aplicasen los parches y cerraran las compuertas, pero conozco a varios administradores que sólo se preocuparon cuando ya era demasiado tarde.

¿No está de acuerdo con mi recomendación? Tiene todo su derecho. Me he limitado a dar mi opinión, y para este tipo de asuntos suelo ser bastante pragmático.

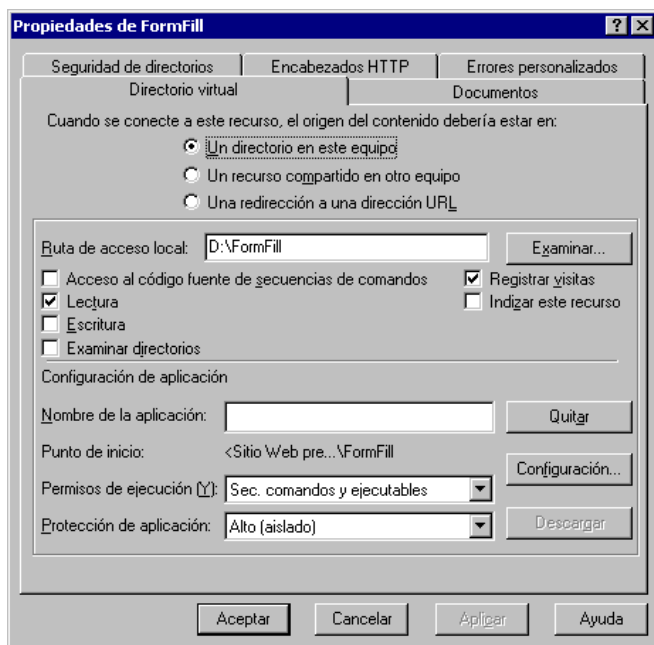
## Internet Information Server

La versión 5 de IIS se instala como un componente más de Windows 2000 Server. Junto con el servicio principal de HTTP, también se instala una amplia colección de herramientas y servicios adicionales. Cualquier persona con un mínimo de experiencia en seguridad le dirá lo mismo que yo: no instale servicios que no vaya a utilizar, y mucho menos permita que se ejecuten. Por ejemplo, si el servidor está ubicado en su propia oficina, prescindir de FTP. El virus RedCode pudo propagarse gracias a un servicio opcional de W2K, el de indexación de ficheros.

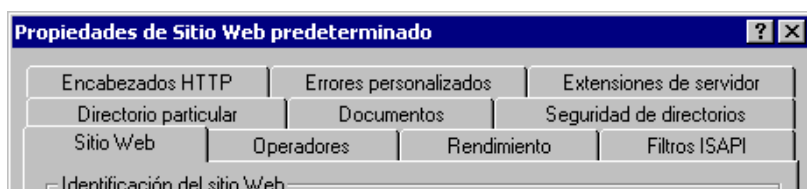
El servicio de Internet se administra desde la Microsoft Management Console, y la forma más sencilla de acceder a ésta es ejecutar, desde el menú de inicio de W2K Server, el comando *Herramientas Administrativas* | *Administrador de Servicios de Internet*:



IIS 5 permite administrar varios dominios o sitios de Internet ubicados en el mismo equipo. Para simplificar, supondremos que hay un solo sitio funcionando, el predeterminado. Como muestra la imagen anterior, a partir de la raíz del servidor se expande un árbol de directorios virtuales. Podemos situarnos en cualquiera de sus nodos y configurar sus propiedades con la ayuda del menú local:



El nodo principal, naturalmente, ofrece más opciones de configuración que los restantes. La siguiente imagen muestra todas las pestañas que tiene el cuadro de propiedades del nodo principal:



La página más importante para un nodo es la titulada *Directorio particular*. En ella se configura el directorio real al que está vinculado el nodo y los permisos de acceso HTTP. He mencionado los permisos con la coletilla del protocolo porque debemos combinarlos con los permisos NTFS existentes sobre el directorio físico. Los permisos asignables básicos son:

- 1 *Lectura*: Hay que habilitarlo si queremos que se puedan cargar documentos estáticos o imágenes situados en el directorio virtual. Más adelante aclararé un par de errores relacionados con este permiso.
- 2 *Escritura*: Ciertas extensiones de HTTP permiten escribir información desde un cliente remoto. Lo más sensato es dejarlo inactivo.
- 3 *Examinar directorios*: Si está activo, el usuario podría teclear el nombre del directorio virtual desde su navegador, y vería la lista de documentos ubicados en su interior. No es buena idea dejarlo activo.

- 4 *Registrar visitas*: Debe activarlo, para auditar los accesos al recurso y detectar posibles ataques.
- 5 *Indizar este recurso*: Activa el servicio de indexación de ficheros para el directorio. Sí, es el mismo servicio que abrió la puerta al maldito Código Rojo.
- 6 *Acceso al código fuente, etcétera*: Si está activo, permitiría ver el código fuente original de las páginas ASP. Mala idea.

Cuando el sitio Web utiliza únicamente páginas estáticas, que es lo más frecuente, sólo es necesario conceder el permiso de lectura. Para sitios basados en aplicaciones CGI/ISAPI, sin embargo, es preferible quitar el permiso de lectura sobre el directorio en el que se ubicará el ejecutable o la DLL. ¿Por qué? Pues porque es habitual que en ese mismo directorio se sitúen ficheros de configuración, plantillas HTML y toda una serie de recursos que, si el usuario pudiera leerlos, le ofrecerían *demasiada* información sobre la implementación de la página. Claro, los sitios basados en aplicaciones necesitan también imágenes y, en ocasiones, algunos documentos HTML estáticos. Pero esos documentos e imágenes estarían mejor en un directorio separado y, esta vez sí, con permiso de lectura.

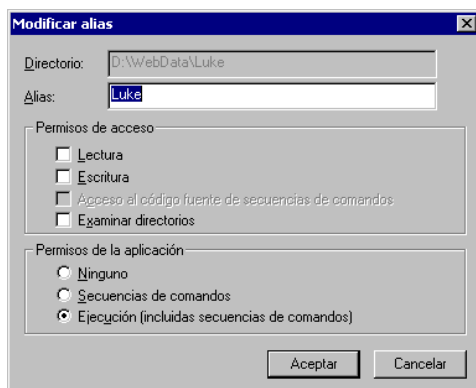
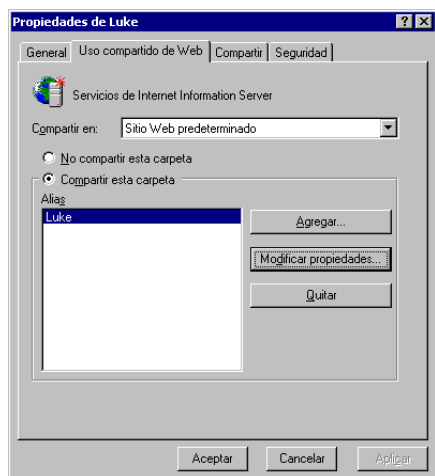
Los permisos mencionados deben combinarse con los permisos de ejecución, para los que existen tres niveles: no permitir ejecución remota, permitir solamente la ejecución de aplicaciones ASP y finalmente, barra libre, permitir aplicaciones CGI. El error más común al configurar un servidor por primera vez para una aplicación CGI es olvidar la activación del permiso de ejecución. Cuando el usuario intenta acceder al sitio, el navegador le propone *descargar* el fichero ejecutable, en vez de ejecutarlo remotamente y enviar el código HTML generado. Es una situación muy fácil de detectar y corregir.

## Directorios virtuales

Para crear un directorio virtual, podemos ejecutar un asistente desde la MMC. El asistente le preguntará por el directorio físico asociado, le pedirá el nombre que desea para el directorio virtual, y los permisos que quiere conceder.

Es más sencillo, al menos para mí, crear y configurar esos directorios desde el propio Explorador de Windows. Si el servicio de Internet está funcionando y activa el diálogo de propiedades de un directorio utilizando el Explorador, verá una página titulada *Uso compartido de Web*.

Si decide publicar la carpeta en Internet, puede configurar el nombre del directorio virtual y los permisos pulsando el botón *Agregar*, y posteriormente mediante *Modificar propiedades*. El diálogo de configuración se muestra a la derecha, en la imagen anterior. Comprobará que la administración de permisos se hace más fácil con este método.



Lo que voy a explicar ahora es importante. Supongamos que necesitamos configurar un directorio virtual para una aplicación CGI. Queremos un directorio con permisos de ejecución, aunque sin permisos de lectura. Pero como necesitamos imágenes estáticas, queremos colocarlas en otro directorio, con permiso de lectura. Necesitamos entonces dos directorios virtuales diferentes (no he incluido el nombre del servidor):

```
/miaplicacion
/miaplicacion/imagenes
```

Usted ha copiado el fichero ejecutable en un directorio físico que, para variar, suponemos que se llama:

```
c:\archivos de programa\mi empresa\miaplicacion
```

Lo que quiero explicarle es que *no es necesario* que el directorio físico de las imágenes se llame:

```
c:\archivos de programa\mi empresa\miaplicacion\imagenes
```

No es que sea incorrecto: podríamos hacerlo así, siempre que después configurásemos correctamente los permisos del subdirectorio, que normalmente hereda los permisos de su directorio padre. Pero también podríamos ubicar las imágenes en un directorio que no guarde relación alguna con el de la aplicación, como:

```
d:\imagenes
```

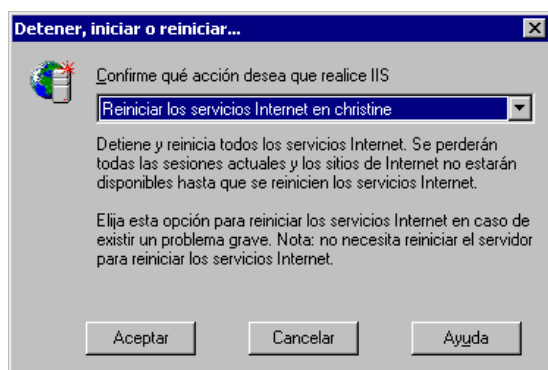
Resumiendo: el que un directorio virtual sea el padre de otro, no implica necesariamente que exista la misma relación entre los correspondientes directorios físicos.

## Módulos ISAPI

Los módulos ISAPI son DLLs que pueden ser invocadas desde el IIS para que generen contenido HTML dinámico. Funcionalmente son equivalentes a las aplicaciones CGI, excepto en pequeños detalles, pero son infinitamente más eficientes.

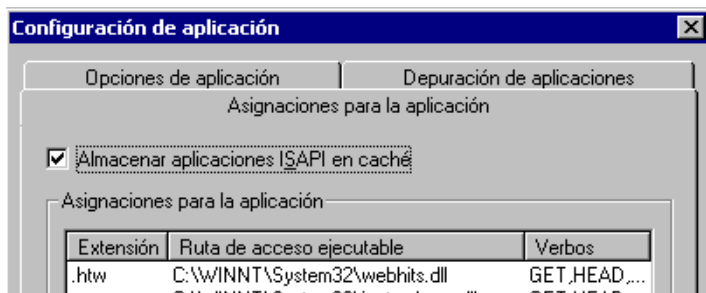
Hay un par de detalles que debe conocer sobre la configuración de Internet Information Server para aplicaciones ISAPI:

- 1 Es cierto que los módulos ISAPI eran peligrosos, en versiones anteriores de IIS. El servidor cargaba las DLLs en su mismo espacio de proceso, y una aplicación mal programada podía corromper la memoria y detener el funcionamiento del servidor. Pero la situación actual es muy diferente: IIS carga esos módulos en un espacio de memoria protegido especial, cortesía del sistema operativo. La comunicación entre el proceso principal y la DLL sigue siendo muy eficiente, pero es más difícil corromper la memoria del proceso. Si la aplicación ISAPI se cuelga, el resto de los procesos puede seguir funcionando sin problemas.
- 2 Cuando una DLL de una aplicación ISAPI se carga en el espacio de proceso de IIS, permanece en ese estado por los siglos de los siglos. Antes, si la DLL se detenía, o si queríamos sustituir la DLL por una nueva versión, era necesario detener completamente el servicio de Internet, y en ocasiones, incluso reiniciar la máquina. Todos sabemos cuánto tarda en reiniciarse un servidor de Windows 2000/NT; si el ordenador está dando servicio a una Web con mucho tráfico, el reinicio es algo a lo que solamente debemos recurrir en casos extremos. Afortunadamente, con IIS se introduce la posibilidad de reiniciar rápidamente los servicios de Internet. Esta posibilidad se ofrece desde el menú local del árbol de directorios virtuales de IIS.



Si estamos desarrollando aplicaciones ISAPI y probándolas directamente sobre un Windows 2000 con IIS, sería incómodo, de todos modos, tener que reiniciar el servicio cada vez que necesitemos recompilar el proyecto. En ese caso, podemos utilizar una opción más drástica: hacer que los módulos ISAPI se comporten como las apli-

caciones CGI, y se carguen y descarguen cada vez que los utilizemos. Para activar esta opción tenemos que ejecutar el diálogo de propiedades del nodo principal del servicio, y en la página *Directorio particular* pulsar sobre el botón *Configuración*, en el panel inferior de la página. En el diálogo que entonces aparece, hay que desactivar la casilla *Almacenar aplicaciones ISAPI en caché*.



Como podrá imaginar, esta opción ralentiza un poco el funcionamiento de las aplicaciones ISAPI, y solamente debería utilizarse en un ordenador de desarrollo, no en un servidor público.

## Personal Web Server

No hace falta tener un Windows 2000 o NT a mano para disponer de un servidor HTTP. He desarrollado la mayoría de mis aplicaciones para Internet sobre un ordenador con Windows 98, utilizando Personal Web Server (PSW) como servidor local de Internet. Esta pequeña herramienta soporta módulos ISAPI, al igual que su hermana mayor, e incluso permite probar el funcionamiento de aplicaciones ASP.

Debo advertirle que Personal Web Server es una especie en extinción. La versión que suelo instalar es la que viene en el CD de instalación de Windows 98 SE. El PWS no se instala por omisión, sino que hay que buscarlo en el directorio *Addons* del CD. Hay un *bug* insidioso en esa instalación, que se manifiesta en ocasiones al terminar: el cliente MTS no funciona correctamente. En la página Web de Microsoft hay un parche para sustituir el fichero *mtssetup.dll*. La sustitución debe hacerse *antes* de instalar el producto, no después. He incluido este fichero corregido en el CD que acompaña al libro.

Hay menos opciones en PWS que en IIS. El siguiente cuadro de diálogo permite administrar los directorios virtuales de forma centralizada. Pero también podemos utilizar el Explorador de Windows para esa tarea:





Un inconveniente importante: no conozco la forma de desactivar la caché de módulos ISAPI con la versión “moderna” de PWS para Windows 98. En cambio, la versión más primitiva, para Windows 95, sí permitía la desactivación cambiando el valor de una entrada en el Registro de Windows:

```
[HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\W3Svc\
Parameters]
CacheExtensions="00"
```

Desgraciadamente, PWS para Windows 95 no soporta aplicaciones ASP.

### ADVERTENCIA

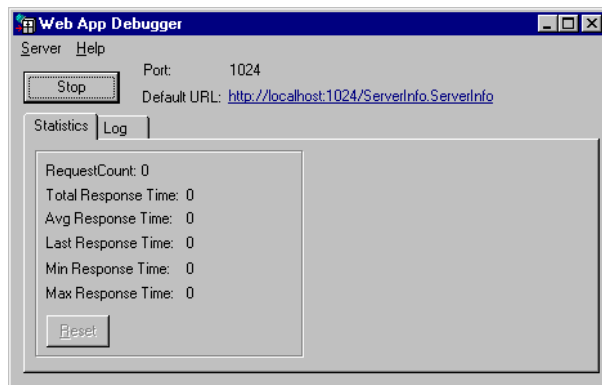
FrontPage 98 viene con una versión “a la medida” de PWS. No se le ocurra utilizar esa versión, porque es más incómoda para configurarla y para trabajar con ella.

## Web App Debugger

Delphi 6 añade otra arma al impresionante arsenal de servidores HTTP: el Web App Debugger. No es un servidor convencional, sin embargo. En primer lugar, recibe las peticiones a través del puerto 1024, al menos por omisión. En segundo, interpreta las URLs que recibe de forma muy particular, porque las considera como *identificadores de programas*, en el sentido que le da COM a la frase. Por lo tanto, no puede utilizarse como servidor para un sistema real en explotación; ha sido diseñado solamente para permitir la depuración de aplicaciones para Internet dentro del Entorno de Desarrollo de Delphi.

En el momento adecuado, veremos cómo preparar un proyecto para su depuración. Por ahora, basta saber que para iniciar este servidor podemos ejecutar el comando de menú *Tools | Web Debugger*, desde el Entorno de Desarrollo. Pero también se puede crear un acceso directo al siguiente fichero:

```
C:\Archivos de programa\Borland\Delphi6\Bin\webappdbg.exe
```



Es muy importante tener en cuenta que Borland incluye una aplicación sencilla que enumera todas las aplicaciones registradas para ser usadas con el Web App Debugger. La aplicación se registra durante la instalación de Delphi 6 con el identificador de programa *ServerInfo.ServerInfo*, y corresponde al siguiente fichero ejecutable:

```
C:\Archivos de programa\Borland\Delphi6\Bin\ServerInfo.exe
```

# Introducción a HTML

**E**XISTEN TRES TIPOS DE PERSONAS: las que piensan principalmente el hemisferio cerebral izquierdo, las que utilizan el derecho, y las que no se molestan en usar ninguno de los dos. O al menos eso dicen los psicólogos. Si le seguimos la corriente a los loqueros profesionales, aquellos que tienen más desarrollada la mitad izquierda tienden a razonar en términos lógicos. Por el contrario, en quienes predomina la mitad derecha suelen tener dotes artísticas. Si eso fuese verdad, yo, con toda probabilidad, tendría mi hemisferio derecho lleno de serrín.

Así que no se asuste: no pretendo enseñarle a diseñar páginas bonitas; eso lo dejo para los artistas. Mi intención es que se familiarice con la mayor cantidad posible de detalles de la sintaxis de HTML. Las aplicaciones CGI generan código HTML dinámicamente, y la clave para triunfar es un buen conocimiento de ese lenguaje.

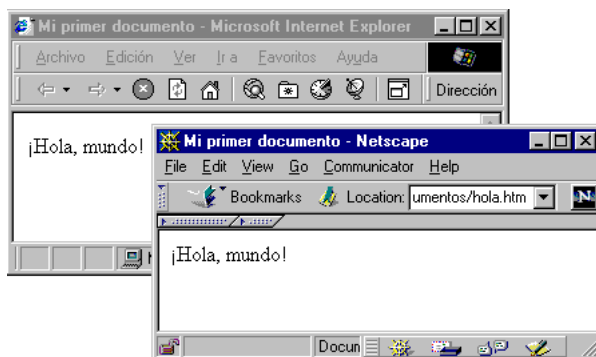
## Aprenda HTML en 14 segundos

No hace falta decirlo por ser de sobra conocido: las páginas Web están escritas en el lenguaje HTML, bastante fácil de aprender sobre todo con una buena referencia al alcance de la mano. HTML sirve para mostrar información, textual y gráfica, con un formato similar al que puede ofrecer un procesador de textos sencillo, pero además permite la navegación de página en página y un modelo simple de interacción con el servidor Web.

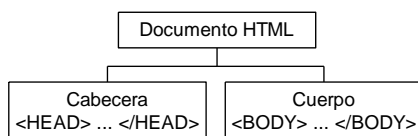
Un documento HTML cercano al mínimo sería como el siguiente:

```
<html>
<head><title>Mi primer documento</title></head>
<body>¡Hola, mundo!</body>
</html>
```

Este documento mostraría el mensaje *¡Hola, mundo!* en la parte de contenido del navegador, mientras que la barra de título del mismo se cambiaría a *Mi primer documento*.



Incluso en una página tan simple se puede distinguir claramente una estructura:



Antes he escrito “cercano al mínimo”, porque HTML es realmente permisivo, y la mayoría de los navegadores actuales permiten omitir muchas de las construcciones sintácticas que hemos mostrado. Para saludar al planeta nos hubiese bastado con escribir sencillamente la frase de saludo.

## Etiquetas a granel

La estructura o sintaxis de HTML viene determinada por lexemas (o grupos de caracteres) conocidos como *etiquetas*, en inglés *tags*. Una etiqueta comienza y termina con paréntesis angulares <>, y en su interior debemos incluir al menos el nombre que identifica a la etiqueta, en primer lugar. Luego es posible añadir atributos para algunas de ellas:

```

<html>
<table width="100%" border=0>

```

Da lo mismo si escribimos el contenido de una etiqueta con mayúsculas o minúsculas. En cuanto al valor de los atributos, es costumbre general encerrarlos entre dobles comillas. Sin embargo, las comillas pueden omitirse cuando no hay posibilidad de error (por ejemplo, espacios en blancos dentro del valor), y esa será la política que seguiremos en este libro. ¿Para qué aumentar inútilmente el tamaño de un fichero HTML, con lo lenta que son las comunicaciones actualmente?

Cuando memorice el significado de una etiqueta, es bueno que la clasifique en uno de dos grupos: aquellas que van por pares y las que se comportan como lobos solitarios. Por ejemplo, para controlar el tipo de letra se utilizan etiquetas que indican el inicio y el fin del cambio de apariencia:

El `<font color=red>mundo</font>` está que arde.

Como podemos observar, la etiqueta de cierre es idéntica a la de apertura, pero precediendo el nombre de la misma por una barra inclinada y eliminando cualquier posible atributo.

Por contraste, la etiqueta que traza una línea horizontal no tiene pareja:

```
Un mundo dividido...
<hr>
... por una línea horizontal.
```

Algunas pocas etiquetas pueden disfrutar de ambos mundos, como sucede con la etiqueta de inicio de párrafos. Estos son dos párrafos separados ortodoxamente:

```
<p>El primer párrafo.</p>
<p>El segundo.</p>
```

Estos otros utilizan un estilo más permisivo:

```
<p>El primer párrafo.
<p>El segundo.
```

Aunque no existen reglas exactas para determinar cuándo podemos omitir el cierre de determinada etiqueta (aparte de echar mano a un manual), una sencilla indicación suele bastar. La etiqueta en cuestión, ¿puede anidarse de forma directa? ¿Tiene sentido “un párrafo dentro de otro párrafo”? Si la respuesta es no, es probable que podamos omitir la etiqueta final.

De todos modos, mi experiencia con los diversos navegadores que surcan el ciberespacio me aconseja no omitir las etiquetas opcionales de cierra, pues algunos de ellos interpretan el texto resultante de las más bizarras e inesperadas maneras.

## Formato básico

Para comprender cómo funcionan las etiquetas de formato básico es útil comparar una página HTML con un documento editado en un procesador de texto. La principal diferencia: en una página Web no existen cambios de página.

El formato más elemental de una página Web consiste en una serie de párrafos consecutivos. Ya hemos visto la etiqueta `<P>`, que delimita párrafos. Ahora presentamos su principal atributo, *align*, que determina el modo de alineación:

```
<p align=right>El que da pan a perro ajeno,
pierde el pan y pierde el perro</p>
<p align=justify>La sabia reflexión que encabeza este artículo
pertenece al insigne filósofo griego Diógenes, representante
de la escuela de los cínicos.</p>
```



Esta situación será tristemente característica: sólo podremos controlar importantes características visuales de una página Web recurriendo a ampliaciones del estándar HTML, en este caso las tan nombradas hojas de estilo en cascada.

## Cambio de aspecto físico o semántico

La siguiente característica en orden de importancia que nos ofrecen tanto los procesadores de texto como HTML es el cambio de aspecto del texto, generalmente asociado al tipo de letra. Sin embargo, en HTML encontramos dos grandes grupos de etiquetas de cambio de aspecto: las que dan una indicación sobre el contenido del texto que modifican, dejando que sea el navegador el responsable de determinar el aspecto exacto, y las que indican directamente y sin ambigüedad los cambios de apariencia que necesitamos.

En el primer grupo de etiquetas podemos clasificar a las siguientes:

Etiqueta	Significado
<H1>, <H2> ...	Encabezamientos
<EM>	Texto enfatizado (casi siempre en cursivas)
<STRONG>	Enfasis aún mayor (casi siempre en negritas)
<CITE>	Una cita textual (casi siempre en cursivas)
<CODE>	Texto de programas (casi siempre letras no proporcionales)

Cuando encerramos un título de sección dentro de una página Web entre etiquetas <H2> y </H2> sabemos que el navegador utilizará algún tipo de letra más destacada que la utilizada para los párrafos. Igualmente sabemos que con <H1> lograríamos un efecto más marcado. Pero no tenemos seguridad alguna del tipo de letra exacto que se aplicará. No se trata de una indecisión perversa por parte de los inventores del HTML: a una página Web, en principio, pueden conectarse los más diversos navegadores ejecutándose sobre sistemas operativos variopintos. Puede incluso que alguien disfrute de nuestras páginas utilizando un navegador basado en el modo de caracteres. Es posible, aunque improbable.

De todos modos, existen etiquetas que indican el aspecto *físico* del texto que modifican. Quizás la más importante es la etiqueta <FONT>, que cambia el tipo de letra del texto que delimita. <FONT> requiere una etiqueta de cierre, y podemos anidar varios cambios de letras. Hay tres atributos aplicables al tipo de letra:

- *FACE*

Es el nombre del tipo de letra. Es una extensión muy útil ofrecida tanto por Netscape como por Microsoft pues HTML estándar, al pensar misericordiosamente en los usuarios de sistemas operativos esotéricos, no puede garantizar que determinado tipo de letra esté presente en cualquier plataforma. Podemos listar varios tipos de letra separados por comas: <FONT FACE="verdana,arial">. Si uno de los tipos no está presente, se ensaya el siguiente.

- **SIZE**

El tamaño del tipo de letra. Aquí también HTML se muestra generoso, y define una escala de tipos virtuales que va desde el 1 al 7. Mientras mayor el número, mayor el tamaño. También pueden utilizarse tamaños relativos al contexto en el que aparece la etiqueta: `<FONT SIZE=+1>`.

- **COLOR**

Existen varios sistemas para identificar el color. Se puede indicar un color de la paleta RGB utilizando esta notación: `<FONT COLOR=#ffff00>`. También existen 16 nombres de color estándar, en inglés (*black, navy, red, silver*).

### ADVERTENCIA

Uno de los mayores retos a los que se enfrenta un programador de Internet es la complejidad interna de las páginas generadas por la mayoría de los diseñadores. Casi siempre estos tratan a la página como si precisamente se tratase de un documento en un procesador de textos, y aplican generosamente retoques al formato sin mirar el contenido HTML resultante. En consecuencia, una página típica contiene montones de etiquetas `<FONT>` dispersas por toda su extensión, en algunos casos incluso redundantes o inútiles.

Una forma de luchar con este problema es obligar al diseñador a utilizar *Hojas de Estilo en Cascada*... pero claro, esto ralentiza la velocidad de trabajo del diseñador, y se produce el inevitable conflicto de intereses.

Para completar el trabajo de `<FONT>` existen otras etiquetas que influyen en el aspecto físico del texto. Algunas de ellas son:

Etiqueta	Propósito
<code>&lt;B&gt; ... &lt;/B&gt;</code>	Texto en <b>negritas</b>
<code>&lt;I&gt; ... &lt;/I&gt;</code>	Texto en <i>cursivas</i>
<code>&lt;U&gt; ... &lt;/U&gt;</code>	Texto <u>subrayado</u>
<code>&lt;STRIKE&gt; ... &lt;/STRIKE&gt;</code>	Texto <del>tachado</del>
<code>&lt;SUB&gt; ... &lt;/SUB&gt;</code>	Subíndices
<code>&lt;SUP&gt; ... &lt;/SUP&gt;</code>	Superíndices

## Enlaces

Por definición, un “hipertexto” es “híper” precisamente porque nos permite navegar de documento en documento mediante marcas convenientemente situadas. Por lo tanto, una de las etiquetas más importantes es el *ancla* (*anchor*), que permite dicha operación. Por ejemplo:

```
<HTML>
<HEAD><TITLE>Mi primer documento</TITLE></HEAD>
<BODY>
  <H1>¡Hola, mundo!</H1>
  <HR>
  <P>Visite la página de
```



```

    <A HREF="http://www.intsight.com">Intuitive Sight</A>
    para más información sobre cursos de Delphi.</P>
</BODY>
</HTML>

```

En este ejemplo la dirección del enlace es una URL completa, que indica toda la ruta de búsqueda desde el principio.

También podemos incorporar enlaces relativos, sin utilizar el nombre del protocolo ni el del dominio. Aparentemente, de este modo resolvemos un problema común: las páginas Web se desarrollan y prueban en una ubicación temporal, antes de situarlas en su alojamiento definitivo. Si las referencias internas entre las páginas de un mismo sitio se realizan mediante enlaces relativos, no deberíamos tener problemas al mover las páginas, al menos en principio. En la práctica esto puede fallar si el sitio está basado en una aplicación CGI. En el siguiente capítulo estudiaremos técnicas para resolver esta dificultad. De todos modos, no está mal que conozca ahora la posibilidad de utilizar la etiqueta `<BASE>` para definir el camino respecto al cual interpretar las URL relativas. Esta etiqueta suele aparecer en la cabecera del documento:

```

<head>
<base href="http://www.marteens.com">
</head>

```

La etiqueta `<A>`, por otra parte, tiene un uso adicional: sirve para definir objetivos de saltos dentro de un documento. El siguiente ejemplo muestra un documento dividido en secciones, con un índice al inicio del mismo que permite saltos directos al inicio de cada sección. Observe que `<A>` no utiliza etiqueta de cierre cuando se utiliza para definir un objetivo de salto.

```

<html>
<body>

  <h2><a name=idx>Indice</h2><ul>
  <li><a href="#sec1">Sección 1</a></li>
  <li><a href="#sec2">Sección 2</a></li></ul>

  <h2><a name=sec1>Sección 1</h2>
  <p>Este es el texto de la primera sección.</p>
  <p><a href="#idx">Ir al índice</a></p>

  <h2><a name=sec2>Sección 2</h2>
  <p>Este es el texto de la segunda sección.</p>
  <p><a href="#idx">Ir al índice</a></p>

</body>
</html>

```

Aparte de los atributos *href* y *name*, podemos utilizar el atributo *title* que cuando está asignado indica el texto de la ayuda flotante que se presenta al pasar el ratón sobre el enlace.

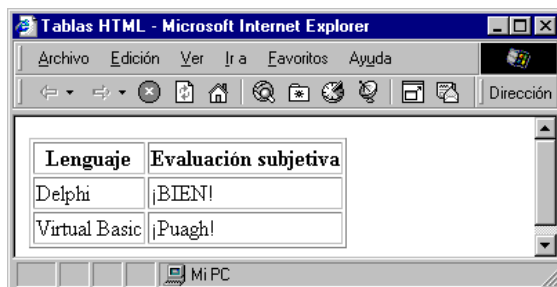
## Tablas HTML

Para poder presumir de conocer HTML es imprescindible familiarizarnos con los más triviales detalles de la sintaxis y uso de las *tablas HTML*. La gran importancia de estos recursos se debe a que actualmente son el medio más popular de controlar el formato y apariencia de los documentos Web. Una página Web típica está montada encima de un esqueleto compuesto por varias tablas anidadas. Y sí: muchas veces se abusa de las tablas sin motivo aparente.

El siguiente ejemplo define una tabla sencilla:

```
<table border=1>
  <tr>
    <th>Lenguaje</th>
    <th>Evaluación subjetiva</th>
  </tr>
  <tr>
    <td>Delphi</td>
    <td>¡BIEN!</td>
  </tr>
  <tr>
    <td>Virtual Basic</td>
    <td>¡Puagh!</td>
  </tr>
</table>
```

La tabla está formada por tres filas y dos columnas; note que no es necesario indicar las dimensiones por adelantado. Dentro de las etiquetas de inicio y fin de tabla, se colocan las etiquetas `<TR>` y `</TR>` para marcar el inicio y fin de cada definición de fila. A su vez, dentro de estas etiquetas pueden definirse dos tipos de celdas: las de cabecera, con la etiqueta `<TH>`, y las “normales”, con `<TD>`. La mayoría de los navegadores distinguen entre ambos tipos de celdas poniendo automáticamente el texto dentro de una celda de cabecera en negritas. Una celda “de cabecera” puede aparecer en cualquier parte de la tabla, no sólo en la primera fila. La siguiente imagen muestra la espantosa apariencia de nuestra primera tabla:



En principio, `<TR>`, `<TH>` y `<TD>` pertenecen al selecto grupo de etiquetas con cierre opcional. Mi consejo es que no se fíe de esta característica. He tenido amargas experiencias con un navegador ... que casualmente no es el de Microsoft. Algunos cardan la lana y otros crían la fama.

Estos son los atributos principales de la etiqueta `<TABLE>`:

Atributo	Propósito
ALIGN	La alineación de la tabla considerada como un todo
BGCOLOR	El color de fondo, por omisión
BORDER	El grosor de las líneas
CELLPADDING	Espacio exterior de separación entre celdas
CELLSPACING	Espacio interior que rodea al contenido de la celda
COLS	Número de columnas
WIDTH	Ancho de la tabla

Por otra parte, los atributos comunes a `<TH>` y `<TD>` son estos:

Atributo	Propósito
ALIGN	La alineación del contenido
BGCOLOR	Color del fondo de la celda
COLSPAN	Número de columnas que abarca
HEIGHT	Altura de la celda
NOWRAP	Evita que se introduzcan cambios de línea automáticos
ROWSPAN	Número de filas que abarca
VALIGN	Alineación vertical del contenido
WIDTH	Ancho de la celda

Finalmente, la etiqueta `<TR>` comparte algunos de los atributos de las etiquetas de celdas: alineación, color del fondo, evitar cambios de líneas...

## Trucos con tablas

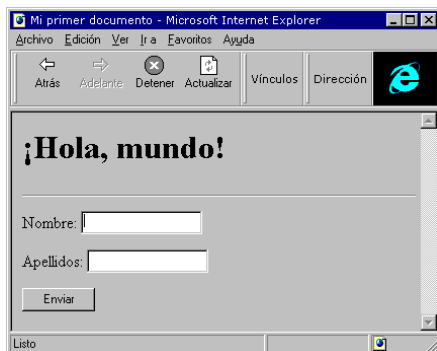
La mala noticia es que muchas de las características visuales de una tabla no pueden controlarse mediante el estándar actual de HTML. Netscape Navigator añade sus propias extensiones para controlar la apariencia de los bordes; Microsoft aporta extensiones diferentes. Un desastre, como puede imaginar. La buena noticia: todas estas características pueden manejarse de forma más conveniente mediante las hojas de estilo en cascada ... que sí, que más adelante veremos en qué consisten.

De todos modos, con las construcciones sintácticas que hemos estudiado podemos entender ya la forma en que se estructuran muchas páginas reales. Casi siempre el diseño de una página comienza por la creación de un esqueleto de tablas anidadas que delimitan con sus celdas distintas zonas dentro de la pantalla, como se muestra en el siguiente esquema:





El aspecto de este documento sobre un navegador es el siguiente:



La etiqueta `<FORM>` marca el inicio del formulario, y define dos parámetros: *method* y *action*. La acción se refiere a una URL que se activa cuando enviamos los datos tecleados en los controles de edición. Casi siempre esta URL se refiere a un programa CGI como los que estudiaremos en el próximo capítulo. Cuando la aplicación de destino recibe los parámetros del formulario tiene libertad para procesarlos como le venga en gana, y a cambio debe suministrar alguna respuesta correcta HTTP, que casi siempre consiste en otra página HTML. Adicionalmente, Netscape Navigator permite el protocolo *mailto:* en la URL de acción, para que los datos del formulario puedan enviarse a una dirección de correo electrónico.

El método debe ser *post* o *get*, e indica de qué manera se pasan los valores tecleados por el usuario al servidor. Si utilizamos *get*, los parámetros y sus valores se añaden a la propia URL de destino del formulario; en breve veremos cómo. En cambio, con el método *post* estos valores se suministran dentro del cuerpo de la petición, como parte del flujo de datos del protocolo HTTP.

## Controles de edición HTML

Los controles encerrados dentro de las etiquetas del formulario determinan los nombres y valores de los parámetros que se envían al servidor. Existen dos grandes grupos de controles: aquellos que se basan en la etiqueta `<INPUT>` ... y todos los demás. Realmente, `<INPUT>` es una etiqueta muy versátil que utiliza un atributo llamado *type* para indicar el tipo concreto de control que el navegador debe crear:

Valor de TYPE	Tipo de control
<i>text</i>	Un cuadro de edición, similar al <i>TEdit</i>
<i>password</i>	Un cuadro de edición para contraseñas
<i>checkbox</i>	Una casilla de verificación, similar a <i>TCheckBox</i>
<i>radio</i>	Botones de radio ( <i>TRadioButton</i> )
<i>button</i>	Un plebeyo botón ( <i>TButton</i> )
<i>submit</i>	Caso especial de botón que envía los datos editados
<i>reset</i>	Caso especial de botón que inicializa los controles

Valor de TYPE	Tipo de control
<i>image</i>	Botones basados en imágenes
<i>file</i>	Permite seleccionar un fichero
<i>hidden</i>	No tiene representación visual alguna

Aparte de los controles `<INPUT>`, HTML ofrece otros dos:

Etiqueta de control	Tipo de control
<code>&lt;TEXTAREA&gt;</code>	Similar al control <i>TMemo</i> de Delphi
<code>&lt;SELECT&gt;</code>	Similar a <i>TComboBox</i> y a <i>TListBox</i>

Tanto `<TEXTAREA>` como `<SELECT>` necesitan una etiqueta de cierre. Dentro del área delimitada por `<SELECT>` se listan las opciones de la selección, cada una precedida por una etiqueta `<INPUT>`. El interior de un `<TEXTAREA>` contiene el texto inicial que debe aparecer dentro del control:

```
<form>
  Tipo de tarjeta: <select name=CARD>
    <option>Visa
    <option>MasterCard
    <option>American Express
  </select><br>
  Ahora cuéntame un cuento:<br>
    <textarea cols=40 rows=5>Erase una vez...
  </textarea>
</form>
```

Veamos ahora el uso de los restantes atributos de los controles `<INPUT>`. Muy importante es el atributo *name*; si este atributo no está presente, el control no puede enviar datos al servidor. Todos los tipos de controles, incluyendo además a `<SELECT>` y `<TEXTAREA>` deben especificar su atributo *name*.

La pareja complementaria de *name* es *value*, pero este atributo puede omitirse en tiempo de diseño, pues su valor corresponde con el valor editado por el usuario. Cuando *value* está presente en la definición del control, se refiere al valor inicial que mostrará el control en pantalla. En un botón, *value* corresponde al título que aparece en el control. El siguiente botón carece de atributo *name*, por lo cual no contribuye a los parámetros enviados por el formulario. Sin embargo, su título será diferente al título por omisión de los botones *submit*, que en español es *Enviar datos*:

```
<input type=submit value="Aceptar">
```

Veamos ahora un ejemplo más o menos real de control de selección:

```
Tipo de tarjeta: <select name=CARD>
  <option value=0 selected>Visa
  <option value=1>MasterCard
  <option value=2>American Express
</select>
```

Tome nota acerca de cómo asignar valores a cada una de las opciones. En un ejemplo anterior mostré un `<SELECT>` cuyas opciones no especificaban valor alguno. En tales casos, el valor de cada opción corresponde con el del texto que sigue a la etiqueta.

Los botones de radio muestran un comportamiento similar a los controles de selección, y tienen también características sintácticas especiales:

```
<input type=radio name=CARD value=0 checked>Visa
<input type=radio name=CARD value=1>MasterCard
<input type=radio name=CARD value=2>American Express
```

Aquí la peculiaridad consiste en que varios controles deben compartir el mismo nombre; con valores diferentes, por supuesto. Observe el uso de *checked* en este ejemplo, y de *selected* en el anterior, para indicar la opción activa.

La mejor forma de familiarizarse con el comportamiento de estos objetos es por medio de la práctica. En el siguiente capítulo veremos cómo las aplicaciones CGI sacan partido de los parámetros enviados por los formularios.

Una técnica sencilla para que nuestros formularios tengan un aspecto decente es crear una tabla y distribuir los controles en sus celdas. Casi siempre basta una tabla con dos columnas. En la columna izquierda se colocan los títulos de los controles, alineados a la derecha, mientras que en segunda se sitúan los propios controles.

## Hojas de estilo

Si lo pensamos bien, HTML parece más un dictador que un amigo; en todo caso, es más amigo de los fabricantes de navegadores que de nosotros. Por ejemplo, nos ofrece una etiqueta `<H1>` para que delimitemos los encabezados de primer nivel. Sin embargo, no nos da garantía alguna sobre el tipo de letra que utilizará el navegador para representarla en pantalla. ¿No sería mucho pedir más control sobre la apariencia de nuestra página?

La respuesta la tiene una técnica conocida como *hojas de estilo* (*style sheets*), que nos permiten definir detalladamente (a veces demasiado) cada uno de los atributos de las etiquetas predefinidas en HTML. En realidad, existen dos estilos de hojas de estilo:

- *Cascading Style Sheets*, o CSS
- *JavaScript Style Sheets*, abreviado a JSS

El primer sistema está soportado tanto por Microsoft como por Netscape. Por el contrario, aunque *JSS* es más potente que *CSS*, solamente Netscape insiste en seguir dándole soporte. Por lo tanto, nuestra decisión ya está tomada: estudiaremos solamente *CSS*.

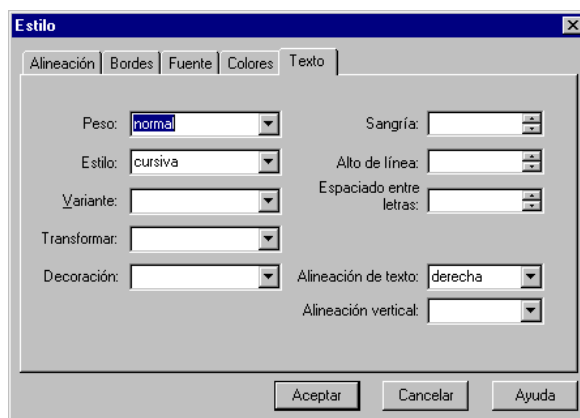


Para comprender rápidamente en que consisten las hojas de estilo, mostraremos un ejemplo sencillo. Podemos incluir dentro una página HTML, preferiblemente en la sección de cabecera, una definición de estilos similar a la siguiente:

```
<style>
  P { font-family: verdana, sans serif; text-align: left }
  TD { font-family: arial, sans serif; background-color: silver }
</style>
```

La letra `P` que aparece dentro de la definición de estilos se refiere simplemente a las etiquetas `<P>` que delimitan párrafos; a continuación, entre llaves, se definen valores para dos atributos de párrafos: *font-family* y *text-align*. Con las definiciones de estilo anteriores, todos los párrafos de la página y todas las celdas de tablas tendrán la apariencia definida sin necesidad de acciones adicionales por parte nuestra.

Puede que le parezca poca cosa, pero le contaré un caso real para convencerle de lo contrario. Al poco tiempo de estar programando páginas Web tropecé con una importante dificultad: no conocía forma razonable alguna de lograr cambiar el tipo de letra dentro de una tabla. He dicho “razonable”. Como no conocía las hojas de estilo intenté colar toda la definición de la tabla dentro de un par de etiquetas `<FONT>` y `</FONT>`. No funcionó; los navegadores ignoraban la sugerencia. Para no agotar su paciencia, fui moviendo las etiquetas de tipo de letra hacia el interior de la tabla hasta que funcionó. Desgraciadamente, la única forma de hacerlo era realizar el cambio de tipo de letra *dentro* de cada celda; dicho en menos palabras, había que repetir las etiquetas `<FONT>` celda por celda. Se dará cuenta de que esta técnica es potencialmente engorrosa, que hace crecer innecesariamente al fichero, que complica el mantenimiento de la página, pues si queremos cambiar el tipo de letra a la semana estamos obligados a realizar cambios coordinados en muchos sitios a la vez. Sin embargo, con las definiciones de estilo nos han bastado cuatro líneas...



También he mencionado antes que las hojas de estilo nos permiten controlar atributos que no están disponibles para HTML estándar. A pesar de todas las buenas razones, no todos los diseñadores de páginas utilizan estilos. Conozco a un diseñador

muy bueno que se sorprendió cuando le mencioné la existencia de esta técnica. La imagen anterior corresponde al diálogo de definición de estilos de Front Page 98.

## Atributos para estilos

Veamos los atributos más comunes que podemos utilizar en una declaración de estilos. Si tiene un manual de HTML a mano verá que muchos de estos atributos no tienen una contrapartida similar en el HTML estándar. Estos atributos controlan la apariencia del texto:

Atributo	Significado
<i>font-family</i>	El nombre del tipo de letra
<i>font-size</i>	Tamaño del tipo de letra
<i>font-weight</i>	Grosor de la letra: negritas o normal
<i>font-style</i>	Normal, itálicas o letras oblicuas
<i>font-variant</i>	Activa el formato de LETRAS VERSALES
<i>text-align</i>	Alineación horizontal del texto
<i>text-decoration</i>	Subrayados
<i>vertical-align</i>	Alineación vertical
<i>color</i>	Color del texto

Algunos de los atributos anteriores pueden unirse para disminuir la longitud de una declaración. Tal es el caso de los atributos que controlan el tipo de letra, que pueden abreviarse mediante el atributo *font*:

```
TH { font: bold 10pt verdana, arial, sans serif }
```

Estos atributos controlan los márgenes, no sólo de los elementos de texto, sino de cualquier otro elemento en general:

Atributo	Significado
<i>margin-top</i>	Margen superior
<i>margin-bottom</i>	Margen inferior
<i>margin-left</i>	Margen izquierdo
<i>margin-right</i>	Margen derecho
<i>text-indent</i>	Sangría de la primera línea

El atributo *margin-bottom*, por ejemplo, es el que controla la separación entre dos párrafos consecutivos que mencioné antes. Es también posible agrupar las definiciones de márgenes. El siguiente ejemplo define de una vez los márgenes superior, derecho, inferior e izquierdo de un párrafo:

```
P { margin: 0mm 1cm 5mm 3cm }
```

Si hubiésemos indicado un solo valor, se habría aplicado a los cuatro márgenes. Con dos valores, el primero correspondería al margen inferior y superior, y el segundo al

margen derecho e izquierdo. Con tres valores estaríamos indicando la secuencia arriba, izquierda/derecha y abajo. Note que las unidades de los valores se han especificado indistintamente como milímetros y centímetros. También se pueden utilizar píxeles (*px*) y “unidades eme” (*em*), que representan el ancho aproximado de una letra *eme*<sup>32</sup>.

Existen también conjuntos de atributos para controlar el borde de tablas y celdas (*border-color*, *border-style*), el fondo de un elemento (*background*), la apariencia de las listas en HTML (*list-style*)... Y están también las correspondientes abreviaturas para que no se nos desgasten los dedos tecleando.

## Clases para etiquetas

Sin embargo, definir un solo estilo para una etiqueta de uso tan frecuente como `<P>` es meternos en una camisa de fuerza dos tallas más pequeña que la nuestra. Podemos evitar este problema utilizando *clases* dentro de la definición de estilos.

```
<style>
P { font-family: arial, sans serif }
P.destacar { background-color: silver }
</style>
```

Para indicar que queremos utilizar el estilo de párrafo “importante” tenemos que incluir el atributo *class* en la etiqueta de apertura:

```
<p class=destacar>Nunca olvide las llaves de casa.</p>
```

Algo muy importante: existe una relación de herencia entre las definiciones de estilo de una etiqueta que no especifican clase y las clases de estilo. En el ejemplo anterior, la clase *destacar* utiliza el tipo de letra Arial aunque no lo haya indicado explícitamente, simplemente porque la declaración de estilo para los párrafos generales sí especifica este tipo de letra.

## Clases especiales y otras triquiñuelas

Hay asociaciones inevitables: una mañana soleada me hace pensar en cierta chica, un cálido sonido de saxo en una película nos indica que viene una escena de romance, un texto rojo en negritas en una página Web nos dice que el autor de la página es un hortera de mucho cuidado ... pero que intenta decirnos algo sumamente importante. Lo que quiero decir es que un texto en negritas con un tamaño ligeramente mayor siempre da la sensación de advertencia, y nos da lo mismo que el texto esté dentro de un párrafo, una celda, una lista o donde se nos ocurra. ¿Qué hacemos, crear una clase especial para cada tipo? No, si sabemos cómo definir *clases de estilo genéricas*:

```
.OJO { font: bold 12pt arial; color: red }
```

---

<sup>32</sup> La *eme* es la letra que más desesperadamente clama por una cura de adelgazamiento.

Ahora puede aplicar esta clase a cualquier etiqueta:

```
<p>No cruces la calle con el  
semáforo en <span class=OJO>rojo</span>.</p>
```

He mezclado deliberadamente dos trucos: el uso de la clase genérica antes definida y una etiqueta `<SPAN>`. ¿Qué hace esta nueva etiqueta? Nada, sinceramente; sólo marca una porción dentro de un texto. Claro que si asociamos alguna clase de estilo a la etiqueta tendremos la posibilidad de cambiar su aspecto. Similar es el uso de la etiqueta `<DIV>`. La diferencia está en que `<DIV>` fuerza un cambio de línea dentro del documento. Piense en esta etiqueta como un sucedáneo de los cambios de secciones en un procesador de texto.

¿Más trucos? Debe tener mucho cuidado con el que ahora le presentaré. La siguiente definición sirve para crear estilos para párrafos y enlaces:

```
P, A { font: 10pt verdana }
```

Sin embargo, esta otra definición crea un estilo para los enlaces que se encuentren *dentro* de los límites de un párrafo:

```
P A { font: bold 10pt verdana }
```

Note que la única diferencia ha sido la ausencia de la coma entre los dos nombres de etiquetas. ¿Recuerda las etiquetas de cambio de aspecto basado en la semántica que mencionaba casi al principio del capítulo? Ahora tenemos la posibilidad de precisar su acción. Por ejemplo:

```
P EM { font-weight: bold }  
A EM { font-style: italic }
```

Con estos estilos, una etiqueta `<EM>` activará las negritas si ocurre dentro de un párrafo, pero cambiará el texto a cursivas si se encuentra dentro de un enlace.

También tenemos *pseudo clases* de estilos. Este término se refiere a ciertos estados o contextos en el que puede aparecer determinada etiqueta. Por ejemplo, un enlace definido mediante la etiqueta `<A>` puede encontrarse en uno de tres estados:

- El estado normal (*link*)
- En proceso de visita (*active*)
- Ya visitado (*visited*)

Para cada uno de estos estados puede definir un estilo diferente si aprovecha la siguiente sintaxis:

```
A: link { color: blue; text-decoration: none }  
A:visited, A:active { color: red }
```

Internet Explorer define un estado adicional para las etiquetas de enlace, *hover*, que se activa cuando el ratón pasa por encima del contenido del enlace.

Las otras dos pseudo clases existentes en estos momentos se aplican a la etiqueta de párrafos `<P>`, y se llaman *first-line* y *first-letter*. Con estos nombres es fácil saber cuáles son sus usos.

## Estilos enlazados y en línea

Hasta el momento he mostrado hojas de estilo incluidas directamente en la página HTML; lo he hecho por comodidad, pues así podía mostrar en un mismo listado la definición del estilo y la forma de utilizarlo. Sin embargo, existen otras dos formas de crear hojas de estilo: mediante vínculos y en línea. En realidad, los estilos vinculados son los de uso más frecuente y recomendable.

Imagine que en vez de escribir una y otra vez la misma lista de estilos al inicio de cada página Web, agrupamos la declaración de estilos en un fichero separado, digamos que *estilos.css*, y lo ubicamos en un directorio del servidor. La extensión *css* no es obligatoria; con ella he querido indicar que se trata de un fichero con hojas de estilo en cascada (*Cascading Style Sheets*).

Una vez realizado el paso anterior, cualquier página de ese directorio puede incluir la siguiente declaración en su cabecera, para tener acceso a las definiciones de estilos comunes:

```
<link rel="stylesheet" href="estilos.css" type="text/css">
```

Por supuesto, no hace falta que la definición de estilos esté físicamente en el mismo directorio que las páginas que la vinculan. En el caso más general, el atributo *href* puede incluir una URL absoluta en vez de una relativa.

Ahora piense qué hacer si en un único fichero, en un único párrafo o celda de una tabla, necesita un estilo muy especial. ¿Estaremos obligados a crear una clase sólo por esto? No, si utilizamos un estilo en línea:

```
<p style="text-indent:-2em; margin-left:2em">
Este párrafo aparece con una sangría francesa. La primera línea
comienza más a la izquierda que las demás.</p>
```

No debemos abusar de los estilos en línea, como debe imaginar, pues mantener una página con abundancia de ellos es una verdadera pesadilla.



## Fundamentos de JavaScript

EL SIGUIENTE PASO EN LA ADICIÓN DE potencia a HTML fue, desgraciadamente, un emplasto chapucero. En vez de extender las capacidades del propio HTML, algunos fabricantes decidieron que éste siguiera siendo un simple lenguaje de descripción, pero que pudieran intercalarse instrucciones de otros lenguajes más “tradicionales”, y que estas instrucciones pudieran actuar sobre el propio código HTML.

Obviamente, estas instrucciones debían ser ejecutadas por el navegador de Internet correspondiente. Por sencillez se decantaron por lenguajes interpretados. Los dos lenguajes con más éxito en esta categoría son actualmente JavaScript, por méritos propios, y VBScript, por ser obra de Microsoft. JavaScript fue introducido originalmente por Netscape, pero también es soportado por Microsoft. Es, por lo tanto, el lenguaje de *script* recomendable para añadir algo de vida a sus páginas Web.

### La estructura de JavaScript

Todos los autores comienzan diciendo que JavaScript no tiene nada que ver con Java, y no voy a ser menos que los demás: JavaScript no tiene nada que ver con Java, excepción hecha de las cuatro primeras letras comunes. De hecho, todo el que conoce la existencia de JavaScript lo asocia con Internet, y aunque es cierto que la Red es el nicho ecológico que hace posible su existencia, en teoría JavaScript podría utilizarse como lenguaje de propósito general.

¿Dije en teoría, verdad? He aquí algunas manías y rarezas de JavaScript:

- Alguien consideró que la diferencia entre la “A” y la “a” era un dogma importante. JavaScript es sensible al tamaño de las letras. Quisiera escuchar una defensa de esta decisión y seguir respetando intelectualmente al defensor.
- JavaScript no comprueba tipos y operaciones estáticamente. Se debe, simplemente, a que un *script* de este lenguaje no se compila: no existe el “tiempo de compilación”.
- JavaScript está inspirado en C, no en C++ o Java. El modelo de objetos, entre otras características, es completamente diferente al conocido modelo de estos lenguajes.

De modo que podemos ahorrarnos algunas explicaciones, al menos sintácticamente. JavaScript se basa en las instrucciones de C de toda la vida: **while**, **if**, **do**, **for**, todas las variantes de la asignación... También son idénticos los operadores. Donde notaremos la primera diferencia importante es en la forma de definir funciones:

```
function fact(x)
{
    var rslt = 1;
    for (var i = 2; i <= x; i++)
        rslt = rslt * i;
    return rslt;
}
```

Recuerde que en C y sus secuelas las declaraciones de funciones y de variables comienzan con un tipo. Como los tipos no existen en tiempo de compilación en JavaScript, había que sustituir estas declaraciones iniciales por algo: la palabra reservada **function**, en el caso de las funciones, y **var**, para las variables.

Aunque está claro que no existen declaraciones de tipos en tiempo de compilación, es lógico e inevitable que sí existan tipos en tiempo de ejecución. JavaScript reconoce tres tipos básicos y dos compuestos:

- Números (tanto enteros como reales)
- Cadenas
- Tipo lógico, o *boolean*
- Vectores (*arrays*)
- Objetos

A cada uno de los tres tipos básicos corresponde una clase de objetos: a los números, la clase *Number*, a las cadenas, la clase *String*. De esta manera, podemos escribir instrucciones como la siguiente, donde tratamos a una variable de cadena como si fuese un objeto:

```
var s1 = "0123456789";
var s2 = s1.charAt(i);
```

## Ejecución durante la carga

Ahora ya tenemos algunos conocimientos acerca de cómo escribir sencillas funciones en JavaScript. Sólo nos falta saber aplicarlas, y lo primero que debemos aprender es cómo incluir instrucciones de JavaScript dentro de un fichero HTML. ¿Es tan complicado? No, basta con delimitar el código dentro de un par de etiquetas `<SCRIPT>`. El único problema consiste en mantener la compatibilidad, dentro de lo posible, con aquellos navegadores que no soportan JavaScript.

El procedimiento de inclusión básico es similar a éste:

```
<script>
```



```
<!--
    document.writeln("Bienvenido a JavaScript");
// -->
</script>
```

El lexema<sup>33</sup> `<!--` es el inicio de comentario en HTML. Si el navegador soporta JavaScript, entonces debe aceptar los comentarios HTML dentro de las etiquetas `<SCRIPT>` ... pero con un ligero cambio: en vez de tratarlo como el inicio de un comentario que abarca varias líneas, debe interpretarlo en este contexto como un comentario hasta el fin de línea. Los navegadores antiguos, por su parte, se saltan alegremente todo el interior del *script*.

Y entonces llegamos a la penúltima línea. JavaScript no reconoce el fin de comentario HTML; Netscape Navigator es particularmente pedante en este punto. Pero hemos tenido la precaución de situar un inicio de comentario de JavaScript en la misma línea. Todos contentos y felices.

... un momento, ¿qué hace la misteriosa instrucción que hemos incluido? La variable *document* se refiere, evidentemente, al documento HTML actual. El método *writeln* añade una cadena al documento. ¡Estamos añadiendo contenido dinámico al documento en tiempo de carga!

¿Qué necesidad tenemos de “desenrollar” el código HTML en el cliente? Muy fácil: puede que queramos elementos HTML distintos en dependencia de las condiciones actuales del cliente, casi siempre, de acuerdo al tipo de navegador que se esté utilizando para visualizar la página. Por ejemplo, usted quiere que en algún lugar de la página aparezca la fecha actual, de acuerdo al ordenador del usuario. Supongamos que la fecha aparecerá dentro de una celda. Puede entonces programar algo similar a esto:

```
<script>
<!--
    document.writeln((new Date()).toLocaleString());
// -->
</script>
```

Claro, existen muchas más aplicaciones para estas instrucciones que se ejecutan en tiempo de carga. Por ejemplo, la siguiente instrucción asigna un valor lógico a una variable global en dependencia del tipo de navegador en que se carga la página. Más adelante, otras instrucciones de JavaScript pueden aprovechar este valor:

```
var isMicro = (navigator.appName.indexOf("Microsoft") != -1);
```

También es posible pasar el foco del teclado a determinado control del documento, una vez leído todo el código HTML:

```
<script><!--
```

---

<sup>33</sup> Uno también lee diccionarios.

```

        document.form1.edit1.focus();
// -->
</script>

```

## Eventos HTML

Sabemos que el axioma número 1 de Delphi dice aproximadamente: todo lo que ocurre en Delphi ocurre en respuesta a un evento. Más o menos lo mismo puede decirse de JavaScript. Incluso la técnica de generación dinámica del documento puede interpretarse como acciones que podemos llevar a cabo en el evento que acompaña a la carga de la página.

Las extensiones a HTML hacen que muchos de los elementos definidos en una página puedan disparar eventos. Los candidatos más evidentes son los controles de un formulario. Analice este sencillo ejemplo:

```

<form name=calculadora>
  <table cols=2>
    <tr>
      <td>Primer operando</td>
      <td><input type=text name=op1></td>
    </tr><tr>
      <td>Segundo operando</td>
      <td><input type=text name=op2></td>
    </tr><tr>
      <td colspan=2>
        <input type=button value="Calcular"
          onClick="evalForm(calculadora);">
      </td>
    </tr>
  </table>
</form>

```

En primer lugar, note que le hemos dado un nombre al formulario, asociando en su etiqueta de inicio un atributo *name*. Después, al botón que tiene el título *Calcular* le hemos asociado, a través de su evento *onclick*, una llamada a cierta función *evalForm*, pasando como parámetro el formulario en que se encuentra. La función en sí es muy sencilla:

```

<script>
function evalForm(f)
{
  var sum = parseInt(f.op1.value) + parseInt(f.op2.value);
  alert("La suma es " + sum);
}
</script>

```

Ahora piense: ¿qué otro sentido tendría la existencia de controles *button* que no sean de las clases *submit* y *reset*? Al menos estas dos clases especiales llevan a cabo acciones predeterminadas. Un botón de clase general, por el contrario, necesitaría ayuda proveniente del programador para saber qué hacer cuando lo pulsan. Bien, pues ese es una de las obligaciones de los eventos HTML.

## DOM es una ruina

Quiero confesarle que soy un sucio tramposo. ¿Se ha percatado de que la suma de los valores ha ido a parar a un cuadro de diálogo emergente? Lo más natural para el ejemplo mostrado hubiera sido modificar el texto de algún elemento estático, como el contenido de una celda de una tabla. ¡Ah, pero para eso necesitamos Internet Explorer! Esta es una nueva versión de la calculadora, que solamente funciona para el navegador de Microsoft:

```
<script>
function evalForm(f) {
    var sum = parseInt(f.op1.value) + parseInt(f.op2.value);
    rslt.innerHTML = sum; // ¡¡¡ CAMBIOS !!!
}
</script>

<form name=calculadora>
  <table cols=2>
    <tr>
      <td>Primer operando</td>
      <td><input type=text name=op1></td>
    </tr><tr>
      <td>Segundo operando</td>
      <td><input type=text name=op2></td>
    </tr><tr>
      <td>Resultado</td>
      <td id=rslt></td>
    </tr><tr>
      <td colspan=2>
        <input type=button value="Calcular"
          onClick="evalForm(calculadora);">
      </td>
    </tr>
  </table>
</form>
```

Hay cambios importantes dentro del formulario, más bien dentro de la tabla que da formato a los campos, y en el código de la función. Una de las celdas de la tabla ha sido “bautizada”, al asociarle un atributo *ID* con un valor igual a *rslt*. En la función *evalForm* es ahora posible hacer referencia a *rslt* como si fuera una variable global que apuntase a un objeto de tipo celda. A este objeto le hemos modificado su propiedad *innerHTML*, que representa el texto encerrado por la celda. Una maravilla, ¿verdad?

Esta magia es posible gracias a DOM, un acrónimo que significa *Document Object Model*, y que representa el conjunto de convenciones necesarias para poder modificar un documento dinámicamente mediante algún lenguaje de *scripts*. DOM nos permite trabajar con el documento como si se tratase de un árbol de elementos, similar al árbol de análisis sintáctico que utiliza internamente el navegador.

Como he dicho antes, la situación actual de este modelo es deplorable, pues Microsoft y Netscape implementan versiones completamente diferentes del mismo, y curiosamente la versión de Microsoft es la más potente y elegante. Posiblemente sea la

que sirva de base al estándar actualmente en elaboración. El problema de Netscape es que representa mediante objetos sólo algunos tipos de elementos HTML: enlaces, anclas y controles de formularios. En el ejemplo anterior podríamos haber utilizado un control de edición *text* para dejar el resultado. Claro, el usuario podría entonces confundirse al ver un resultado estático dentro de un control modificable.

No sólo hay diferencias entre la estructura de los documentos, sino incluso en el modelo de eventos de los dos navegadores principales. En primer lugar, no todos los elementos HTML disparan eventos en Netscape Navigator. ¡Incluso hay diferencias entre las versiones de este producto para distintos sistemas operativos! Además, modelo de eventos soportado por Microsoft vuelve a ser más completo y poderoso que el de su rival. Por ejemplo, para Internet Explorer es correcto el siguiente fragmento de código:

```
function DoMouseOver() {
    window.event.srcElement.style.color = "lime";
}
```

El último evento disparado deja una referencia en una propiedad del objeto global *window* y esta referencia, a su vez, apunta al elemento que causó su disparo.

## Validación de formularios

La aplicación más importante de JavaScript, por lo menos para los desarrolladores de aplicaciones Web, es la validación de los valores tecleados en un formulario HTML. Como sabemos, existen tres momentos importantes durante la entrada de datos en un cuadro de diálogo o en un formulario en los cuales podemos verificar que los valores suministrados por el usuario son correctos:

- Cada vez que se efectúa un cambio en el control activo.
- Cuando se abandona el control activo o, en algunos casos, cuando se pulsa determinada tecla, casi siempre INTRO, sin abandonar el control.
- Al terminar la entrada de datos.

La plebe usuaria, en general, piensa que el método más potente, y por consiguiente más apropiado, es el primero. No obstante, es el método que más difícil es de programar, a no ser que tengamos alguna ayuda por parte del sistema. Además, puede ser bastante incómodo convivir con controles demasiado estrictos: piense que para editar una fecha, por ejemplo, debemos pasar por estados intermedios en los cuales el control de edición no contiene una fecha correcta.

El segundo método de validación puede implementarse en HTML si recurrimos al evento *onblur* de los controles de edición. El tercer método se basa en interceptar el evento *onsubmit* del formulario que deseamos validar. Si la función que ejecutamos devuelve **false**, el formulario no envía sus datos. Este será el método que ejemplifi-

caremos, pues es el más sencillo y no nos obliga a sacrificar ningún tipo de comprobaciones.

La siguiente función escrita en JavaScript verifica que hayamos tecleado algo en el control *edit1* del formulario que se le pasa como parámetro. Si no es así, se muestra un mensaje y se pasa el foco del teclado al control majadero:

```
<script>
function ValidateForm(f) {
  if (f.edit1.value.length == 0) {
    f.edit1.focus();
    alert("Debe teclear algo en este control");
    return false;
  }
  return true;
}
</script>
```

La función se asocia al evento *onsubmit* de un formulario de la siguiente manera. Observe el uso de **this** para pasar el formulario como parámetro, y el hecho de que debe existir un control nombrado *edit1* para que *ValidateForm* funcione:

```
<form method=get action="http://www.yoQueSe.com/encuesta.dll"
  onSubmit="return ValidateForm(this);">
  <p>Dígame algo agradable, aunque sea mentira:
  <input type=text name=edit1><br>
  <input type=submit value="Enviar"></p>
</form>
```

## Expresiones regulares

El ejemplo de validación anterior es simple, quizás demasiado. Mis primeras maniobras en este asunto las realicé imitando el código JavaScript que genera Front Page. Esta herramienta, sin embargo, no genera un código muy recomendable, imagino que buscando compatibilidad con las primeras versiones del navegador de Microsoft. Un buen día descubrí casi por azar la posibilidad de utilizar *expresiones regulares* para la validación de cadenas. Y se hizo la luz.

Las expresiones regulares son objetos de JavaScript que pertenecen a la clase *RegExp*. Tienen una particularidad: podemos escribir constantes de esta clase mediante una notación especial que se asemeja a la de las cadenas de caracteres:

```
var patron = /^\\d\\d?\\/\\d\\d?\\/\\d\\d(\\d\\d)?$/;
```

La constante va encerrada entre barras inclinadas a la derecha. Una expresión regular representa un patrón que podemos aplicar sobre una cadena para ver si se ajusta o no. En el ejemplo anterior he utilizado un patrón para reconocer fechas. Parece una fórmula extraída de un tratado de alquimia, pero veremos que es muy fácil de interpretar.

Comencemos por el primer y el último carácter: el acento circunflejo corresponde al inicio de línea, mientras que el símbolo de dólar representa el final. El patrón los ha incluido para exigir que la cadena que comparemos contenga sólo una fecha; en caso contrario, podríamos tener una cadena que, entre otras cosas, encerrase una fecha.

Los caracteres listados a continuación representan comodines dentro de una expresión regular:

Carácter	Significado
.	Cualquier carácter distinto del cambio de línea
\w	Una letra, un dígito o el subrayado
\W	Lo contrario a \w
\s	Espacio en blanco, tabulación, cambio de línea ...
\S	Lo contrario a \s
\d	Un dígito
\D	Cualquier carácter excepto un dígito

La barra invertida frente a cualquier otro carácter significa el propio carácter. Es el truco necesario para poder incluir la barra inclinada de los separadores sin que JavaScript piense que se trata del final del patrón.

Para crear comodines más potentes debemos encerrar entre corchetes un conjunto de caracteres. Por ejemplo:

Combinación	Significado
[A-Fa-f0-9]	Un dígito hexadecimal
[AEIOUaeiou]	Las vocales
[^AEIOUaeiou]	Cualquier carácter menos las vocales

Ahora vienen algunos operadores que tienen que ver con la repetición de patrones:

Operador	Significado
<i>elem?</i>	El elemento anterior es opcional
<i>elem*</i>	El elemento anterior se repite cero o más veces
<i>elem+</i>	El elemento anterior se repite una o más veces
<i>elem</i> { <i>n</i> , <i>m</i> }	El elemento anterior se repite de <i>n</i> a <i>m</i> veces
<i>elem</i> { <i>n</i> }	El elemento anterior se repite exactamente <i>n</i> veces
<i>elem</i> { <i>n</i> , }	El elemento anterior se repite <i>n</i> o más veces

En nuestro ejemplo hemos utilizado el signo de interrogación para indicar que el día puede constar de uno o dos dígitos, lo mismo que el mes, y que el año comienza con dos dígitos y admite un grupo de dos dígitos opcionales a continuación.

Existen varios métodos para comparar una cadena con un patrón. El más sencillo es *search*, que cuando encuentra una ocurrencia del patrón dentro de la cadena devuelve la posición, y en caso contrario devuelve un valor negativo:

```
function isOk(editor, patron)
{
    if (editor.value.search(patron) == -1)
    {
        editor.focus();
        alert("El valor de este campo es incorrecto");
        return false;
    }
    return true;
}
```

## Juego de espejos

Verdaderamente dudo que el siguiente ejemplo le sea de alguna utilidad. Quien sabe. Por el momento échele un vistazo:

```
<HTML>
<HEAD>
  <TITLE>Juego de espejos</TITLE>
</HEAD>

<SCRIPT>
function continuar()
{
    return "<head><title>Juego de espejos</title></head>" +
           "<body><p>...que soñaba que era Chu-Lin.</p>" +
           "<a href=\"javascript:history.back(1);\">" +
           "Continuar la historia</a></body>";
}
</SCRIPT>
<BODY>
  <p>Chu-Lin soñó que era una mariposa...</p>
  <a href="javascript:continuar();">Continuar la historia</a>
</BODY>
</HTML>
```

Con este documento he querido presentar una interesante característica de la mezcla entre HTML y JavaScript, además de rememorar una conocida fábula china. La característica presentada son las URLs de JavaScript: observe que el atributo *href* del enlace situado en el cuerpo del documento no hace referencia a una URL normal. Por lo general, una URL absoluta comienza mencionando un protocolo, casi siempre *http*;, o *file*:. En este caso comienza con *javascript*;, y cuando pulsamos el ratón sobre el enlace simplemente se ejecutan la lista de sentencias JavaScript que contiene la URL. Puede suceder una de dos cosas:

- La última instrucción devuelve un texto. En ese caso, ese texto se entiende como el contenido de un documento HTML y se muestra en el propio navegador.
- La última instrucción no tiene valores de retorno. Entonces no pasa nada, aparte de los efectos que puedan causar las instrucciones JavaScript ejecutadas.

La principal utilidad de las URLs de JavaScript, de acuerdo a mi estilo de programación, se encuentra en la posibilidad de simular el envío de formularios mediante enlaces. Suponiendo que *form1* es un formulario definido antes, cuando pulsamos el siguiente enlace se envían los datos del formulario a su destino, exactamente igual que si hubiéramos pulsado un botón *submit*:

```
<a href="javascript:form1.submit();">Enviar datos</a>
```

En el próximo capítulo comprenderemos la necesidad de esta simulación.

No me gusta dejar una historia a medias, aunque no tenga un final feliz. Algún desaprensivo hizo sonar un gong en el jardín y la mariposa, de repente, despertó.



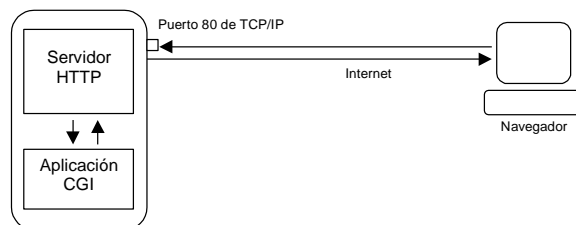
## WebBroker

AUNQUE DELPHI SOPORTA VARIOS MODELOS DE desarrollo para Internet, el más importante, por sus posibilidades, es la programación de aplicaciones CGI, que se ejecutan en el servidor; claro está, con sus respectivas variantes especiales para Internet Information Server (ISAPI), Netscape (NSAPI) y Apache. Es tan importante este tipo de programas que Delphi incluye no una, sino dos plataformas de desarrollo para ellos: WebBroker, la más antigua y segura, y WebSnap, moderno, prometedor... y aún plagado de problemas.

Comenzaremos nuestra excursión por WebBroker, no sólo por ser el sistema más viejo y conocido, sino además porque WebSnap utiliza gran parte del código propio de WebBroker para sus propios fines.

### Aplicaciones CGI e ISAPI

Las primeras aplicaciones de Internet para el lado servidor consistían en ficheros ejecutables: programas desarrollados con algún lenguaje de propósito general como C, Pascal o Perl. A este tipo de programas creadores de páginas dinámicas se les dio el nombre colectivo de aplicaciones CGI, del inglés *Common Gateway Interface*. Es responsabilidad del servidor HTTP el ejecutar la extensión CGI adecuada a la petición del cliente, y pasarle los parámetros mediante la entrada estándar o las variables de entorno. El texto generado por el programa se envía de vuelta al servidor HTTP a través de la salida estándar de la aplicación, un mecanismo muy a lo UNIX que revela los orígenes de la técnica.



Como puede imaginar, todo este proceso de llamar al ejecutable, configurar el flujo de entrada y recibir el flujo de salida es un proceso costoso para el servidor HTTP.

Además, si varios usuarios solicitan simultáneamente los servicios de una aplicación CGI, esta será cargada en memoria tantas veces como sea necesario, con el deducible desperdicio de recursos del ordenador. Los dos principales fabricantes de servidores HTTP en el mundo de Windows, Microsoft y Netscape, idearon mecanismos equivalentes a CGI, pero que utilizan DLLs en vez de ficheros ejecutables. La interfaz de Microsoft se conoce como ISAPI, mientras que la de Netscape es la NSAPI, y éstas son algunas de sus ventajas:

- El proceso de carga es más rápido, porque el servidor puede mantener cierto número de DLLs en memoria un tiempo determinado. Una segunda petición de la misma extensión no necesita volver a cargarla en memoria.
- La comunicación es más rápida, porque el servidor ejecuta una función de la DLL, pasando datos y recibiendo la página mediante parámetros en memoria.
- La ejecución concurrente de la misma extensión debido a peticiones simultáneas de cliente es menos costosa en recursos, porque el código se carga una sola vez.
- Como veremos, es más fácil depurar una extensión ISAPI/NSAPI que una aplicación CGI.

¿Desventajas? Hay una importante, sobre todo para los malos programadores (es decir, que no nos afecta):

- La extensión pasa a formar parte del espacio de procesos del servidor. Si se cuelga, el servidor (o uno de sus hilos) también se cuelga.

También puede tener problemas con las aplicaciones ISAPI/NSAPI durante el desarrollo de acuerdo al tipo de servidor HTTP que utilice. Como el servidor normalmente deja cargada la DLL en memoria en espera de posteriores peticiones, no es posible sustituir la DLL como parte del proceso normal de prueba, cambios y compilación. Algunos servidores permiten desactivar esta característica con mayor facilidad que otros.

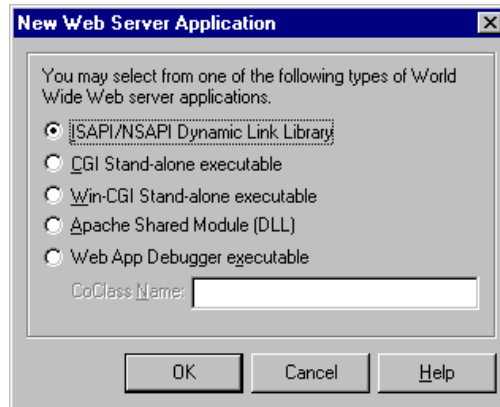
En cualquier caso queda claro que es preferible utilizar una DLL antes que una aplicación CGI.

## Módulos Web

Podemos crear aplicaciones CGI fácilmente con cualquier lenguaje sencillo: basta con desarrollar una aplicación de tipo *consola* y manejar directamente las variables del entorno y la salida estándar. Recuerde que en Delphi las aplicaciones de tipo consola se crean en el diálogo de opciones del proyecto, y que son aproximadamente equivalentes a las aplicaciones “tradicionales” de MS-DOS. Pero seríamos responsables de todo el proceso de análisis de la URL, de los parámetros recibidos y de la generación del código HTML con todos sus detalles. Si nuestra extensión Web es muy simple, como puede serlo la implementación de un contador de accesos, no sería demasiado

asumir todas estas tareas, pero las aplicaciones más complejas pueden irsenos de las manos. Lo mismo es aplicable a la posibilidad de programar directamente DLLs para las interfaces ISAPI y NSAPI: tarea reservada para tipos duros de matar.

Para crear una aplicación Web, debemos invocar al Depósito de Objetos, comando *File|New*, y elegir el icono *Web Server Application*. El diálogo que aparece a continuación nos permite especificar qué modelo de extensión Web deseamos:



En cualquiera de los casos, aparecerá en pantalla un módulo de datos, de nombre *WebModule1* y perteneciente a una clase derivada de *TWebModule*. Lo que varía es el fichero de proyecto asociado. Si elegimos crear una aplicación CGI, Delphi generará el siguiente fichero *dpr*:

```
program Project1;

{$APPTYPE CONSOLE}

uses
  WebBroker,
  CGIApp,
  Unit1 in 'Unit1.pas' {WebModule1: TWebModule};

{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TWebModule1, WebModule1);
  Application.Run;
end.
```

Como se puede ver, el proyecto será compilado como un fichero ejecutable. En cambio, si elegimos ISAPI/NSAPI, Delphi creará código para una DLL:

```
library Project1;

uses
  WebBroker,
```

```

ISAPIApp,
Unit1 in 'Unit1.pas' {WebModule1: TWebModule};

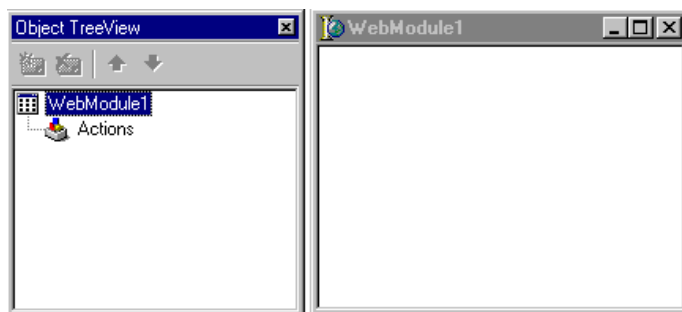
{$R *.RES}

exports
  GetExtensionVersion,
  HttpExtensionProc,
  TerminateExtension;

begin
  Application.Initialize;
  Application.CreateForm(TWebModule1, WebModule1);
  Application.Run;
end.

```

¿Tres funciones para exportar? Sí, se trata de las funciones que el servidor Web espera que tenga nuestra extensión, y las llamará durante la carga de la DLL, para la generación de documentos dinámicos y para descargar finalmente la DLL cuando haga falta. Delphi implementa estas tres funciones por nosotros.



En ambos modelos, el objeto *Application* que se inicializa y “ejecuta” no pertenece a la conocida clase *TApplication* que utilizan los programas de la interfaz gráfica. Esta vez *Application* se refiere a una variable global definida en la unidad *WebBroker* y que ha sido declarada mediante el tipo *TWebApplication*. Las aplicaciones CGI inicializan *Application* con un objeto de la clase derivada *TCGIApplication*, mientras que las ISAPI se basan en *TISAPIApplication*.

Existe una alternativa a la creación de nuevos módulos Web mediante el asistente de Delphi. Si ya tenemos un módulo de datos, desarrollado en otra aplicación, podemos crear el esqueleto inicial tal como hemos explicado, eliminar del proyecto el módulo Web y añadir entonces el módulo existente. ¡Pero el módulo eliminado descende de *TWebModule*, mientras que el módulo añadido es un vulgar descendiente de *TDataModule*! No importa, porque a continuación añadiremos al módulo un componente *TWebDispatcher*, de la página *Internet*. Este componente contiene una propiedad llamada *Actions*, que es precisamente la que marca la diferencia entre los módulos de datos “normales” y los módulos Web.

## El proceso de carga y respuesta

Es muy importante que comprenda al detalle la forma en que una aplicación escrita en Delphi maneja las peticiones HTTP y genera las respuestas adecuadas. De este conocimiento depende, en gran medida, el éxito de su funcionamiento. Comenzaremos con una descripción de lo que sucede dentro de una CGI, por ser un modelo más sencillo que el de una aplicación ISAPI.

Una petición HTTP llega al servidor y éste, tras analizarla sintácticamente, llega a la conclusión que debe ejecutar una aplicación CGI. Cuando el ejecutable se carga en memoria ocurre todo lo siguiente:

- 1 La unidad *CGLApp*, en su código de inicialización, crea un objeto de tipo *TCGLApplication* y lo asigna en la variable global *Application*, declarada en la unidad *WebBroker*.
- 2 La ejecución prosigue en el código de inicialización del proyecto, en el fichero *dpr*. Se llama entonces al método *Initialize* del objeto de aplicación, que se encarga de algunas tonterías necesarias para poder utilizar COM, de ser necesario.
- 3 Es importante saber lo que pasa ahora. Se ejecuta el método *CreateForm* de la clase *TCGLApplication*. Contrariamente a lo que sería de suponer, no se crea un formulario en este paso, como sucedería en una aplicación tradicional. Por el contrario, lo único que hace el objeto *Application* es sacar una copia del tipo de clase a la que pertenece el módulo Web que hemos diseñado. El segundo parámetro es ignorado, simplemente.
- 4 Es el método *Run* de *TCGLApplication* el que crea una instancia del módulo Web y pide a éste que responda a la petición HTTP. Luego, destruye la instancia y termina.

Recuerde que en una aplicación tradicional el método *Run* contiene la implementación del bucle de mensajes. Aquí, en Internet, las aplicaciones son efímeras, y no hay bucles que las mantengan con vida.

En sí mismas, las aplicaciones CGI son predecibles y no muy interesantes, y podríamos obviar los escabrosos detalles que acabo de contar. ¡Ah!, pero las ISAPI son otro tipo de nenas, y me he detenido en estas minucias para que pueda comparar ambos algoritmos. Supongamos que una petición HTTP convence a Internet Information Server que debe ejecutar una extensión ISAPI. Entonces:

- 1 El servidor examina su espacio de procesos, a ver si ha cargado la DLL necesaria. Si no lo ha hecho, se produce la carga del módulo, y los pasos numerados del 2 al 5 se ejecutan por primera y única vez. Si la DLL ya ha sido cargada, vamos directamente al paso 6.
- 2 Al inicializar la DLL por primera vez se ejecutará el código de inicialización de la unidad *ISAPLApp*, que crea una instancia de la clase *TISAPIApplication* y la asigna a la variable global *Application*.

- 3 La llamada a *Initialize* sigue siendo inocua.
- 4 Una vez más, *CreateForm* se limita a registrar el tipo de clase, y tampoco crea una instancia del módulo.
- 5 La implementación de *Run* no hace casi nada. Y termina enseguida. Pero tranquilícese; recuerde que estamos hablando de un módulo dinámico que se queda cargado en memoria.
- 6 Cuando IIS recibe la petición y comprueba que la DLL se encuentra cargada, llama a su función global *HttpExtensionProc*. A través de una serie de métodos de la clase *Application*, llegamos a la función *ActivateWebModule* (¡mucha atención!) que nos devuelve un puntero a una instancia del módulo Web.
- 7 La petición se pone a disposición del módulo activado, se genera la correspondiente respuesta y se desactiva el módulo mediante una llamada a *DeactivateWebModule*.

Y, como podrá sospechar, hay grandes dosis de intriga y misterio en la “activación” y “desactivación” de módulos Web, pero las despejaremos en la siguiente sección.

## La caché de módulos

Quiero advertir antes que todo lo que voy a escribir en esta sección atañe solamente a las aplicaciones ISAPI (también a las NSAPI). Repasemos un par de hechos:

- Internet Information Server carga, como máximo, una instancia de la DLL que implementa la extensión.
- Todas las peticiones que recibe el servidor son tratadas por esa única instancia. Ahora aclaro que el tratamiento de estas peticiones posiblemente concurrentes tiene lugar en hilos separados, y que es responsabilidad del servidor HTTP (no de Delphi) la creación y gestión de esos hilos.
- Por lo tanto, existe un único segmento de datos común a todas las peticiones recibidas en un servidor, y todas ellas comparten las mismas variables globales. En particular, el objeto asociado a *Application* es también común a todas ellas.

Veamos entonces cómo transcurre la activación de un módulo, separando los distintos casos que pueden presentarse:

- Primer caso: es la primera vez que el servidor procesa una petición ISAPI. En tal caso, se crea un nuevo módulo Web a partir de la clase registrada con *CreateForm*. Después de cumplir su tarea, el módulo no se destruye al ser desactivado, sino que va a parar a una lista interna de módulos inactivos.
- Segundo caso: mientras el primer módulo se encuentra ocupado procesando su respuesta, llega una segunda petición. IIS crea un nuevo hilo como siempre hace para una nueva petición. Pero Delphi no tiene módulos inactivos todavía. Entonces se crea una segunda instancia del módulo Web.

- Tercer caso: la primera petición ha terminado y ya está en la lista de módulos inactivos. La segunda petición sigue con su fiesta. Llega una tercera y Delphi saca al primer módulo (el inactivo) de su letargo. No se crean instancias nuevas, y la nueva petición debe tener mucho cuidado, porque el módulo reactivado conserva sus variables de instancia en el mismo estado en que las dejó al finalizar su respuesta.
- Cuarto caso: se ha producido una inesperada ráfaga de peticiones. No hay módulos inactivos, y ya hay 32 módulos danzando en el centro de la pista. Llega el trigésimo tercer invitado. Delphi, nuestro portero de discoteca, lanza una excepción y se niega a dar una respuesta amable: el usuario al otro lado de Internet recibe un áspero mensaje tipo *"El servidor está muuuuy ocupado"*. El resto de las peticiones siguen bailando despreocupadamente.

Espero que al leer la lista anterior se hayan encendido dos lucecitas diferentes de alarma en su sistema nervioso central. La primera: ¿por qué treinta y dos? ¿Es un número mágico? No se preocupe. Por una parte, 32 peticiones simultáneas es algo horrible pero poco probable... a no ser que su aplicación tarde un siglo en contestar una petición<sup>34</sup>. Y si ve que este valor se le hace pequeño, por otra parte, puede cambiarlo en el código fuente del proyecto:

```
begin
  Application.Initialize;
  Application.MaxConnections := 64;
  Application.CreateForm(TWebModule1, WebModule1);
  Application.Run;
end.
```

La otra luz de alarma: resulta que nuestra conciencia ecológica nos obliga ahora también a reciclar módulos de datos usados, gastados, raídos, vapuleados por la vida. Sí, es un peligro, qué quiere que le diga, pero este reciclaje nos abre las puertas del Paraíso.

Veamos primero en qué consiste el problema. Suponga que su aplicación utiliza en algún momento una consulta con parámetros. Usted asigna los parámetros, abre la consulta, la usa y no la cierra. Usted confía en que al terminar la consulta Delphi destruirá el módulo de datos. Y eso es cierto, si se trata de una CGI. Pero si convierte su aplicación al modelo ISAPI, la consulta se quedará abierta (recursos desperdiciados). Peor aún, si el módulo vuelve a ser utilizado, como es probable, la asignación de parámetros y la reapertura de la consulta no tendrán efecto alguno. Y este tipo de error es muy difícil de encontrar.

Ahora las buenas noticias. Si usted no comete la tontería de cerrar la base de datos al terminar la respuesta, la próxima vez que se use el módulo habrá ahorrado el precioso tiempo necesario para establecer la conexión. Esta técnica se conoce en inglés

---

<sup>34</sup> Como un funcionario de un Ministerio, digámoslo claramente.

como *database pooling*. Es posible diseñar algoritmos de *pooling* más complicados que el que acabamos de describir, pero el existente cumple dignamente su función.

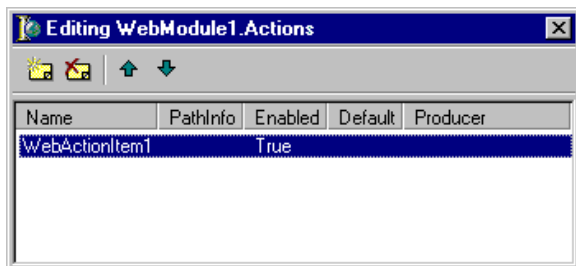
En resumen, si quiere conservar la cordura y, a la vez, desarrollar aplicaciones para Internet medianamente decentes, observe los siguientes mandamientos:

- I. Serás ambicioso como una urraca. Que tus aplicaciones CGI estén listas para pasar al modelo ISAPI en cuanto puedas.
- II. No te comportarás como un cerdo. Si abres una consulta, asegúrala de cerrarla. Si creas objetos auxiliares, asegúrate de liberarlos.
- III. Serás astuto como una víbora. El único recurso que no liberarás al terminar con él será el principal: la conexión a la base de datos.
- IV. Serás más precavido que una hormiga. No amontones tus inicializaciones en la respuesta al método *OnCreate* del módulo Web, pues ésta solamente se ejecutará una vez. Por el contrario, utilizarás el evento *BeforeDispatch* (que veremos más adelante) para tus inicializaciones. Y serán largos tus días sobre la faz de la Tierra<sup>35</sup>.

## Acciones

Cada vez que un navegante solicita un documento dinámico a una extensión de servidor, se ejecuta la correspondiente aplicación CGI, o la función diseñada para este propósito de la DLL ISAPI ó NSAPI. Sin embargo, en cada caso los parámetros de la petición contenidos en la URL pueden ser distintos. Ya he mencionado que el primer parámetro verdadero contenido en la URL va a continuación del nombre del módulo, separado por una barra inclinada, y se conoce como *información de camino*, o *path info*. Para discriminar fácilmente entre los valores de este parámetro especial los módulos Web de Delphi ofrecen las *acciones*.

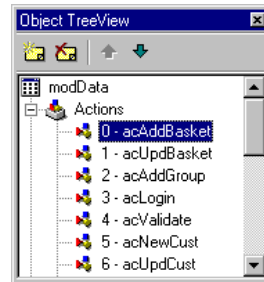
Tanto *TWebModule* como *TWebDispatcher* contienen una propiedad *Actions*, que es una colección de elementos de tipo *TWebActionItem*. La siguiente imagen muestra el editor de la propiedad:



<sup>35</sup> ... si no haces el borrico, eso es.



Es más fácil gestionar la lista de acciones de un módulo Web mediante el árbol de objetos (SHIFT+ALT+F11):



Cada objeto *WebActionItem* de esta colección posee las siguientes propiedades:

Propiedad	Significado
<i>PathInfo</i>	El nombre de “ruta” especificado en la URL, después del nombre de la aplicación
<i>MethodType</i>	El tipo de petición ante la cual reacciona el elemento
<i>Default</i>	Si es <i>True</i> , se dispara si no se encuentra una acción más apropiada
<i>Enabled</i>	Indica si la acción está activa o no
<i>Producer</i>	Un componente asociado para generar páginas HTML

Estos objetos tienen un único evento, *OnAction*, que es el que se dispara cuando el módulo Web determina que la acción es aplicable, y su tipo es éste:

```

type
  THTTPMethodEvent = procedure (Sender: TObject;
    Request: TWebRequest; Response: TWebResponse;
    var Handled: Boolean) of object;

```

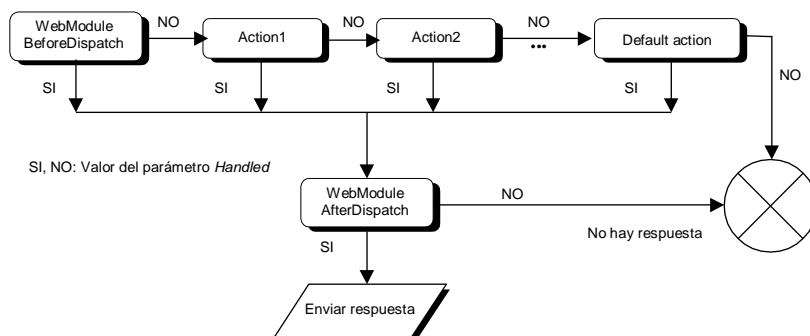
Toda la información acerca de la petición realizada por el cliente viene en el parámetro *Request* del evento, que analizaremos en una próxima sección. El propósito principal de los manejadores de este evento es asignar total o parcialmente el texto HTML que debe enviarse al cliente como respuesta, en el parámetro *Response*. Digo parcialmente porque para una misma petición pueden dispararse varias acciones en cascada, como veremos enseguida; cada acción puede construir una sección de la página HTML final. Pero, además, en la respuesta al evento asociado a la acción se pueden provocar efectos secundarios, como grabar o modificar un registro en una base de datos, alterar variables del módulo Web, y cosas parecidas.

Supongamos que el cliente pide la siguiente URL:

```
http://www.wet_wild_woods.com/scripts/buscar.dll/clientes
```

La información de ruta viene a continuación del nombre de la DLL: *clientes*. Cuando nuestra aplicación recibe la petición, busca todas las acciones que tienen asignada

esta cadena en su propiedad *PathInfo*. El siguiente diagrama muestra la secuencia de disparo de las distintas acciones dentro del módulo:



Antes de comenzar a recorrer las posibles acciones, el módulo Web dispara su propio evento *BeforeDispatch*, cuyo tipo es idéntico al de *OnAction*. Así el módulo tiene la oportunidad de preparar las condiciones para la cadena de acciones que puede dispararse a continuación. El evento utiliza un parámetro *Handled*, de tipo lógico. Si se le asigna *True* a este parámetro y se asigna una respuesta a *Response* (paciencia, ya contaré cómo), no llegarán a dispararse el resto de las acciones del módulo, y se devolverá directamente la respuesta asignada.

En caso contrario, el módulo explora secuencialmente todas las acciones cuya propiedad *PathInfo* sea igual a *clientes*, ya que pueden existir varias. Para que la cadena de disparos no se interrumpa, cada acción debe dejar el valor *False* en el parámetro *Handled* del evento. Si después de probar todas las acciones que corresponden por el valor almacenado en su *PathInfo*, ninguna ha marcado *Handled* como verdadera (o no se ha encontrado ninguna), se trata de ejecutar aquella acción que tenga su propiedad *Default* igual a *True*, si es que hay alguna.

Al finalizar todo este proceso, y si alguna acción ha marcado como manejada la petición, se ejecuta el evento *AfterDispatch* del módulo Web.

## Generadores de contenido

El propósito de los manejadores de eventos de acciones es, principalmente, generar el contenido o parte del contenido de una página HTML. Esto se realiza asignando el texto correspondiente a la página en la propiedad *Content* del parámetro *Response* del evento. Por ejemplo:

```

procedure TWebModule1.WebModule1WebActionItem1Action(
    Sender: TObject; Request: TWebRequest; Response: TWebResponse;
    var Handled: Boolean);
begin
    Response.Content := '<HTML><BODY>;Hola, colega!</BODY></HTML>';
end;
  
```

Por supuesto, siempre podríamos asignar una larga cadena generada por código a esta propiedad, siempre que contenga una página HTML válida. Pero vemos que, incluso en el sencillo ejemplo anterior, ésta es una tarea pesada y es fácil cometer errores de sintaxis. Por lo tanto, Delphi nos ofrece varios componentes en la página *Internet* para facilitar la generación de texto HTML. Para comenzar, el componente *TPageProducer* tiene las siguientes propiedades relevantes:

Propiedad	Significado
<i>Dispatcher</i>	El módulo en que se encuentra, o el componente <i>TWebDispatcher</i> al que se asocia
<i>HTMLDoc</i>	Lista de cadenas que contiene texto HTML
<i>HTMLFile</i>	Alternativamente, un fichero con texto HTML

La idea es que *HTMLDoc* contenga el texto a generar por el componente, de forma tal que este texto se especifica en tiempo de diseño y se guarda en el fichero *dfm* del módulo, para que no se mezcle con el código. Pero también puede especificarse un fichero externo en *HTMLFile*, para que pueda modificarse el texto generado sin necesidad de tocar el ejecutable de la aplicación.

### ¿DOCUMENTO O FICHERO?

En una aplicación profesional debemos utilizar, sin duda alguna, la propiedad *HTMLFile*. Pero debemos tener mucho cuidado con la ruta que utilicemos en el nombre del fichero. Mi truco favorito consiste en modificar la ruta dinámicamente de acuerdo a la ubicación donde se encuentre la aplicación, llamando a la función *GetModuleName* del API de Windows. Así la aplicación puede cambiar de directorio sin que las propiedades *HTMLFile* dejen de ser correctas.

En cualquier caso, el ejemplo de respuesta al evento *OnAction* se escribiría así:

```
procedure TWebModule1.WebModule1WebActionItem1Action(
    Sender: TObject; Request: TWebRequest; Response: TWebResponse;
    var Handled: Boolean);
begin
    Response.Content := PageProducer1.Content;
end;
```

Ahora bien, en una aplicación con decenas de acciones sería muy laborioso tener que crear manejadores de eventos para cada una de ellas si el código de los mismos es tan simple como el que hemos mostrado. Para evitarnos la molestia, a partir de Delphi 5 cada acción tiene una propiedad *Producer*, que podemos hacer que apunte a alguno de los generadores de contenido existentes en el módulo. Si asignamos algo a *Producer*, cuando se detecta que la acción es aplicable a la petición en curso se produce la siguiente secuencia de sucesos:

- Se asigna automáticamente a *Response.Content* la página generada por el componente asociado a *Producer*.

- Si además hemos asignado un manejador de evento a *OnAction*, se ejecuta. Así podemos retocar la respuesta, llevar un registro de disparos de la acción o cualquier otra cosa que se nos ocurra.

## Etiquetas transparentes

No obstante, si nos limitamos al tipo de generación de contenido que acabamos de describir, las páginas producidas por nuestra extensión de servidor siempre serán páginas estáticas. La principal ventaja del uso de *TPageProducer* es que podemos realizar la sustitución dinámica de *etiquetas transparentes* por texto. Una etiqueta transparente es una etiqueta HTML cuyo primer carácter es la almohadilla: #. Estas etiquetas no pertenecen en realidad al lenguaje, y son ignoradas por los navegadores. En el siguiente ejemplo, el propósito de la etiqueta `<#HORA>` es el de servir como comodín para ser sustituido por la hora actual:

```
<HTML><BODY>
¡Hola, colega! ¿Qué haces por aquí a las <#HORA>?
</BODY></HTML>
```

¿Por qué se sustituye la etiqueta transparente anterior por la hora? ¿Acaso hay algún mecanismo automático que detecte el nombre de la etiqueta y ...? No, por supuesto. Cuando utilizamos la propiedad *Content* del productor de páginas, estamos iniciando en realidad un algoritmo en el que nuestro componente va examinando el texto HTML que le hemos suministrado, va detectando las etiquetas transparentes y, para cada una de ellas, dispara un evento durante el cual tenemos la posibilidad de indicar la cadena que la sustituirá. Este es el evento *OnHTMLTag*, y el siguiente ejemplo muestra sus parámetros:

```
procedure TmodWebData.PageProducer1HTMLTag(Sender: TObject;
  Tag: TTag; const TagString: string; TagParams: TStrings;
  var ReplaceText: string);
begin
  if TagString = 'HORA' then
    ReplaceText := TimeToStr(Now);
end;
```

Hay que tener un poco de cuidado con las mayúsculas y minúsculas en el nombre de las etiquetas. La forma más segura de programar el evento anterior es la siguiente:

```
if CompareText(TagString, 'HORA') = 0 then
  ReplaceText := TimeToStr(Now);
```

Si la plantilla que estamos utilizando contiene muchas etiquetas transparentes diferentes, la discriminación de la etiqueta en el evento *OnHTMLTag* puede tardar lo suyo, en términos relativos, claro está. Por este motivo Borland definió seis etiquetas predefinidas, asociándoles valores enumerativos. Con sólo comprobar el valor del parámetro *Tag* podemos saber si Delphi ha encontrado alguna de las etiquetas especiales:

Valor de Tag	Valor de TagString	Debe generar...
<i>tgLink</i>	LINK	Un enlace
<i>tgImage</i>	IMAGE	Una imagen
<i>tgTable</i>	TABLE	Una tabla
<i>tgImageMap</i>	IMAGEMAP	Una imagen con “puntos calientes”
<i>tgObject</i>	OBJECT	Un control ActiveX
<i>tgEmbed</i>	EMBED	Un <i>plug-in</i> de Netscape
<i>tgCustom</i>	Cualquier otro	Lo que nos apetezca

Si vamos a ser sinceros, debo confesar que casi nunca utilizo el valor de *Tag*, porque no sigo el convenio anterior de nombres de etiquetas.

## Etiquetas con parámetros

Aún podemos ganar más control y flexibilidad sobre la generación de contenido si utilizamos parámetros dentro de las etiquetas transparentes. Tomemos como ejemplo la etiqueta que nos hemos inventado: *HORA*. En la primera versión de la página de bienvenida hicimos que se expandiese para mostrar la hora utilizando el formato predeterminado por la función *TimeToStr*. Más adelante, sin embargo, al cliente que nos ha encargado la página se le pueden ocurrir ideas perversas tales mostrar también la fecha, o el día de la semana. ¿Qué hacemos, modificamos la aplicación o matamos al cliente?

A pesar de lo atractiva que nos resulte la segunda opción, podemos evitar sus consecuencias judiciales incluyendo un parámetro dentro de la etiqueta, al que arbitrariamente llamaremos *FMT*. El diseñador de páginas puede modificar el texto del documento de la siguiente manera:

```
<HTML><BODY>
¡Querido cliente potencial!

Hoy es <#HORA FMT="dddd">, y son las <#HORA FMT="hh:nn">.
Es un buen momento para que usted derroche su pasta
comprando mis inútiles baratijas.
</BODY></HTML>
```

Entonces, para expandir correctamente cada una de las apariciones de la etiqueta, sustituimos la respuesta del evento *OnHTMLTag* por esta otra:

```
procedure TmodWebData.PageProducer1HTMLTag(Sender: TObject;
  Tag: TTag; const TagString: string; TagParams: TStrings;
  var ReplaceText: string);
begin
  if TagString = 'HORA' then
    ReplaceText := FormatDateTime(TagParams.Values['FMT'], Now);
end;
```

Incluso si el diseñador olvida la existencia del parámetro *FMT* no ocurre ningún desastre. En tal situación el valor del parámetro es una cadena vacía. La función *For-*

*matDateTime* la acepta como cadena de formato correcta, y devuelve una fecha en formato “corto” (día/mes/año de dos dígitos) más la hora en su formato tradicional.

## ¿Etiquetas anidadas?

Durante mucho tiempo fui víctima de un error personal: basándome en la similitud entre las etiquetas transparentes del Web Broker de Delphi y las etiquetas normales de HTML, supuse que no era posible situar una etiqueta transparente dentro de otra etiqueta. Para que nos entendamos, HTML no permite este tipo de construcciones:

```
<!-- INCORRECTO -->
<etiqueta1 attr1="val1" attr2="val2" <etiqueta2>>
```

Parece algo evidente porque, además, es difícil encontrarle algún sentido a incluir una etiqueta dentro de otra. Quiero que sepa que el siguiente tipo de anidamiento es también incorrecto; la diferencia respecto al ejemplo anterior es que ahora ambas etiquetas son del tipo empleado por Delphi:

```
<!-- NO SE PUEDE -->
<#etiqueta1 attr1="val1" attr2="val2" <#etiqueta2>>
```

Sin embargo, la siguiente construcción *sí* es aceptada por los componentes generadores de código HTML de Delphi:

```
<!-- PERFECTAMENTE POSIBLE (y además muy útil) -->
<etiqueta1 attr1="val1" attr2="val2" <#etiqueta2>>
```

En este caso, la etiqueta transparente se sitúa inesperadamente dentro de una etiqueta HTML. Tuve que leer el código fuente de los componentes Web Broker para darme cuenta de esta posibilidad.

¿Qué utilidad tiene encajar una etiqueta transparente dentro de una etiqueta normal? Pongamos un primer ejemplo muy sencillo: su página Web debe hacer referencia a cierta ruta dentro de cierto dominio para incluir imágenes. Usted desarrolla en su empresa, y utiliza el servidor situado en la máquina *Christine*<sup>36</sup>. Por lo tanto, su plantilla HTML se refiere a las imágenes más o menos de este modo:

```

```

Desgraciadamente, como el usuario final posee un dominio completamente diferente, es una locura incluir URLs y rutas absolutas dentro de las plantillas, por el imaginable problema del mantenimiento de las mismas. Lo mismo sucede con otras etiquetas que utilizan URLs, especialmente con los enlaces <A> y formularios <FORM>.

---

<sup>36</sup> Si hay algún psiquiatra cerca, que escuche mi historia. Suelo darle nombres de chica a mis ordenadores “fijos”: Naroa, Christine, Michelle, etc. Sin embargo, casi siempre me identifico con mis portátiles: Ian, Lonewolf (lobo solitario) ...

Tenemos una solución algo miope al alcance de la mano: habilitar etiquetas transparentes que generen etiquetas de imágenes, de enlaces y de formularios. Estas nuevas etiquetas necesitarán parámetros. Por ejemplo, usted puede incluir en su página esta etiqueta, y hacer que en tiempo de ejecución se añada el nombre del dominio adecuado a la URL que apunta al fichero de imagen:

```
<#LOCALIMAGE SRC="thumb-up.gif">
```

Lo malo de esta técnica es que nos obliga a crear varios tipos de etiquetas transparentes: para los enlaces y los formularios, por ejemplo. Una solución más flexible es poder utilizar una etiqueta personalizada que directamente se expanda en el nombre del dominio donde se ejecuta la aplicación:

```

```

Tome nota de que la nueva etiqueta se ha introducido incluso dentro de las dobles comillas del atributo.

## Propiedades de una petición

Como ya sabemos generar código HTML en respuesta a una acción, aprovechemos para demostrar en directo las propiedades más útiles contenidas en una petición que llega al servidor, es decir, en un objeto de tipo *TWebRequest*. He aquí algunas de las más importantes:

Propiedad	Significado
<i>Accept</i>	Formatos de contenidos aceptados por el navegador
<i>UserAgent</i>	Información sobre el navegador
<i>RemoteAddr</i>	Dirección IP del cliente
<i>From</i>	El correo electrónico del cliente, si lo ha registrado
<i>Referer</i>	En qué página se produjo la petición

Vamos a crear una nueva aplicación CGI, y le daremos como nombre *WebRequest*. Añada en primer lugar un componente de tipo *TPageProducer* al módulo Web, cambie su nombre a *ppInfo* y teclee dentro de su propiedad *HTMLDoc* el texto mostrado a continuación:

```
<style>
  TD { font: 10pt verdana, arial, sans serif;
        background-color: silver }
  TD.PROP { font-weight: bold; text-align: right;
        background-color: white }
</style>

<table>
  <tr><td class=prop>From</td><td><#FROM></td></tr>
  <tr><td class=prop>Referer</td><td><#REFERER></td></tr>
  <tr><td class=prop>Remote address</td><td><#REMOTEADDR></td></tr>
  <tr><td class=prop>User agent</td><td><#USERAGENT></td></tr>
```

```

<tr><td class=prop>Accept</td><td><#ACCEPT></td></tr>
<tr><td class=prop>Protocol version</td>
  <td><#PROTOCOLVERSION></td></tr>
<tr><td class=prop>Method</td><td><#METHOD></td></tr>
<tr><td class=prop>If modified since</td>
  <td><#IFMODIFIEDSINCE FMT="dd/mm/yyyy hh:nn:ss"></td></tr>
</table>

<a href="http://naroa/scripts/webrequest.exe/info">
  Más información...</a>

```

La inclusión de una hoja de estilo sólo pretende hacer más agradable el aspecto del resultado. Observe que he incluido un enlace a la propia página para probar el funcionamiento de la propiedad *Referer*.

El siguiente paso es dar respuesta al evento *OnHTMLTag* de *ppInfo*. Inevitablemente, la longitud del método será considerable, y puede aumentar si usted completa las etiquetas y propiedades que he omitido a propósito:

```

procedure TwebData.ppInfoHTMLTag(Sender: TObject;
  Tag: TTag; const TagString: string; TagParams: TStrings;
  var ReplaceText: string);
begin
  if SameText(TagString, 'FROM') then
    ReplaceText := Request.From
  else if SameText(TagString, 'REFERER') then
    ReplaceText := Request.Referer
  else if SameText(TagString, 'REMOTEADDR') then
    ReplaceText := Request.RemoteAddr
  else if SameText(TagString, 'REMOTEHOST') then
    ReplaceText := Request.RemoteHost
  else if SameText(TagString, 'USERAGENT') then
    ReplaceText := Request.UserAgent
  else if SameText(TagString, 'ACCEPT') then
    ReplaceText := Request.Accept
  else if SameText(TagString, 'PROTOCOLVERSION') then
    ReplaceText := Request.ProtocolVersion
  else if SameText(TagString, 'METHOD') then
    ReplaceText := Request.Method
  else if SameText(TagString, 'IFMODIFIEDSINCE') then
    ReplaceText := FormatDateTime(TagParams.Values['FMT'],
      Request.IfModifiedSince);
end;

```

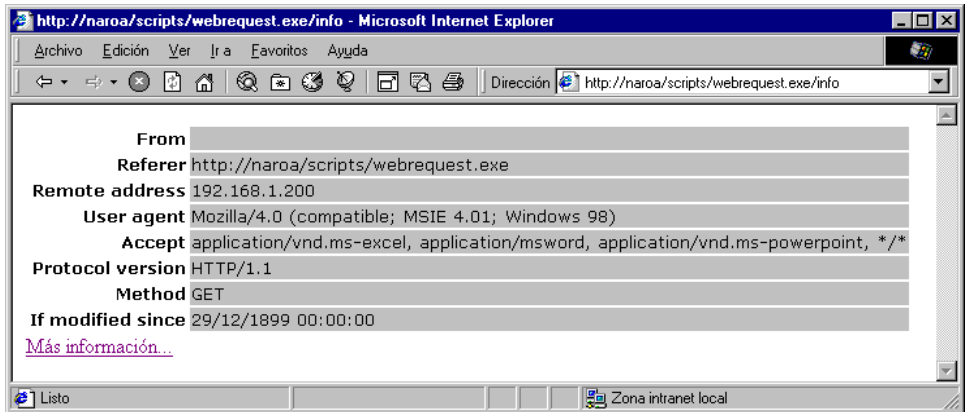
El método hace referencia a cierta variable *Request*, que sin embargo no se pasa como parámetro, tal como sucede con el evento *OnAction*. La explicación es que *Request* es también una propiedad del módulo Web, una decisión de diseño muy sensata que nos evita tener que propagar valores desde la respuesta a *OnAction* a otros eventos.

Si ha transformado un módulo de datos “normal” en un módulo Web, tendrá que buscar la propiedad *Request* en el componente *WebDispatcher* que ha añadido al módulo.

Por último, añada una acción al módulo Web, asigne *info* en su propiedad *PathInfo*, y cambie el valor de *Producir* para que apunte a *ppInfo*. Compile la aplicación y muévela



al directorio *scripts* del directorio raíz de su servidor HTTP. Cuando pidamos la ejecución de esta aplicación desde Internet Explorer 4 “punto algo”, obtendremos:



Preste atención sobre todo al valor de *UserAgent*. Podemos programar un método en nuestro módulo para saber si estamos lidiando con el navegador de Microsoft o el de Netscape<sup>37</sup>:

```
function TwebData.IsMicrosoft: Boolean;
begin
    Result := Pos('MSIE', Request.UserAgent) <> 0;
end;
```

## Recuperación de parámetros

¿Dónde vienen los parámetros de la petición? Evidentemente, en las propiedades del parámetro *Request* del evento asociado a la acción. La propiedad concreta en la que tenemos que buscarlos depende del tipo de acción que recibimos. Claro está, si no conocemos qué tipo de método corresponde a la acción, tenemos que verificar la propiedad *MethodType* de la petición, que pertenece al siguiente tipo enumerativo:

```
type
    TMethodType = (mtAny, mtGet, mtPut, mtPost, mtHead);
```

Estamos interesados principalmente en los tipos *mtGet* y *mtPost*. El tipo *mtAny* representa un comodín, incluso para otros tipos de métodos menos comunes no incluidos en *TMethodType*, como *OPTIONS*, *DELETE* y *TRACE*; en estos casos, hay que mirar también la propiedad *Method*, que contiene la descripción literal del método empleado.

Supongamos ahora que el método sea *GET*. La cadena con todos los parámetros viene en la propiedad *Query* de *Request*. Por ejemplo:

URL:      `http://www.WetWildWoods.com/find.dll/animal?kind=cat&walk=alone`

<sup>37</sup> En estos momentos podemos ignorar la posibilidad de que se trate de otro navegador.

*Query:*     *kind=cat&walk=alone*

Pero más fácil es examinar los parámetros individualmente, mediante la propiedad *QueryFields*, de tipo *TStrings*. En el siguiente ejemplo se aprovecha la propiedad vectorial *Values* de esta clase para aislar el nombre del parámetro del valor:

```
procedure TWebModule1.WebModule1WebActionItem1Action(
  Sender: TObject; Request: TWebRequest; Response: TWebResponse;
  var Handled: Boolean);
begin
  if Request.QueryFields.Values['kind'] = 'cat' then
    // ... están preguntando por un gato ...
  end;
```

Por el contrario, si el método es POST los parámetros vienen en las propiedades *Content*, que contiene todos los parámetros sin descomponer, y en *ContentFields*, que es análoga a *QueryFields*. Si el programador no quiere depender del tipo particular de método de la acción para el análisis de parámetros, puede utilizar uno de los procedimientos auxiliares *ExtractContentFields* ó *ExtractQueryFields*:

```
procedure TWebModule1.WebModule1WebActionItem1Action(
  Sender: TObject; Request: TWebRequest; Response: TWebResponse;
  var Handled: Boolean);
begin
  if Request.MethodType = mtPost then
    Request.ExtractContentFields(Request.QueryFields);
  if Request.QueryFields.Values['kind'] = 'cat' then
    // ... están preguntando por un gato ...
  end;
```

## Generadores de tablas

Un caso particular de generadores de contenido son los generadores de tablas que incorpora Delphi: *TDataSetTableProducer*, y *TQueryTableProducer*. Ambos son muy parecidos, pues descienden de la clase abstracta *TDSTableProducer*, y generan una tabla HTML a partir de los datos contenidos en un conjunto de datos de la VCL.

*TQueryTableProducer* se diferencia en que el conjunto de datos debe ser obligatoriamente una consulta, y en que esta consulta puede extraer sus parámetros de los parámetros de la petición HTTP en curso, ya sea a partir de *QueryFields*, en el caso de acciones GET, o de *ContentFields*, en el caso de POST.

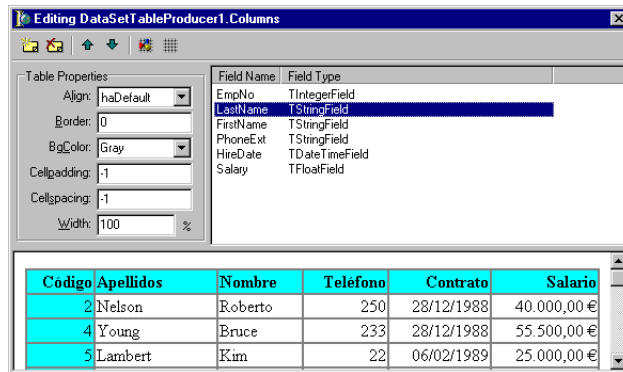
Las principales propiedades comunes a estos componentes son:

Propiedad	Significado
<i>DataSet</i>	El conjunto de datos asociado
<i>MaxRows</i>	Número máximo de filas de datos generadas
<i>Caption, Caption.Alignment</i>	Permite añadir un título a la tabla
<i>Header, Footer</i>	Para colocar texto antes y después de la tabla
<i>Columns, Row.Attributes,</i>	Formato de la tabla

Propiedad	Significado
-----------	-------------

<i>Column.Attributes</i>	
--------------------------	--

Para dar formato a la tabla, es conveniente utilizar el editor asociado a la propiedad *Columns* de estos componentes. La siguiente imagen muestra el aspecto del editor de columnas del productor de tablas, que permite personalizar columna por columna el aspecto del código generado:



Para ambos componentes, la función *Content* genera el texto HTML correspondiente. En el caso de *TQueryTableProducer*, la propia función se encarga de extraer los parámetros de la solicitud activa, y de asignarlos al objeto *TQuery* asociado.

El evento *OnFormatCell*, común a ambos componentes, es muy útil, porque permite retocar el contenido de cada celda de la tabla generada. Podemos cambiar colores, alineación, tipos de letras, e incluso sustituir completamente el contenido. Por ejemplo, el siguiente ejemplo muestra cómo se pueden generar cuadros de edición en la primera columna de una tabla:

```
procedure TForm1.DataSetTableProducer1FormatCell (
  Sender: TObject; CellRow, CellColumn: Integer;
  var BgColor: THTMLBgColor; var Align: THTMLAlign;
  var VAlign: THTMLVAlign; var CustomAttrs, CellData: string);
begin
  if (CellRow > 0) and (CellColumn = 0) then
    CellData := '<INPUT TYPE=TEXT VALUE=' +
      AnsiQuotedStr(CellData, '''') + '>';
end;
```

## No sólo de HTML vive la Web

Un error bastante frecuente es pensar que la respuesta a una petición HTTP, o a una acción, según el punto de vista de un programador de aplicaciones CGI, solamente puede consistir en contenido HTML.

Supongamos que hemos definido una tabla *Banners* con el siguiente esquema (utilizo el dialecto de InterBase), y que el contenido del campo *Imagen* es precisamente una imagen en formato JPEG:

```
create table BANNERS (
    IDBanner integer not null primary key,
    URL       varchar(127) not null unique,
    Imagen    blob
);
```

Para simplificar, asumiremos que la columna *IDBanner* contiene valores enteros sucesivos, contando desde cero. No es muy complicado mantener esta secuencia si tras cualquier inserción o borrado en la tabla ejecutamos un procedimiento de enumeración. Entonces el siguiente método de respuesta a una acción muestra una de las imágenes de la tabla, suponiendo que se recibe el código asociado a la misma en uno de los parámetros de la petición:

```
procedure TWebModule1.acBannerAction(Sender: TObject;
    Request: TWebRequest; Response: TWebResponse;
    var Handled: Boolean);
begin
    qrCount.Open;
    try
        qrBanner.ParamByName('BannerID').AsInteger :=
            Random(qrCount.Fields[0].AsInteger);
    finally
        qrCount.Close;
    end;
    qrBanner.Open;
    try
        Response.ContentType := 'image/jpeg';
        Response.ContentStream :=
            qrBanner.CreateBlobStream(qrBannerImage, bmRead);
    finally
        qrBanner.Close;
    end;
end;
```

El ejemplo utiliza dos consultas, *qrCount* y *qrImage*, basadas en estas dos instrucciones de selección:

```
select count(*) from BANNERS

select Image from BANNERS where BannerID = :BannerID
```

En primer lugar, la respuesta al evento abre una consulta para recuperar el número de imágenes que hay en la tabla. Sin pausa alguna, pasa el valor obtenido por la función *Random* para obtener un identificador de fila aleatorio. Entonces recupera la fila en cuestión.

#### NOTA IMPORTANTE

Hay que tener mucho cuidado con la función *Random*. Esta función genera una sucesión de números de apariencia aleatoria a partir de un valor inicial, o *semilla*. El caso

es que si no tomamos medidas especiales, el valor de la semilla es siempre el mismo. Si, por ejemplo, esta respuesta se incluye dentro de una aplicación CGI, cada vez que se ejecute obtendremos inevitablemente la misma imagen. La solución está en modificar la semilla al principio del programa, llamando al método *Randomize* en la inicialización de la unidad del módulo.

Ahora viene la parte importante: se modifica el valor de la propiedad *Content/Type* de la respuesta. En el ejemplo anterior he asumido que en *Imagen* se almacena una representación binaria directa en formato JPEG. Si queremos soportar otros formatos, podemos añadir otra columna a la tabla que indique el tipo de contenido almacenado en *Imagen*.

A continuación se llama al método *CreateBlobStream* para crear y abrir un flujo de datos correspondiente al contenido del campo blob. El nuevo objeto se asigna a la propiedad *ContentStream* de la respuesta... ¡y tome nota de que no debemos liberar el objeto de flujo! De ello se encargará la propia respuesta cuando haya entregado su contenido y ella misma sea destruida.

## Redirección

La técnica recién presentada se utiliza con frecuencia para presentar publicidad rotativa en una página Web. En ese caso, la tabla *Banners* debería contener campos adicionales con información sobre la empresa que nos contrata para que mostremos su imagen, quizás la página Web de dicha empresa y, lo más importante, un contador que nos diga el número de veces que se ha mostrado la página a algún navegante. El contador de marras debería incrementarse en el mismo método que mostramos en el ejemplo anterior.

Ahora bien, en el mundo de la publicidad por Internet se distingue nítidamente entre el número de veces que se muestra un anuncio y el número de ocasiones en las que el anuncio ha sido efectivo... porque el navegante ha sentido curiosidad y ha pinchado la imagen insensatamente. Esta última estadística es fácilmente controlable por la empresa que nos contrata, porque en definitiva se trata de un salto a una página bajo su control, y en la información de referencia de la petición (*Referrer*) aparecerá la URL de nuestra propia página. Pero desconfiar es humano, y como no queremos nos timen, se nos ocurre llevar nuestra propia estadística.

Debemos comenzar por analizar el código HTML de la página que contiene la imagen publicitaria. Es normal que esta página madre sea generada por la misma aplicación CGI/ISAPI, pero aquí nos desentenderemos de esta parte y solamente nos ocuparemos del código final. He aquí un ejemplo de cómo podría diseñarse la referencia a la imagen:

```
<a href=" www.intsight.com">
</a>
```

Cuando el usuario pulsa sobre la imagen, en realidad está pulsando sobre el enlace que la rodea, que lo lleva directamente a la página de la otra empresa. Por lo tanto, cambiemos la URL del enlace del siguiente modo:

```
<a href=" www.marteens.com/redirect?url=www.intsight.com">
</a>
```

Ahora el enlace nos lleva cierta acción de nombre *redirect* dentro de nuestra aplicación, y el parámetro *url* nos indica a cuál página en realidad queremos saltar. La aplicación, en vez de entregar como respuesta una página HTML, simplemente debe instruir al navegador para que vaya a otra dirección. El truco es muy fácil de programar, como se puede ver en el siguiente método:

```
procedure TWebModule1.acRedirectAction(Sender: TObject;
    Request: TWebRequest; Response: TWebResponse;
    var Handled: Boolean);
var
    NewURL: string;
begin
    if Request.MethodType = mtPost then
        NewURL := Request.ContentFields.Values['url']
    else
        NewURL := Request.QueryFields.Values['url'];
    Response.SendRedirect(NewURL);
end;
```

## Mantenimiento del estado

**E**L PROBLEMA PRINCIPAL DE LOS SERVIDORES WEB ES su corta memoria. Le pides a uno de ellos: dame, por favor, la lista de los diez discos más vendidos, y el servidor te responderá con mil amores. Pero si se te ocurre preguntar por los diez que siguen, el servidor fruncirá una ceja: ¿y quién eres tú?

Evidentemente, no tiene sentido que el servidor recuerde la conversación que ha mantenido con nosotros. Después que leamos la página que nos ha enviado la primera vez como respuesta, es muy probable que apaguemos el navegador, o nos vayamos a navegar a otra parte. No merece la pena guardar memoria de los potenciales cientos o miles de usuarios que pueden conectarse diariamente a una página muy transitada.

### Información sobre el estado

Pero no hay problemas insolubles, sino preguntas mal planteadas. En nuestro ejemplo anterior, acerca de la lista de éxitos, podemos formular la petición de este otro modo: ¿cuáles son los discos de la lista que van desde el 11 al 20? O en esta otra forma: hola, soy Ian Marteens (conexión número 12345678), ¿cuáles son los próximos diez discos? En este último caso, por ejemplo, necesitamos que se cumplan estas dos condiciones:

- 1 El servidor debe asignar a cada conexión una identificación, del tipo que sea. Además, debe llevar en una base de datos un registro de las acciones realizadas por la “conexión”.
- 2 El usuario debe disponer a su vez de este identificador, lo que implica que el servidor debe pensar en algún método para comunicar este número al cliente.

Sigamos aclarando el asunto: observe que yo, Ian Marteens, puedo ser ahora la conexión 3448 para cierto servidor, pero al apagar el ordenador y volver a conectarme al día siguiente, recibiré el número 5237. Por supuesto, podemos idear algún mecanismo de identificación que nos permita recuperar nuestra última identificación, pero esta es una variación sencilla del mecanismo básico que estamos explicando.

¿Cómo puede comunicar el servidor al cliente su número de identificación? Planteémoslo de otra manera: ¿qué es lo que un servidor de Internet puede suministrar a un cliente? ¡Documentos HTML, qué diablos! Pues bien, introduzca traicioneramente el identificador de la conexión dentro del documento HTML que se envía como respuesta. Se supone que se recibe un número de conexión porque queremos seguir preguntando tonterías al servidor (bueno, ¿y qué?). Entonces es muy probable que el documento contenga un formulario, y que podamos utilizar un tipo especial de campo conocido como *campos ocultos* (*hidden fields*):

```
<HTML>
<HEAD><TITLE>Canciones más solicitadas</TITLE></HEAD>
<BODY>
  <H1>Lista de éxitos</H1>
  <H2>(del 1 al 5)</H2>
  <HR>
  <OL START=1><LI>Dust in the wind</LI>
    <LI>Stairway to heaven</LI>
    <LI>More than a feeling</LI>
    <LI>Wish you were here</LI>
    <LI>Macarena</LI>
  </OL><HR>
  <FORM METHOD=GET
    ACTION="http://www.marteens.com/scripts/prg.exe/hits">
    <INPUT TYPE=HIDDEN NAME=USER VALUE=1234>
    <INPUT TYPE=SUBMIT VALUE="Del 6 al 10">
  </FORM>
</BODY>
</HTML>
```

De todo el documento anterior, la cláusula que nos interesa es la siguiente:

```
<INPUT TYPE="HIDDEN" NAME="USER" VALUE="1234">
```

Esta cláusula no genera ningún efecto visual, pero como forma parte del cuerpo del formulario, contribuye a la generación de parámetros cuando se pulsa el botón de envío. El atributo NAME vale USER en este ejemplo, pero podemos utilizar un nombre arbitrario para el parámetro. Como el método del formulario es GET, la pulsación del botón solicita la siguiente URL:

```
http://www.marteens.com/scripts/prg.exe/hits?USER=1234
```

Cuando el servidor Web recibe esta petición puede buscar en una base de datos cuál ha sido el último rango de valores enviado al usuario 1234, generar la página con los próximos valores, actualizar la base de datos y enviar el resultado. Otra forma de abordar el asunto sería incluir el próximo rango de valores dentro de la página:

```
<INPUT TYPE=HIDDEN NAME=STARTFROM VALUE=6>
```

De esta forma, no es necesario que el servidor tenga que almacenar en una base de datos toda la actividad generada por una conexión. Este enfoque, no obstante, es mejor utilizarlo en casos sencillos como el que estamos exponiendo.



## NOTA

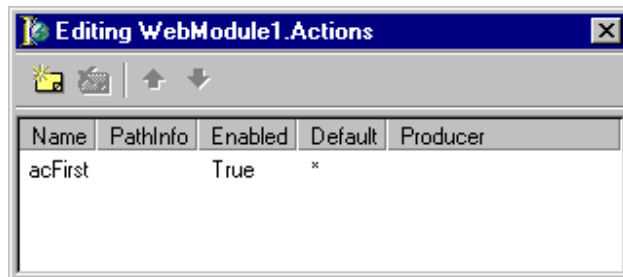
¿Recuerda el barullo que se produjo al salir el Pentium III al mercado? Cada uno de estos procesadores trae de fábrica un número interno y se garantiza que éste es único. Supongamos que se modifica el código de un navegador, o incluso el propio protocolo HTTP, para que junto a cada petición se envíe automáticamente al servidor la identificación del procesador. Está claro que así se puede simplificar enormemente el desarrollo de aplicaciones Web, pues esta técnica resuelve más de la mitad de los problemas asociados al mantenimiento del estado de la aplicación. Pero existe un lado oscuro: la identificación puede ser utilizada por los proveedores de contenido para fines no demasiado santos. Resultado: los Pentium III se vendieron con el código de marras inhabilitado.

## Un simple navegador

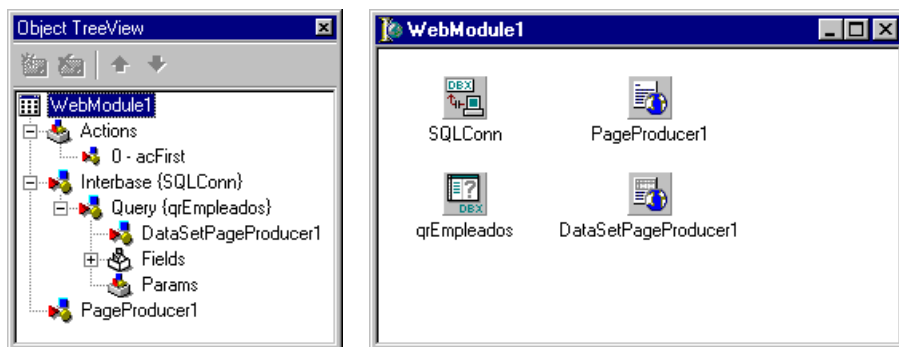
Es conveniente organizar todo el contenido que hemos expuesto mediante un ejemplo sencillo. ¿Qué tal si le muestro cómo navegar registro a registro sobre una tabla alcanzable desde el servidor HTTP? Mostrar un registro de una tabla en una página HTML será muy sencillo, gracias al componente *TDataSetPageProducer*. Lo más complicado será coordinar al cliente y al servidor, de modo que cuando el cliente pida el “próximo” registro, el servidor sepa de qué registro está hablando. La técnica que voy a emplear está inspirada en la forma en el que el BDE implementa la navegación sobre tablas cliente/servidor.

Iniciemos una nueva aplicación Web, mediante el asistente *Web Server Application* del Almacén de Objetos; da lo mismo el modelo que elija. Guarde el proyecto con el nombre de *WebBrowse*, y la unidad del módulo de datos como *Datos*.

Nuestra aplicación solamente implementará una acción, a la que no asociaremos nombre alguno. Para controlar el código HTML que genera el servidor utilizaremos un par de parámetros: *Direccion* y *Codigo*. En el primero indicaremos en qué dirección desea navegar el usuario: *First*, *Prior*, *Next* o *Last*. Cuando se trata de *Prior* o *Next* hay que indicarle al servidor en qué registro se encontraba el usuario, para lo cual utilizaremos el parámetro *Codigo*. Por lo tanto, pulse sobre la propiedad *Actions* del módulo Web, y añada una nueva acción, de nombre *acFirst*, marcada como *Default*, y deje vacía su propiedad *PathInfo*:



Sobre el módulo Web generado añadiremos los siguientes componentes:



Más sencillo imposible: tenemos una conexión DB Express, que hemos bautizado *SQLConn*, a la conocida base de datos de InterBase:

*C:\Archivos de programa\Archivos comunes\Borland Shared\Data\mastsql.gdb*

Hay también un *TADOQuery* conectado al componente de conexión. La he llamado *qrEmpleados*, y he puesto la siguiente consulta en su interior:

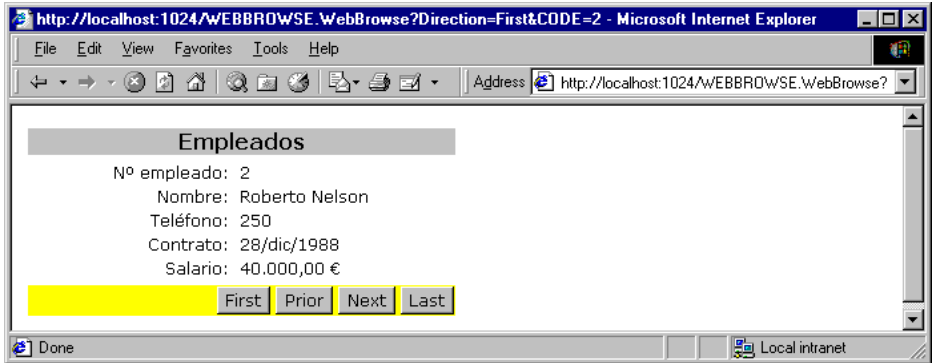
```
select *
from   EMPLOYEE
where  EmpNo = -1
```

La consulta no devuelve fila alguna, pero nos permite crear los componentes de acceso a campos y configurarlos a nuestro antojo. La instrucción SQL de la consulta se cambiará en tiempo de ejecución, cada vez que se realice una petición de página al servidor.

Añada luego dos productores HTML, *DataSetPageProducer1* y *PageProducer1*. Debemos hacer que la propiedad *DataSet* del primero de ellos apunte a *qrEmpleados*. Y hay que teclear la siguiente plantilla dentro de su propiedad *HTMLDoc*.

```
<HTML>
<BODY>
  <H1>Employees</H1><HR>
  <FORM METHOD=GET>
    <P>
      Code: <#EmpNo><BR>
      Name: <#FirstName> <#LastName><BR>
      Extension: <#PhoneExt><BR>
      Salary: <#Salary><BR>
      Hire date: <#HireDate>
    </P><HR>
    <#CODE>
    <INPUT TYPE=SUBMIT NAME=Direction VALUE=First>
    <INPUT TYPE=SUBMIT NAME=Direction VALUE=Prior>
    <INPUT TYPE=SUBMIT NAME=Direction VALUE=Next>
    <INPUT TYPE=SUBMIT NAME=Direction VALUE=Last>
  </FORM>
</BODY>
</HTML>
```

Observe la presencia de etiquetas con los nombres de campos de la tabla de empleados. Cuando le pidamos a este generador HTML su contenido, dichas etiquetas se sustituirán por el valor de los campos del conjunto de datos asociado. Además de las etiquetas que corresponden directamente a campos, he incluido una etiqueta especial, `<#CODE>`, para que la página “recuerde” cuál es el código del registro visualizado. Este `<#CODE>` será expandido como un campo de formulario de tipo `HIDDEN`. El aspecto de la página generada será el siguiente:



Por su parte, el contenido de *PageProducer1* se mostrará únicamente en una situación muy especial, como veremos más adelante:

```
<HTML>
<BODY>
  <H1>Employees</H1><HR>
  <FORM METHOD=GET>
    <P>Record/key deleted.</P><HR>
    <INPUT TYPE=SUBMIT NAME=Direction VALUE=First>
    <INPUT TYPE=SUBMIT NAME=Direction VALUE=Last>
  </FORM>
</BODY>
</HTML>
```

Después hay que interceptar el evento *OnAction* de la única acción del módulo, para que se ejecute durante el mismo el algoritmo de generación y apertura de la consulta:

```
procedure TWebModule1.WebModule1acFirstAction(
  Sender: TObject; Request: TWebRequest; Response: TWebResponse;
  var Handled: Boolean);
var
  Direction: string;
  CurrRecord: Integer;
begin
  CurrRecord := 0;
  // Extraer parámetros
  if Request.QueryFields.IndexOfName('CODE') <> -1 then
  begin
    CurrRecord := StrToInt(Request.QueryFields.Values['CODE']);
    Direction := AnsiUpperCase(
      Request.QueryFields.Values['Direction']);
  end;
end;
```

```

// Generar contenido
if (Direction = 'FIRST') or (Direction = '') then
    GenerateQuery('order by EmpNo asc', CurrRecord)
else if Direction = 'PRIOR' then
    GenerateQuery('where EmpNo < %d order by EmpNo desc',
        CurrRecord)
else if Direction = 'NEXT' then
    GenerateQuery('where EmpNo > %d order by EmpNo asc',
        CurrRecord)
else
    GenerateQuery('order by EmpNo desc', CurrRecord);
if qrEmpleados.Eof then
    GenerateQuery('where EmpNo = %d', CurrRecord);
if qrEmpleados.Eof then
    Response.Content := PageProducer1.Content
else
    Response.Content := DataSetPageProducer1.Content;
qrEmpleados.Close;
end;

```

Quiero que preste atención a esta parte del método:

```

if qrEmpleados.Eof then
    GenerateQuery('where EmpNo = %d', CurrRecord);
if qrEmpleados.Eof then
    Response.Content := PageProducer1.Content
else
    Response.Content := DataSetPageProducer1.Content;

```

Supongamos que estamos ya en el último registro de la tabla, y que el usuario pide el siguiente. Es evidente que *qrEmpleados* estará vacía, porque estaremos solicitando el registro cuyo código es mayor que el mayor de los códigos. En tal caso, generamos una nueva consulta que pida el mismo registro que tenía el usuario antes. Normalmente, esta petición no falla, pero puede darse el caso en que algún proceso concurrente elimine al empleado deseado. Para cubrirnos las espaldas, mostramos entonces la página contenida en *PageProducer1*, que contiene el mensaje de error *Record or key deleted*, y dejamos al usuario la libertad de dirigirse al primer o al último registro de la tabla.

Todo el algoritmo anterior está basado en el uso de un método auxiliar, *GenerateQuery*, que se implementa del siguiente modo:

```

procedure TWebModule1.GenerateQuery(const Tail: string;
    CurrRecord: Integer);
begin
    qrEmpleados.Close;
    // Generar contenido
    qrEmpleados.SQL.Clear;
    qrEmpleados.SQL.Add('select * from Employee');
    qrEmpleados.SQL.Add(Format(Tail, [CurrRecord]));
    qrEmpleados.Open;
end;

```

Lo único que nos queda es la sustitución de etiquetas durante la generación del contenido de *DataSetPageProducer1*:

```

procedure TWebModule1.DataSetPageProducer1HTMLTag(
    Sender: TObject; Tag: TTag; const TagString: string;
    TagParams: TStrings; var ReplaceText: string);
begin
    if SameText(TagString, 'CODE') then
        ReplaceText := Format(
            '<input type=hidden name=CODE value=%d>',
            [qrEmpleadosEmpNo.Value]);
end;
    
```

## ¿Le apetece una galleta?

Existen muchas otras formas de mantener la información de estado relacionada con un cliente. Una de ellas son las *cookies*, que Delphi soporta mediante propiedades y métodos de las clases *TWebRequest* y *TWebResponse*. Todo comienza cuando una extensión HTTP envía junto a una página HTML una de estas *cookies*. El método de envío pertenece a la clase *TWebResponse*, y tiene el siguiente aspecto:

```

procedure TWebResponse.SetCookieField(Values: TStrings;
    const ADomain, APath: string;
    AExpires: TDateTime; ASecure: Boolean);
    
```

La información básica de este método se pasa en el parámetro *Values*, como una lista de cadenas con el formato *Parámetro=Valor*. Por ejemplo:

```

procedure EnviarIdentificadorUsuario(
    Response: TWebResponse; UserID: Integer);
var
    Valores: TStrings;
begin
    Valores := TStringList.Create;
    try
        Valores.Add('UserID=' + IntToStr(UserID));
        Response.SetCookieField(valores, 'marteens.com', '',
            UTCNow + 1, False);
    finally
        Valores.Free;
    end;
end;
    
```

El objetivo de una *cookie* es que sea devuelta en algún momento al servidor. Los parámetros *ADomain* y *APath* indican a qué dominio y ruta se debe enviar de vuelta esta información. *AExpires* establece una fecha de caducidad para la *cookie*; en el ejemplo anterior, sólo dura un día. Observe que he llamado a esta función:

```

function UTCNow: TDateTime;
var
    S: TSystemTime;
begin
    GetSystemTime(S);
    Result := EncodeDate(S.wYear, S.wMonth, S.wDay) +
        EncodeTime(S.wHour, S.wMinute, S.wSecond, S.wMilliseconds);
end;
    
```

El problema es que hay que indicar las fechas en el sistema de referencia UTC, o *Universal Time Coordinates*, para evitar malentendidos si el usuario y el servidor se encuentran en distintos husos horarios. Y es que *Now* habría devuelto la hora local.

Por último, el parámetro *ASecure* de *SetCookieField* especifica, cuando vale *True*, que sólo debe devolverse la información de la *cookie* a través de una conexión segura.

La otra cara de la moneda es cómo recibir esta información de vuelta. Para ello, la clase *TWebRequest* tiene las propiedades *Cookie*, de tipo **string**, y *CookieFields*, que apunta a una lista de cadenas. El trabajo con estas propiedades es similar al de las propiedades *Query/QueryFields* y *Content/ContentFields*.

La diferencia principal entre las *cookies* y el uso de parámetros en la consulta o en el contenido consiste en que la información almacenada por el primer mecanismo puede recuperarse de una sesión a otra, mientras que los restantes sistemas de mantenimiento de información solamente sirven para encadenar secuencias de página durante una misma sesión.

Sin embargo, a pesar de que esta técnica es inofensiva y relativamente poco intrusiva, goza de muy poca popularidad entre los usuarios de la Internet. Cuando una *cookie* es recibida por un navegador Web, se le pregunta al usuario si desea o no aceptarla, y es muy probable que la rechace. También hay que contar con la fecha de caducidad, y con la posibilidad de que el propietario del ordenador se harte de tener la mitad de su disco duro ocupada por porquerías bajadas desde la Internet, y borre todo el directorio de archivos temporales, *cookies* incluidas. Por lo tanto, utilice este recurso para guardar información opcional, de la que pueda prescindir sin mayor problema.

#### NOTA

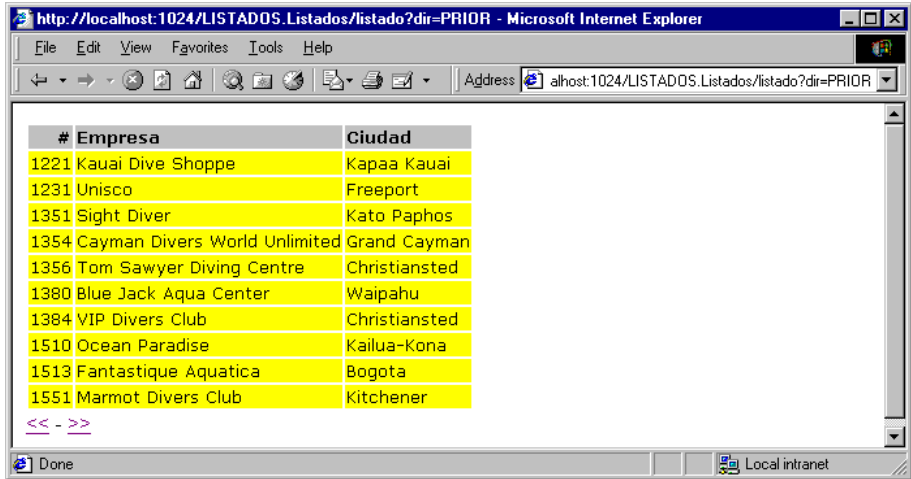
Más adelante veremos que el mantenimiento del estado en las aplicaciones ASP está basado principalmente en el uso de *cookies*. Es cierto que ASP nos permite también prescindir de las *cookies*, pero entonces el mantenimiento del estado debe lograrse con técnicas similares a las que acabamos de estudiar.

## Navegación por grupos de registros

Me veo obligado a mostrar un ejemplo del uso de *cookies*, aunque me gustaría escurrir el bulto. Programaremos una página que presente los registros de determinada tabla, en grupos de 10 registros, digamos. Y la dificultad consistirá en que añadiremos enlaces, o botones, o el elemento gráfico que más le guste, para recuperar en cualquier momento el grupo de registros siguiente o el anterior.

¿Parece fácil? No lo es, sin embargo, al menos para quien resuelve el ejercicio por primera vez. Una de las dificultades es que no podremos utilizar, al menos de forma directa, los productores *TDataSetTableProducer* y *TQueryTableProducer*, por motivos que comprenderemos enseguida. Además, en dependencia del criterio de ordenación de

los registros que elijamos, la generación de consultas puede fastidiarnos una mañana de sol. La parte de “mantenimiento del estado” es quizás la menos complicada.



#	Empresa	Ciudad
1221	Kauai Dive Shoppe	Kapaa Kauai
1231	Unisco	Freeport
1351	Sight Diver	Kato Paphos
1354	Cayman Divers World Unlimited	Grand Cayman
1356	Tom Sawyer Diving Centre	Christiansted
1380	Blue Jack Aqua Center	Waipahu
1384	VIP Divers Club	Christiansted
1510	Ocean Paradise	Kailua-Kona
1513	Fantastique Aquatica	Bogota
1551	Marmot Divers Club	Kitchener

<< - >>

Note la presencia de dos enlaces, uno para el grupo anterior y otro para el siguiente. El tamaño de cada grupo, 10 registros, ha sido elegido arbitrariamente. La tabla base pertenece también a la base de datos de InterBase *mastsql.gdb*. Se trata esta vez de *CUSTOMER*, que contiene datos de empresas. Para simplificar asumiremos que se desea ordenar los registros por el número del cliente: la columna *CustNo*.

Comenzamos, como ya es habitual, creando una aplicación vacía, con su módulo de datos Web, y bautizamos el fichero principal del proyecto con el nombre *listados.dpr*. Inmediatamente después creamos una acción dentro del módulo, que llamaremos *acListado*. Debemos asociar a su propiedad *PathInfo* el valor */listado*, y es conveniente que activemos su propiedad *Default*.

Traiga entonces un componente de conexión, *TSQLConnection*, y un *TADOQuery*. Les he dado los nombres, respectivamente, de *Database* y *qrCust*. Haga que la propiedad *SQLConnection* de *qrCust* apunte a *Database*, y teclee la siguiente instrucción en su propiedad *SQL*, que aunque no devuelve filas, nos permitirá configurar los campos:

```
select *
from CUSTOMER
where CustNo = -1
```

¿Listos para añadir el código? Primero declaramos una constante de cadenas dentro de la unidad, que contenga nuestra definición de estilos en cascada:

```
const
    MyStyles = '<style> ' +
        'td { font: 10pt verdana; background-color: #F0F0F0 } ' +
        'th { font: bold 10pt verdana; background-color: silver } ' +
        '</style>';
```

A continuación crearemos varios métodos auxiliares en la sección **protected** de la clase del módulo Web:

```
// Métodos auxiliares de generación HTML
function TmodData.GenerarCabecera: string;
begin
    Result := '<tr><th align=right>#</th>' +
              '<th align=left>Empresa</th><th align=left>Ciudad</th></tr>';
end;

function TmodData.GenerarFila: string;
begin
    Result := Format(
        '<tr><td align=right>%s</td><td>%s</td><td>%s</td></tr>',
        [qrCustCustNo.DisplayText, qrCustCompany.DisplayText,
        qrCustCity.DisplayText]);
end;

function TmodData.GenerarNavegador: string;
begin
    Result :=
        '<a href="listado?dir=PRIOR">&lt;&lt;</a> - ' +
        '<a href="listado?dir=NEXT">&gt;&gt;</a>';
end;
```

Como puede ver, se trata de manipulaciones muy sencillas de cadenas de caracteres. La primera función genera una cabecera constante para una tabla de tres columnas: número de cliente, nombre y ciudad. La segunda se encarga de generar una fila de datos. La última es la que devuelve el texto correspondiente a los dos enlaces. Preste atención a este punto: cada enlace vuelve a llamar a la propia acción *listado*, pero con un valor diferente en el parámetro *dir*. Eso quiere decir que *listado* puede ejecutarse con uno de tres valores en *dir*: *next* para el siguiente grupo, *prior* para el anterior ... y la cadena vacía para volver a mostrar el grupo actual. O para recuperar el primer grupo de la tabla.

Quizás usted se pregunte: ¿por qué complicarse tanto para generar una simple tabla HTML? La mejor forma de explicarlo es mostrar primero el código de un método auxiliar que utilizaremos para crear una instrucción SQL para la consulta:

```
function TmodData.GenerarInstruccion(Direccion: SmallInt;
    Amin, AMax: Integer): string;
begin
    Result := 'select * from CUSTOMER ';
    if Direccion > 0 then
        Result := Result + 'where CustNo > ' + IntToStr(AMax) +
                    ' order by CustNo asc'
    else if Direccion = 0 then
        Result := Result + 'where CustNo >= ' + IntToStr(Amin) +
                    ' order by CustNo asc'
    else
        Result := Result + 'where CustNo < ' + IntToStr(Amin) +
                    ' order by CustNo desc';
end;
```



Habíamos quedado en que el parámetro *dir* podría tener uno de tres valores. Para simplificar las comparaciones, vamos a utilizar entonces un convenio muy frecuente, representando la dirección de avance mediante un entero corto: si es negativo nos movemos hacia atrás, si es cero nos quedamos en nuestro sitio, y si es positivo adelantamos.

### PROPUESTA

Esta es una oportunidad estupenda para probar la cláusula **rows**, recién introducida por InterBase 6.5.

A *GenerarInstruccion* le pasamos, además de la dirección de avance, un valor mínimo y otro máximo, que corresponden a los códigos de clientes del grupo anterior. Esos dos valores representarán el *estado* de nuestra sencilla aplicación en todo momento. Por el momento, no se cuestione cómo vamos a lograr “mantener” esos valores de petición a petición. Lo que quiero que note ahora es que, cuando nos movemos al grupo anterior, utilizamos una consulta ordenada en forma descendente. Claro, los resultados hay que mostrarlos al revés, desde los códigos menores hasta los mayores. Es por eso que no nos valen directamente los productores de tablas de Delphi. Y es el motivo que tenemos para definir el siguiente método auxiliar:

```
function TmodData.GenerarTabla(Direccion: SmallInt;
    Filas: Integer; var AMin, AMax: Integer): string;
begin
    qrCust.Open;
    try
        Result := '';
        AMin := 999999999;
        AMax := 0;
        while not qrCust.Eof and (Filas > 0) do
            begin
                if Direccion >= 0 then
                    Result := Result + GenerarFila
                else
                    Result := GenerarFila + Result;
                    AMin := Min(qrCustCustNo.AsInteger, AMin);
                    AMax := Max(qrCustCustNo.AsInteger, AMax);
                    qrCust.Next;
                    Dec(Filas);
                end;
                Result := '<table>' + GenerarCabecera + Result + '</table>';
            finally
                qrCust.Close;
            end;
        end;
    end;
```

¿Se da cuenta de que es un truco tonto? En vez de liarnos con movimientos estrambóticos del cursor de la consulta, siempre recorremos la tabla desde el primer registro devuelto. Pero cuando la dirección de avance es negativa añadimos las nuevas filas *antes*, y no *después*, de las filas acumuladas. Una advertencia: los parámetros de referencia *AMin* y *AMax* sirven para recuperar los valores mínimo y máximo del nuevo grupo. En cambio, en *GenerarInstruccion* almacenaban los mismos valores, pero del grupo anterior.

Ahora ya estamos listos para integrar todos los métodos en la respuesta de la acción *listado*:

```
procedure TmodData.acListadoAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse;
  var Handled: Boolean);
var
  AMin, AMax: Integer;
  Direccion: SmallInt;
begin
  // Estado actual de la aplicación
  AMin := StrToIntDef(Request.CookieFields.Values['MIN'], 0);
  AMax := StrToIntDef(Request.CookieFields.Values['MAX'], 0);
  // Parámetros: el estado "transitorio"
  with Request.QueryFields do
    if CompareText(Values['DIR'], 'NEXT') = 0 then
      Direccion := 1
    else if CompareText(Values['DIR'], 'PRIOR') = 0 then
      Direccion := -1
    else
      Direccion := 0;
  // Generación del texto HTML
  qrCust.SQL.Text := GenerarInstruccion(Direccion, AMin, AMax);
  Response.Content := MyStyles +
    GenerarTabla(Direccion, 10, AMin, AMax) + GenerarNavegador;
  // Modificar el estado actual de la aplicación
  with Response.Cookies.Add do begin
    Name := 'MIN';
    Value := IntToStr(AMin);
  end;
  with Response.Cookies.Add do begin
    Name := 'MAX';
    Value := IntToStr(AMax);
  end;
end;
```

Concentraremos la atención en la primera y última secciones del método. Primero, se recupera el estado actual de la aplicación dentro de las variables locales correspondiente. Observe que hay que asignar valores por omisión sensatos para la primera vez que se llama a la página. Después de generar el texto HTML correspondiente al estado de la aplicación (el mínimo y el máximo que traen las *cookies*) y al comando de movimiento pasado en la URL, se añaden *cookies* a la respuesta, pero conteniendo los valores que representan el estado final.

### EJERCICIO PROPUESTO

Cuando eche a andar este ejemplo comprobará, entre otras cosas, que la navegación es circular ... pasando por una página vacía, que bien podríamos sustituir por un mensaje del tipo *"No hay más registros"*. Le propongo que intente las modificaciones pertinentes. Investigue también cómo "activar" y "desactivar" los enlaces de navegación cuando no hay más registros en una u otra dirección.

## Aplicaciones basadas en conexiones

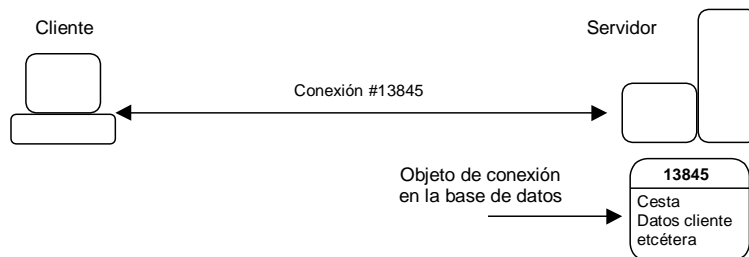
En el ejemplo anterior, el estado de la aplicación podía representarse mediante dos parámetros: el valor mínimo y el máximo de los códigos de clientes mostrados en un momento dado. Dos valores pueden transmitirse fácilmente de página en página, ya sea mediante *cookies*, campos ocultos o la técnica que decidamos. Pero una aplicación real tendrá un estado interno mucho más complejo, y necesitará más parámetros para transmitirlo.

Tomemos la aplicación típica: una tienda en Internet que haga uso de un “carro de la compra” (*shopping cart*) y que implemente la búsqueda de productos por palabras claves y la navegación por secciones. Sin pensarlo demasiado identificamos los siguientes datos como parte del estado de la aplicación:

- La última condición de búsqueda utilizada. Esta puede incluir la sección de la tienda en que se encuentra el cliente, más las palabras claves que haya utilizado.
- El estado de la navegación sobre el conjunto resultado de la búsqueda anterior. Como ya hemos comprobado, al menos necesitamos un valor máximo y un valor mínimo para el conjunto de columnas por el que aparece ordenado el resultado.
- La propia cesta de la compra, que a su vez constituye un valor “complejo”. La cesta consiste principalmente en una lista de registros; cada registro contiene una referencia de producto y una cantidad solicitada, por lo menos.
- Si el cliente se ha identificado ante la aplicación, los datos del cliente también forman parte del estado a transmitir.

¿Se imagina la pesadilla que sería tener que propagar todos esos datos manualmente? Estoy pensando sobre todo en “objetos” como la cesta de la compra. Si aceptásemos el uso de *cookies* sería una tarea complicada aunque posible, pero si tenemos que utilizar campos ocultos, el desarrollo de la aplicación se nos escaparía de las manos.

Por suerte, existe una solución muy elemental: encapsular parte del estado en un objeto, almacenar dicho objeto en el lado del servidor, y transmitir solamente la identidad del objeto:



Las reglas del juego son las siguientes:

- Cada petición que se haga al servidor desde nuestra aplicación debe ir acompañada por un identificador numérico. El identificador corresponderá a la clave primaria de un registro de cierta tabla especial de la base de datos mantenida por el servidor.
- La primera vez que se conecta un usuario al servidor no dispone de ese identificador. El servidor debe detectar ese caso especial y crear un nuevo registro. Siempre que el servidor responda al cliente, la respuesta debe incluir el mismo identificador correspondiente a la petición, o uno nuevo si se trata de la primera.
- El estado de la aplicación lo cambia el cliente en el navegador, pulsando sobre enlaces o modificando información en formularios.
- Es responsabilidad del servidor “duplicar” esos cambios realizados en la parte cliente sobre el registro correspondiente.

Como los registros de que hablamos se crean cuando un usuario inicia una *conexión*, la tabla que los contiene suele denominarse *CONEXIONES* y el identificador que se pasa de una página a otra, el *identificador de conexión* (*connection ID*). Analice el siguiente *script* de InterBase:

```
create table CLIENTES (
    IDCliente    integer    not null,
    EMail        varchar(127) not null,
    Nombre       varchar(50)  not null,
    Direccion    varchar(50)  not null,

    primary key (IDCliente),
    unique      (EMail)
);

create table CONEXIONES (
    IDConexion   integer    not null,
    UserAgent    varchar(127) not null,
    Referer      varchar(127) not null,
    Hora         datetime    default 'Now' not null,
    IDCliente    integer    null,

    primary key (IDConexion),
    foreign key (IDCliente) references CLIENTES(IDCliente)
                        on update cascade on delete set null
);
```

Me he visto obligado a incluir la tabla de clientes porque lo típico es que cada conexión se identifique, más tarde o temprano, con uno de los clientes que ya conocemos de visitas anteriores. O que creamos un nuevo registro de cliente y lo asociemos a la conexión. Observe que la referencia al código de cliente en la tabla de conexiones admite valores nulos.

Podemos entonces crear un procedimiento almacenado para crear nuevas conexiones cuando estemos que sea necesario. He aquí una posibilidad:

```
set term ^;
```

```

create generator GenConn^

create procedure NuevaConexion (
    agente varchar(127),
    referencia varchar(127))
returns (idcon integer) as
begin
    idcon = gen_id(GenConn, 1);
    /* La hora se asume por omisión como la actual */
    insert into CONEXIONES(IDConexion, UserAgent, Referer)
    values (:idcon, :agente, :referencia);
end ^
    
```

El mejor momento para controlar que una petición HTTP vaya acompañada de su número de conexión es durante la respuesta al evento *BeforeDispatch* del módulo de datos Web. Supongamos que hemos declarado una variable de atributo *FConexion*, de tipo entero, dentro de la clase del módulo de datos, y que hemos conectado el componente *spNuevaConexion* al correspondiente procedimiento almacenado:

```

type
    TmodData = class(TWebModule)
        // ...
        spNuevaConexion: TStoredProc;
        // ...
    private
        FConexion: Integer;
        function Parametro(const AName: string): string;
        function ParamEntero(const AName: string): Integer;
    end;
    
```

El siguiente manejador de eventos se encarga del asunto, con la ayuda de dos métodos auxiliares:

```

function TmodData.Parametro(const AName: string): string;
begin
    Result := Request.ContentFields.Values[AName];
    if Result = '' then
        Result := Request.QueryFields.Values[AName];
    if Result = '' then
        Result := Request.CookieFields.Values[AName];
end;

function TmodData.ParamEntero(const AName: string): Integer;
begin
    Result := StrToIntDef(Parametro(AName), -1);
end;

procedure TmodData.WebModuleBeforeDispatch(Sender: TObject;
    Request: TWebRequest; Response: TWebResponse;
    var Handled: Boolean);
begin
    FConexion := ParamEntero('IDCON');
    if FConexion < 0 then
        begin
            spNuevaConexion.ParamByName('agente').AsString :=
                Request.UserAgent;
            spNuevaConexion.ParamByName('referencia').AsString :=
                Request.Referer;
        
```

```

spNuevaConexion.ExecProc;
FConexion := spNuevaConexion.ParamByName('idcon').AsInteger;
with Response.Cookies.Add do
begin
    Name := 'IDCON';
    Value := IntToStr(FConexion);
end;
end;
end;

```

Cada vez que se recibe una petición, se examinan los parámetros pasados con los método GET y POST, y las *cookies* de la petición. Aquí he decidido que el parámetro a buscar se llame *IDCON*, pero está claro que puede utilizar el nombre que menos escozor le cause. Si no existe tal parámetro, pedimos un nuevo registro a la base de datos, y adicionalmente enviamos el valor como una *cookie* dentro de la respuesta. Esto último no es obligatorio, sin embargo. Su objetivo es permitir el mantenimiento del número de conexión en ciertas circunstancias fuera de nuestro control.

¿Cuáles son entonces las circunstancias “dentro de nuestro control”? Supongamos que una de sus páginas HTML deba contener un formulario. Añada la siguiente línea a la plantilla:

```

<form name=unNombre action="loquequiera">
    <!-- ... aquí va el contenido del formulario ... -->
    <#IDCON>
</form>

```

Es decir, añada una etiqueta especial *#idcon* dentro de cada uno de sus formularios. Luego haga la siguiente sustitución en el productor de contenido adecuado:

```

// ...
if CompareText(TagString, 'IDCON') = 0 then
    ReplaceText := Format(
        '<input type=hidden name="IDCON" value="%d">',
        [FConexion])
// ...

```

Con esta sencilla técnica, todos los saltos de página que realice mediante formularios mantendrán la continuidad del número de conexión asignado la primera vez. ¿Qué pasa, sin embargo, con los enlaces “normales”? Existen varias soluciones, desde sustituir las referencias directas a recursos HTML por código en JavaScript, como vimos en el capítulo 37, hasta analizar sintácticamente el código de la plantilla y hacer sustituciones automáticas. Puede utilizar la solución más laboriosa, que consiste en añadir manualmente el identificador de conexión a cada enlace explícito. Por ejemplo, si en la plantilla original existe un enlace como el siguiente:

```

<a href="www.marteens.com/tienda.exe/pagar">Pagar</a>

```

podríamos “retocarlo” del siguiente modo:

```

<a href="www.marteens.com/tienda.exe/pagar?idcon=<#idcon>">Pagar</a>

```

Recuerde que no hay problema alguno en incrustar una etiqueta especial reconocida solamente por Delphi dentro de una etiqueta de HTML e incluso dentro de una cadena de caracteres.





## Páginas Activas en el Servidor

**Y** AHORA, UN ACTO DE MAGIA. VAMOS A MEZCLAR dos tecnologías que aparentemente no tienen nada en común: Internet y la programación COM. Es interesante saber que hay más de una forma de realizar este enlace. En el lado servidor podemos crear objetos de automatización que pueden invocarse desde páginas ASP. En el lado cliente veremos cómo incluir controles ActiveX en una página HTML; en particular, Delphi hace posible darle a uno de estos controles la apariencia y el comportamiento de un formulario. Para terminar el capítulo, explicaremos brevemente qué es Internet Express... y por qué no le he dedicado un capítulo en esta edición.

### ASP: páginas activas en el servidor

Usted comienza a programar aplicaciones para Internet utilizando las técnicas aprendidas en las páginas anteriores. Cada vez que define una plantilla, diseña etiquetas que deben expandirse una y otra vez en código HTML. Encuentra entonces parecidos, semejanzas y puntos en común, y añade parámetros a algunas etiquetas. Finalmente, descubre que siempre utiliza un núcleo común bien delimitado de etiquetas de una aplicación en otra. Le añade instrucciones de control y acaba de crear su propio lenguaje de *script*: un conjunto de pseudo etiquetas HTML que pueden incluirse en cualquier plantilla y que son interpretadas en el lado del servidor.

*Active Server Pages*, o ASP, es una opción de Internet Information Server y de las versiones más recientes de Personal Web Server que permite utilizar lenguajes de *script* en el lado del servidor. Un error habitual es pensar que ASP es un lenguaje, cuando en realidad se trata de un motor extensible. Actualmente soporta VBScript y JScript (la versión microsfónica de JavaScript), pero como casi todos los libros de programación ASP se basan en el primer lenguaje, existe la falsa impresión generalizada de que ASP y Visual Basic Script son sinónimos.

Las sentencias que interpreta ASP se encierran entre etiquetas especiales. En vez de utilizar etiquetas transparentes que comiencen con la almohadilla, ASP aprovecha el carácter de porcentaje:

```
<%@ LANGUAGE = "JScript" %>  
<html>
```

```
<body>
<p>Hoy es <% Response.Write(Date()); %>.</p>
</body>
</html>
```

La primera línea del ejemplo muestra cómo indicarle al servidor el lenguaje en el que están programadas las sentencias intercaladas en el texto HTML. Tenga bien presente que si omite esta indicación, el servidor asumirá que el lenguaje es VBScript.

Sigamos. ¿Recuerda los ejemplos de generación dinámica del documento mediante JavaScript? El código antes mostrado es muy similar ... excepto que la generación se produce en el servidor en vez de en el cliente. Aquí está la clave de la potencia de ASP. ¿Dónde debe situarse una base de datos para que dé servicio a una aplicación, en el servidor o en el cliente?<sup>38</sup>

En particular, la página ASP que hemos visto devuelve al navegador un texto HTML similar al siguiente:

```
<html>
<body>
<p>Hoy es Vie 4 Jul 10:34:06 2000.</p>
</body>
</html>
```

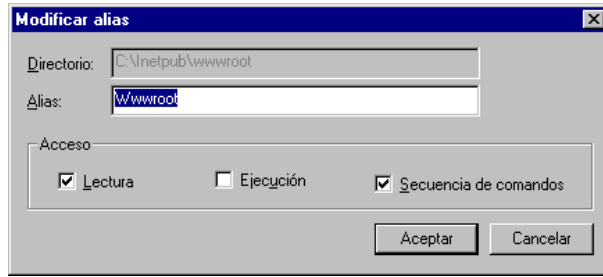
Observe que han desaparecido los marcadores ASP originales. El marcador que especificaba el lenguaje del *script* no ha dejado huella, mientras que el segundo marcador ha sido sustituido con la fecha y hora actual (la hora del servidor).

Active Server Pages funciona gracias a una aplicación ISAPI que acompaña a IIS, y a PWS para Windows 98. A estos servidores podemos pedirle un documento con extensión *asp* como si se tratase de un documento HTML vulgar y corriente. Sin embargo, la extensión mencionada se encuentra registrada por omisión por IIS, y antes de devolver el documento al cliente, éste se pasa por un *filtro*, que se encarga de interpretar las sentencias ASP y de eliminarlas de la respuesta. Esto es muy importante: el cliente jamás llega a ver las sentencias ASP que incrustamos en la página original.

Para que un cliente pueda leer documentos ASP de un determinado directorio virtual del servidor, hay que conceder antes permisos de ejecución de “secuencias de comandos” sobre el directorio. La imagen de la siguiente página muestra el diálogo de permisos Web de un ordenador con Windows 98 y PWS.

---

<sup>38</sup> Si responde que en el cliente, puede vender el ordenador y dedicarse a alguna otra actividad más lucrativa, como la cría de avestruces.



## Los objetos globales de ASP

No voy a insistir demasiado en detalles triviales como la sintaxis de las instrucciones básicas de ASP, del mismo modo que tampoco lo hice en su momento con JavaScript. Por el contrario, nos concentraremos en las posibilidades que nos ofrece el entorno de ejecución de estos guiones.

No es sorprendente encontrar un gran parecido con las aplicaciones CGI; en definitiva, ASP se ejecuta en el servidor, al igual que estas aplicaciones. El entorno de ejecución nos da acceso a cinco objetos globales básicos. No tenemos que preocuparnos por su creación, porque de ello se encarga Internet Information Server, y son:

Objeto	Qué representa
<i>Request</i>	La petición HTTP
<i>Response</i>	La respuesta que enviaremos
<i>Application</i>	Datos globales de nuestra “aplicación”
<i>Session</i>	Datos globales de la conexión actual
<i>Server</i>	Contiene utilidades, y permite crear otros objetos ActiveX

Ya hemos visto un ejemplo sencillo en el que ejecutábamos el método *Write* del objeto *Response*. Al igual que sucede con una aplicación CGI, la respuesta a una petición HTTP es un documento HTML, en la mayoría de las ocasiones. Es tan frecuente la escritura de texto HTML en la respuesta, que ASP proporciona un método alternativo abreviado. Cuando hay un signo igual inmediatamente al principio de una etiqueta ASP, se evalúa la expresión que debe venir a continuación, se traduce a HTML y se incrusta dentro de la respuesta:

```
<html>
<body>
<p>Hoy es <%= Date() %>.</p>
</body>
</html>
```

Naturalmente, también podemos manipular otros métodos y propiedades del objeto *Response*, como *ContentType* y *Redirect* para enviar otros contenidos al navegador, o para desviar la petición a otra página o dominio.

El objeto *Request* es importante principalmente porque contiene los parámetros de entrada de la página. Los parámetros enviados mediante `POST` se reciben en la propiedad *Form*, y los de tipo `GET` en la propiedad vectorial *QueryString*. Las *cookies* se reciben, por supuesto, en una propiedad llamada *Cookies*. El siguiente ejemplo muestra la forma común en que se trabaja con las tres propiedades anteriores:

```
<script runat="server" language="javascript">
    if (Request.QueryString("Nombre").Count > 1) {
        Response.Write("Nombres:<br><ol>");
        for (var i = 1; i <= Request.QueryString("Nombre").Count; i++)
            Response.Write("<li>" + Request.QueryString("Nombre")(i));
        Response.Write("</ol>");
    }
    else {
        Response.Write("Nombre: ");
        Response.Write(Request.QueryString("Nombre"));
    }
}</script>
```

En primer lugar, observe que he agrupado las instrucciones dentro de etiquetas `<SCRIPT>`. Lo único diferente a ejemplos anteriores es el parámetro *runat*, para indicar que las instrucciones deben ejecutarse en el servidor.

```
if (Request.QueryString("Nombre").Count > 1) {
```

La instrucción anterior comprueba el número de ocurrencias del parámetro llamado *Nombre* en la petición. Por costumbre, cada nombre parámetro se utiliza una sola vez en cada petición, pero es posible repetir un parámetro, como en la siguiente URL:

```
www.marteens.com/comprar.asp/cesta?code=101&qty=2&code=102&qty=1
```

Si el parámetro aparece una vez nada más, su valor se obtiene mediante la expresión:

```
Request.QueryString("Nombre")
```

Pero si hay más de una ocurrencia deberemos añadir un nivel adicional de índices. Las ocurrencias comienzan a contarse a partir de uno:

```
Request.QueryString("Nombre")(1)
Request.QueryString("Nombre")(2)
```

## ASP y objetos ActiveX

Por muy potente que sea un lenguaje de guiones, es imposible para su diseñador incluir todos los recursos que algún programador necesite en algún remoto instante del futuro. Ya pasaron a la historia aquellos horribles lenguajes “de cuarta generación” que presumían de ser lenguajes de propósito específico. El ejemplo más sencillo: necesitamos tener acceso a bases de datos. ¿Qué hacemos, dotamos al lenguaje de instrucciones especiales que controlen el motor de ODBC o ADO? Esto es pan para hoy, ¿pero qué sucederá mañana cuando el motor de moda sea otro?

La solución es permitir que el lenguaje se extienda con bibliotecas arbitrarias que no tengan necesariamente que haber sido creadas con el propio lenguaje. En ASP este concepto se traduce en la posibilidad de crear objetos ActiveX y de poder ejecutar cualquiera de sus métodos. El encargado de la creación de objetos es el objeto global *Server*. En el siguiente ejemplo, el guión crea un par de objetos de ADO para obtener un simple listado HTML:

```
<script runat=server language=JScript>
    var conexion = Server.CreateObject("ADODB.Connection");
    var consulta = Server.CreateObject("ADODB.Recordset");

    conexion.Open(
        "Provider=SQLOLEDB.1;Data Source=narao;Initial Catalog=pubs",
        "sa", "");
    consulta.Open("select * from authors", conexion);

    while (! consulta.EOF)
    {
        Response.Write(consulta("au_lname"));
        Response.Write("<br>");
        consulta.MoveNext();
    }
</script>
```

Se supone que *pubs* es el nombre de una fuente de datos de ODBC, y que ADO utilizará el puente ODBC para abrir dicha conexión. En el ejemplo he dado por sentado que *pubs*, además, se refiere a una de las bases de datos que se instalan con SQL Server 7. Usted puede cambiar la conexión y la consulta de acuerdo a cómo tenga configurado su ordenador.

#### NOTA

Si esta página reside en un servidor NT, la fuente de datos *pubs* debe ser una fuente de sistema, para que pueda ser utilizada desde un servicio. Además, he identificado al usuario de la conexión mediante su *login* de SQL Server, sin utilizar seguridad integrada. Esta otra restricción es también consecuencia de que la página ASP se expanda dentro del espacio de un proceso de servicio.

Primero veamos cómo se crea un objeto desde ASP. El ejemplo, en realidad, crea dos objetos diferentes: una conexión y una consulta, pero la forma de hacerlo es similar:

```
var conexion = Server.CreateObject("ADODB.Connection");
```

La palabra reservada **var** declara una variable para que apunte al objeto ActiveX. En la misma línea se llama al método *CreateObject* del objeto global *Server* para crear un objeto ActiveX dado el nombre de su clase. Por supuesto, la clase en cuestión debe estar registrada en el ordenador que está ejecutando el servidor HTTP, y puede haber sido implementada lo mismo mediante un servidor dentro del proceso (una DLL) o fuera del proceso (un ejecutable).

El resto del ejemplo es predecible. Una vez abierta la consulta utilizamos la propiedad *EOF* y el método *MoveNext* para recorrerla. Cada vez que localizamos una fila,

añadimos al texto HTML el valor de la columna *au\_lname*, el apellido de un autor, más un cambio de línea:

```
Response.Write(consulta("au_lname"));
Response.Write("<br>");
```

Por supuesto, si nuestro objetivo fuese programar directamente en ASP sería sumamente conveniente aprender el manejo de las clases de ADO y, en general, de otras clases que acompañan a las implementaciones de ASP. Pero tengo otra proposición más interesante para usted.

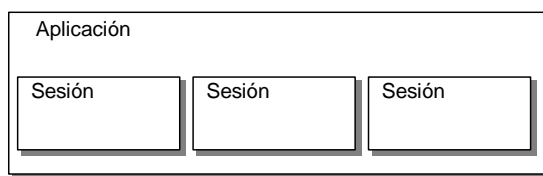
## Mantenimiento del estado

Uno de los grandes atractivos aparentes de las aplicaciones ASP es la posibilidad de crear variables globales a un conjunto de páginas. Los objetos globales *Application* y *Session* permiten almacenar información a la medida, utilizando como índice de búsqueda un nombre literal. Por ejemplo:

```
<% Session("Nombre") = Request.Form("Nombre"); %>
```

Estamos suponiendo que la página donde se encuentra la instrucción anterior se llama desde un formulario donde el usuario teclea su nombre. La instrucción asigna el valor recibido (se supone que con el método `POST`) dentro de una variable de sesión llamada también *Nombre*. Desde el momento en que se ejecuta esta asignación, podremos preguntar por el valor de dicha variable en cualquier página del mismo dominio, y recibiremos de vuelta el valor asignado.

Observe que he utilizado el objeto *Session* para almacenar las variables, pero también es posible utilizar *Application* con el mismo propósito. La diferencia consiste en que cada cliente que se conecta a la página tiene su propio juego de variables de sesión, pero todos comparten las mismas variables a nivel de aplicación.



El siguiente grupo de instrucciones demuestra de forma sencilla cómo se mantienen los valores de las variables de sesión de una página a otra. He supuesto que en *Session* se va a almacenar la variable *Veces* que actúa como un contador. Cuando un usuario entra por primera vez en la página, no se ha creado todavía la variable y el valor que se obtiene de la misma es el valor especial *undefined*. Para detectar esta situación se lleva a cabo la complicada conversión de la primera línea del *script*. De ahí en adelante, se modificará el valor de la variable de acuerdo al parámetro *dir* que se pasa junto con la URL.

```
<script runat="server" language="JScript">
    if (isNaN(parseInt(Session("Veces"))))
        Session("Veces") = 1;
    else if (Request.QueryString("dir") == 1)
        Session("Veces") = Session("Veces") + 1;
    else if (Request.QueryString("dir") == -1)
        Session("Veces") = Session("Veces") - 1;
    Response.Write("Veces: " + Session("Veces"));
    Response.Write("<br>");
</script>

<p><a href="prueba.asp?dir=1">Incrementa</a>
    &nbsp;|&nbsp;
    <a href="prueba.asp?dir=-1">Decrementa</a></p>
```

¿Se da cuenta de lo sencillo que se convierte el mantenimiento del estado de la aplicación gracias a estas variables? Evidentemente, hay felino encerrado: la implementación de variables de sesión se basa en el uso de *cookies*, ¿de qué otro modo podría ser? Si el cliente que se conecta a una página desactiva la recepción de *cookies* simplemente porque le da su real gana, nuestra aplicación ASP basada en variables de sesión se va ... digamos que a hacer encajes de bolillos.

De todos modos, ASP puede también prescindir del mantenimiento automático de variables; lógicamente, la velocidad de respuesta del servidor aumenta. Entonces podemos utilizar cualquiera de las técnicas que ya conocemos para mantener el estado: URL con parámetros o valores ocultos en formularios. Para desactivar las variables de aplicación y sesión hay que incluir la siguiente instrucción como primera línea de las páginas afectadas:

```
<%@ EnableSessionState=False %>
```

## Creación de objetos para ASP en Delphi

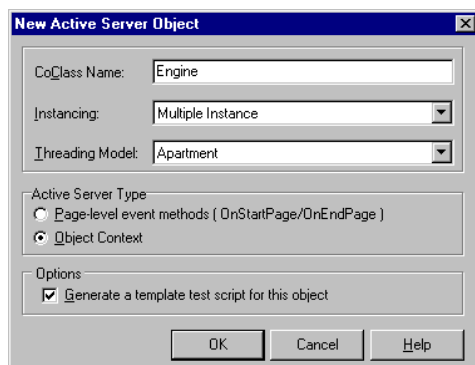
Si ASP nos permite manejar objetos ActiveX arbitrarios, ¿por qué no crear estos en Delphi? ¿Hay alguna particularidad que deba tener una clase ActiveX para poder utilizarse dentro de una página ASP? No, dentro de una aplicación ASP se pueden crear objetos ActiveX pertenecientes a clases arbitrarias. Supongamos que hemos creado una clase COM, que su identificador de programa es *ImUtils.Calendario*, y que uno de sus métodos cumple el tonto objetivo de devolver en una cadena de caracteres el código HTML que representa el calendario del mes actual. Podemos utilizar la clase del siguiente modo:

```
<script runat="server" language="JScript">
    var calendario = Server.CreateObject("ImUtils.Calendario");
    Response.Write(calendario.Imprimir);
</script>
```

Pero si queremos que la clase tenga alguna utilidad práctica, debemos ser capaces de acceder dentro de ella a los objetos globales de ASP. Como mínimo, al objeto *Response*. De este modo, nuestro generador de calendarios podría consultar los pará-

metros que se han pasado a la página, o escribir directamente sobre la respuesta, ahorrándonos código interpretado ASP.

Delphi nos ofrece un asistente para crear objetos COM que puedan interactuar con el motor de Active Server Pages. Se encuentra en la segunda página del Depósito de Objetos, y el icono tiene el subtítulo *Active Server Object*. Cuando se ejecuta el asistente aparece el siguiente cuadro de diálogo:



Los tres primeros campos de edición son los usuales en todo objeto ActiveX. El primero determina el nombre de la clase. Recuerde que este nombre se añade al del módulo DLL o ejecutable en el cual se define el objeto para formar el identificador de programa que se utiliza en la llamada a *CreateObject*. Por otra parte, el modelo de instancias y el de concurrencia se inicializan automáticamente con los valores más apropiados para este tipo de objetos.

A continuación hay que indicar qué tipo de objeto servidor queremos crear. El problema es que, de acuerdo a la versión de Internet Information Server que utilicemos existen diferentes métodos para que el objeto ASP reciba información sobre el contexto en el cual se ejecuta; la información de contexto sirve para que el objeto obtenga punteros a los objetos globales de ASP.

Las variantes de inicialización son las siguientes:

- Métodos de evento a nivel de página: Es el modo tradicional, empleado por las versiones de Internet Information Server anteriores a la 5. Se crea un objeto derivado de la clase *TASPObj*, suministrada por Delphi, y se implementan dos métodos, *OnStartPage* y *OnEndPage*, que son ejecutados convenientemente por el motor ASP. En particular, durante la llamada del primer evento se nos pasa como parámetro un puntero al contexto de ejecución. El código generado por Delphi se encarga de copiar este puntero en un atributo interno, y de implementar el acceso a los objetos globales mediante este puntero interno.
- Contexto de objeto: Es el modo de inicialización más moderno, y es el requerido por Internet Information Server 5. El objeto creado es un descendiente de la

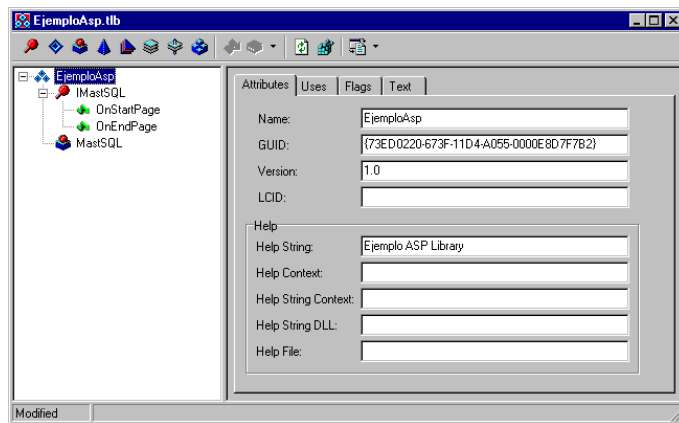


clase *TASPMTSObject*, y se nos presenta directamente con el acceso necesario al contexto de ejecución.

Por último, el asistente nos propone la creación de un simple *script* ASP, que puede ser de utilidad si no recordamos la sintaxis de la creación de objetos ActiveX.

## Un ejemplo elemental

Mostraré un ejemplo sencillo de objeto para ASP. Inicie un nuevo proyecto; éste puede ser un ejecutable o una DLL, pero la mayor eficiencia se logra con las bibliotecas de carga dinámica, no sólo por su mayor velocidad de carga, sino por la posibilidad de aprovechar los recursos de MTS. Aquí asumiré que hemos creado una DLL, y que su nombre es *EjemploAsp*. Al ser una DLL que contendrá objetos ActiveX, la mejor forma de crearla es utilizar el asistente *ActiveX Library* de la segunda página del Depósito de Objetos. A continuación, ejecute el asistente *Active Server Object*, y teclee *MastSQL* como nombre de la clase. Con los nombres utilizados, el identificador de programa que necesitaremos para crear posteriormente el objeto desde una página ASP será *EjemploAsp.MastSQL*. Por ser en estos momentos el caso más frecuente, utilizaré la inicialización del contexto al estilo de IIS 3 y 4.



Una vez terminada la ejecución del asistente, Delphi nos presenta el Editor de la Biblioteca de Tipos, en el que aparece la clase recién creada, la interfaz *IMastSQL* que soporta la clase y los métodos de dicha interfaz. Si selecciona el nodo raíz en este editor y activa la pestaña *Text* en el panel de la derecha, verá el equivalente en modo texto de la descripción gráfica de los objetos del nuevo servidor. El código, redactado en el lenguaje IDL, será más o menos similar al siguiente:

```
[
    uuid(6AB58501-5507-11D4-A055-0000E8D7F7B2),
    version(1.0),
    helpstring("Dispatch interface for MastSQL Object"),
    dual, oleautomation
]
```

```

interface IMastSQL: IDispatch
{
  [id(0x00000001)]
  HRESULT _stdcall OnStartPage( [in] IUnknown * AScriptingContext );
  [id(0x00000002)]
  HRESULT _stdcall OnEndPage( void );
};

```

Además, Delphi crea dos unidades. Al texto de la primera de ellas podemos llegar pulsando F12 sobre el editor de la biblioteca de tipos. Muy importante: ¡no debe modificar esta unidad manualmente! Delphi genera su contenido a partir de la Biblioteca de Tipos de la aplicación. Si modificamos algo en la unidad, probablemente se pierda cuando guardemos la aplicación o realicemos cualquier operación sobre la biblioteca de tipos.

La segunda unidad recibe un nombre muy ilustrativo, tal como *Unit1*, y contiene la implementación del objeto ASP. La declaración de la clase correspondiente es similar a la que sigue:

```

type
  TMastSQL = class(TASPObject, IMastSQL)
  protected
    procedure OnEndPage; safecall;
    procedure OnStartPage(const AScriptingContext: IUnknown);
      safecall;
  end;

```

Como vemos, Delphi suministra la clase base *TASPObject* con el propósito de implementar el esquema básico de interacción con ASP. Recuerde que si hubiésemos utilizado la forma alternativa de inicialización del contexto, la clase base habría sido *TASPMTSObject*.

Ahora estamos listos para añadir un nuevo método al objeto, que hasta el momento ha presumido de una inutilidad ejemplar. La adición se lleva a cabo en el Editor de la Biblioteca de Tipos<sup>39</sup>; si usted es de los que prefieren el café sin azúcar, puede hacerlo también directamente en IDL. En cualquier caso, declare un método *MostrarClientes*, con un parámetro entero nombrado *MaxRows*. Su propósito: mostrar un listado de clientes, recorriendo cada vez un número predeterminado de filas.

La implementación del nuevo método es responsabilidad nuestra, y puede parecerse al siguiente engendro:

```

procedure TMastSQL.MostrarClientes(MaxRows: Integer);
var
  LastCust: Integer;
begin
  with TTable.Create(nil) do
    try
      DatabaseName := 'DBDEMOS';
      TableName := 'CUSTOMER.DB';

```

---

<sup>39</sup> Me estoy hartando de utilizar mayúsculas cada vez que menciono el maldito editor...

```

Open;
LastCust := StrToIntDef(Session['LastCust'], -1);
if (LastCust > 0) and Locate('CustNo', LastCust, []) then
    Next;
LastCust := -1;
while not Eof and (MaxRows > 0) do
begin
    LastCust := FieldValues['CustNo'];
    Response.Write(FieldValues['Company'] + '<br>');
    Next;
    Dec(MaxRows);
end;
if LastCust = -1 then
    Response.Write('No existen (más) registros');
Session['LastCust'] := LastCust;
finally
    Free;
end;
end;

```

Observe cómo accedemos a las propiedades *Session* y *Response* de la clase *TMastSQL* para obtener la información de la página ASP. Un ejemplo decentemente programado necesitaría también, con toda probabilidad, utilizar *Request* para aprovechar los parámetros de llamada de la página.

#### NOTA APESADUMBRADA

Incluso este sencillo ejemplo nos ilustra un problema nada despreciable. ¿Se da cuenta de que hemos creado dinámicamente los objetos de acceso a datos? Cuando desarrollamos objetos ASP no disponemos de módulos de datos que sirvan de soporte a las conexiones y consultas necesarias.

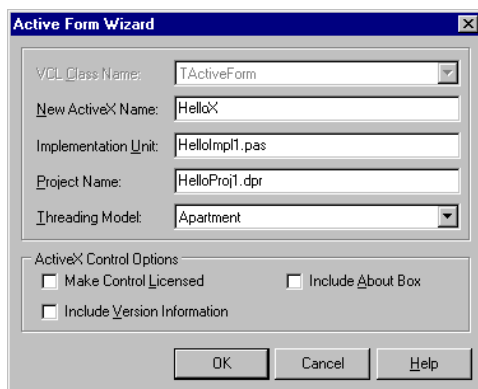
## ActiveForms: formularios en la Web

Regresamos al lado cliente de la red. Usted ya sabe cómo crear páginas atractivas y útiles, combinando sabiamente HTML, JavaScript y grandes dosis de suerte e intuición. Pero esta combinación no es lo suficientemente potente para abordar cualquier tarea de programación en el navegador. Si hace falta más potencia debemos entonces echar mano de algún lenguaje más complejo y general. Una de las alternativas consiste en el uso de *applets* de Java (no confundir con JavaScript). Estas serían pequeñas aplicaciones diseñadas para ejecutarse en el contexto del navegador de Internet, escritas en Java y traducidas a un código binario interpretable, que es independiente del procesador y del sistema operativo. Otra alternativa casi equivalente consiste en incluir dentro de la página controles ActiveX, que podemos programar con el propio Delphi. La desventaja de esta técnica es que los controles ActiveX nos atan a un tipo de procesador (Intel) y a un sistema operativo (Windows).

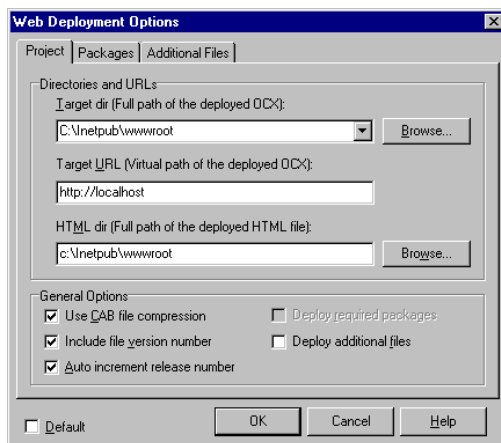
La forma más sencilla de incluir controles ActiveX en una página Web utilizando Delphi consiste en crear un *formulario activo*, o *ActiveForm*, que es sencillamente un control ActiveX con algunas características de un formulario “normal”: puede contener otros componentes VCL y puede actuar como receptor de los eventos por ellos

disparados. Para crear un formulario activo debemos ejecutar el asistente *ActiveForm* de la página *ActiveX* del Depósito de Objetos, que se muestra en la página siguiente.

El formulario activo debe crearse dentro del contexto de una biblioteca dinámica *ActiveX*, para obtener un servidor COM dentro del proceso. De no ser éste el caso, Delphi cierra el proyecto activo y crea un nuevo proyecto de este tipo. Observe que el modelo de concurrencia debe ser *Apartment*, como exige Internet Explorer:

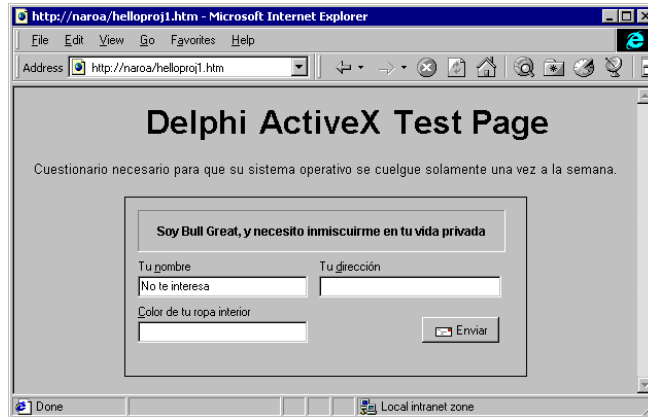


Para el programador, el formulario activo se comporta en tiempo de diseño como cualquier otro formulario. Así que puede colocar cuantos controles desee, asignar propiedades, crear manejadores de eventos, etc. Una vez terminado el diseño del formulario, debe ejecutar el comando de menú *Project | Web Deployment Options*:



Delphi genera automáticamente una página HTML de prueba para nuestra *ActiveForm*, y mediante el diálogo anterior podemos especificar las opciones de la página y la forma en la que vamos a distribuir el control. En la primera página, por ejemplo, debemos indicar dos directorios y una URL:

Opción	Significado
<i>Target dir</i>	Directorio donde se va a ubicar el control ActiveX compilado y listo para su distribución
<i>Target URL</i>	URL, vista desde un explorador Web, donde irá el formulario activo
<i>HTML dir</i>	Directorio donde se va a crear la página HTML



En el resto del diálogo podemos indicar si queremos comprimir el formulario, algo muy recomendable, si vamos a utilizar paquetes en su distribución, si necesitamos ficheros adicionales, etc. Una vez completado el diálogo, podemos llevar la página HTML y el formulario activo a su ubicación final mediante el comando *Projects | Web deploy*. La imagen anterior muestra un formulario activo cargado localmente con Internet Explorer.

Ahora bien, ¿qué posibilidades reales tenemos de crear aplicaciones serias de bases de datos con esta técnica? En primer lugar, podemos hacer que una aplicación basada en extensiones de servidor utilice formularios activos en el lado cliente como sustitutos de los limitados formularios HTML. Ganaríamos una interfaz de usuario más amigable, pero a costa de perder portabilidad.

La unidad *URLMon* contiene declaraciones de interfaces y rutinas para que nuestro formulario ActiveX pueda comunicarse con el navegador que lo está visualizando. En particular, pueden interesarnos las funciones con el prefijo *Hlink*, como *HlinkGoForward* y *HlinkGoBack*, que nos permiten cambiar la página activa. También podemos utilizar los componentes de la página *Internet* para enviar correo electrónico, o abrir *sockets* que se comuniquen con el servidor HTTP.

La otra oportunidad consiste en situar en el formulario activo componentes de acceso a bases de datos. ¿Quiero decir conjuntos de datos del BDE o de ADO? ¡No! En tal caso estaríamos complicando aún más las condiciones de configuración en el cliente. De lo que se trata es de utilizar clientes “delgados” como los que se pueden desarrollar con DataSnap, para lo cual debemos colocar un servidor de capa intermedia en una dirección accesible para los usuarios de Internet. El cliente se conecta-

ría al servidor utilizando esa dirección IP, o el correspondiente nombre de dominio, con un componente *TSocketConnection* o *TWebConnection*.

¿Cuál de los dos tipos de conexión es mejor? *TSocketConnection* es ligeramente más veloz que *TWebConnection*, que utiliza una vía de acceso con más estaciones intermedias, y que debe arrastrar la sobrecarga del protocolo HTTP. Las conexiones con *TSocketConnection* nos permiten, además, utilizar una clase COM de intercepción de datos, ya sea para comprimir o para codificar los paquetes que se envían y reciben. Y una tercera ventaja, que puede llegar a ser decisiva: el componente *TWebConnection* no permite que el cliente reciba eventos disparados por el servidor. Si usted necesita eventos, olvídense de *TWebConnection*.

Pero *TWebConnection* tiene también sus partidarios. En primer lugar, no hay que preocuparse por la presencia de cortafuegos; esto es lo más importante, en la mayoría de los casos. Además, *TWebConnection* nos ofrece un regalo casi gratuito: la posibilidad de implementar una “caché de servidores” de forma transparente, de forma tal que un objeto remoto creado por un cliente pueda ser reutilizado por otra conexión.

## Internet Express: por qué no

Hay otra técnica en Delphi que mezcla la programación para Internet con la programación COM: Internet Express. La idea que da vida a Internet Express es atractiva. Estas son las condiciones iniciales:

- 1 Tenemos un usuario final que utilizará un navegador de Internet para acceder a nuestra aplicación.
- 2 Ese navegador debe soportar una versión decente de JavaScript.
- 3 El usuario se conecta a una aplicación que puede haber sido programada con WebBroker o con WebSnap, indistintamente.
- 4 Al otro lado de la red, tenemos instalado un servidor de capa intermedia, que publica información de una base de datos y permite actualizarla, utilizando la resolución de DataSnap.
- 5 No es necesario que el cliente tenga acceso directo al servidor de capa intermedia. Este último puede estar escondido tras un cortafuegos, pongamos por caso.

Supongamos ahora que el usuario llena un formulario y pide datos de un grupo de productos. La aplicación genera la página con ayuda de los componentes de Internet Express: *TXMLBroker* y *TInetXPageProducer*. El primero es el encargado de conectarse al servidor DataSnap y poner sus datos a disposición del segundo. Este, a su vez, es un productor de página sui generis, del que daremos más detalles enseguida.

La característica que diferencia a Internet Express es que los datos de productos se envían en formato XML al navegador, igual que si se tratase del envío de un paquete de datos a una aplicación por medio de DataSnap. El usuario puede editar localmente esos datos, utilizando formularios HTML y enviar las modificaciones de vuelta al

servidor, que entonces las incorpora a la base de datos. En resumen: el funcionamiento es similar al de un sistema basado en DataSnap, pero sustituyendo la capa visual de presentación al usuario por páginas y formularios HTML.

Claro, para que un formulario HTML demuestre las habilidades mínimas de edición, hay que utilizar mucho código JavaScript en el lado cliente. Y esta es la causa del primer problema grave de Internet Express:

*"Internet Express depende demasiado de la versión del navegador que esté utilizando el usuario final"*

No es que sea pesimista y esté imaginando terrores que nunca sucederán: he pasado por el amargo trago de desarrollar una aplicación con Internet Express y ver cómo dejaba de funcionar cuando mi cliente instaló la novedad de aquel momento: Internet Explorer 5.

El segundo problema tiene un carácter más técnico: el componente *TInetXPageProducer* genera para nosotros páginas enteras, formularios y funciones en JavaScript, y es quien determina la apariencia de la aplicación. Para generar HTML, el componente utiliza un árbol de objetos que se configura en tiempo de diseño, dentro de Delphi.

Pues bien, eso es un error muy grave. Cada vez que haya que tocar cualquier tontería dentro de la página, tendrá que cargar el proyecto en el Entorno de Desarrollo... y ocuparse usted mismo los cambios: cámbiame el fondo de ese botón, por favor, pon en mayúsculas este aviso, mueve dos píxeles tal imagen, que así no me gusta. Al final, los costes de mantenimiento se disparan, y programar deja de ser rentable.

Pero el motivo principal por el que no incluí finalmente el capítulo que *ya* había redactado sobre Internet Express es mucho más prosaico: al hacer las pruebas con Delphi 6, encontré errores aterradores que me hicieron imposible desarrollar incluso los ejemplos más sencillos de esta técnica. Créame que lo siento, porque tuve que echar a la papelera el fruto de unas cuantas noches sin dormir, pero esto puede ser una indicación del interés que tiene Borland en esta técnica para próximas versiones.









## WebSnap: conceptos básicos

**S**I HACEMOS CASO A BORLAND, EL UNIVERSO entero ha estado conspirando durante milenios, creando y desbaratando imperios, enviando cometas en fechas oportunas para destrozarse dinosaurios, y moviendo otros hilos en las esferas exteriores, para que a principios del siglo XXI, los magos de Scotts Valley llenasen de gozo nuestros corazones al desvelar la existencia del grandioso ... (trompetas y redoble de timbales, please) ... ¡WebSnap!

Para mi desgracia, mientras más pasa el tiempo, más escéptico y cínico me vuelvo. Quisiera ser el propietario de una fe ciega pero luminosa, que me hiciera más fácil el Camino<sup>40</sup> ... pero antes que eso, me gustaría que para variar algún producto de software funcionase a la primera, que viniera con una documentación razonablemente completa y correcta. Y que por dos tonterías que le añadiesen al cabo de un año de sufrimiento, no me hicieran pagar una pasta para adquirir la actualización para la próxima gran conspiración del Universo.

### ¿En qué consiste WebSnap?

WebSnap es un *application framework* para el desarrollo de extensiones HTTP. Comparte muchas características con WebBroker: por ejemplo, permite que desarrollemos, con un solo conjunto de fuentes, extensiones CGI, ISAPI, NSAPI y módulos Apache. Pero su principal objetivo de diseño ha dado como resultado un producto final muy diferente. Ese objetivo fue facilitar el trabajo en equipo para el desarrollo de este tipo de aplicaciones. WebBroker, como es sabido, tenía dos serios inconvenientes:

- 1 Toda la programación suele concentrarse en un mismo módulo Web y, por consiguiente, en un único fichero físico. Si dos personas trabajan a la misma vez en la aplicación, tienen que pelearse para obtener acceso y modificar dicho fichero.
- 2 El sistema de etiquetas transparentes de WebBroker se lleva muy mal con la mayoría de los editores HTML.
- 3 Es necesario saber algo de magia para lograr un diseño flexible con el sistema de etiquetas tan elemental que ofrece WebBroker. Se echa de menos algunas ins-

---

<sup>40</sup> No sé por qué, pero los Místicos siempre usan Mayúsculas para cualquier Chorrada.

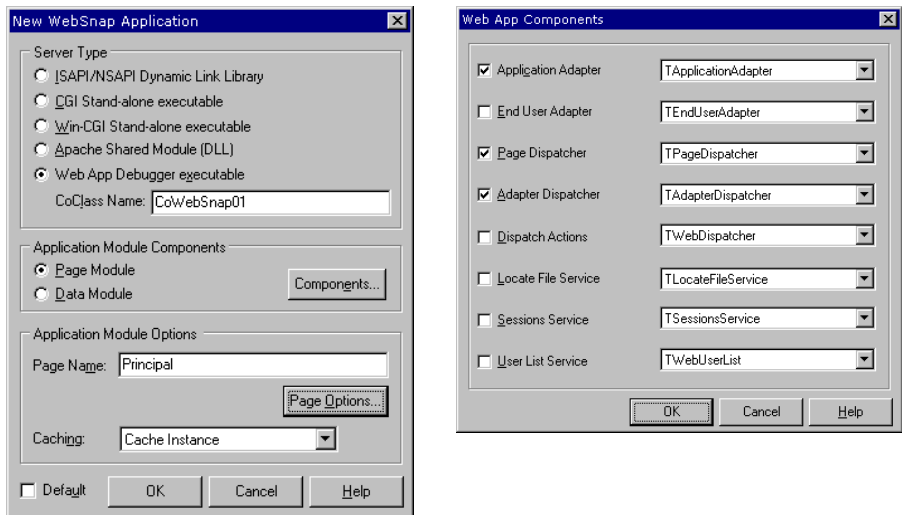




El primer botón lanza el asistente para crear una aplicación WebSnap. Los dos siguientes permiten añadir a una aplicación existente dos de los tipos especiales de módulos de WebSnap: módulos de páginas y módulos de datos especiales. El último botón se activa solamente cuando hay un fichero HTML asociado al módulo activo; ésta es una novedad en Delphi 6. Al hacerlo, se ejecuta un editor externo para el fichero HTML, que debemos haber configurado antes en la página *Internet* del diálogo *Tools | Environment Options*.

## El asistente para aplicaciones

Comencemos por crear una aplicación lo más sencilla posible, con la ayuda del asistente de WebSnap. El diálogo inicial es el que aparece a la izquierda:

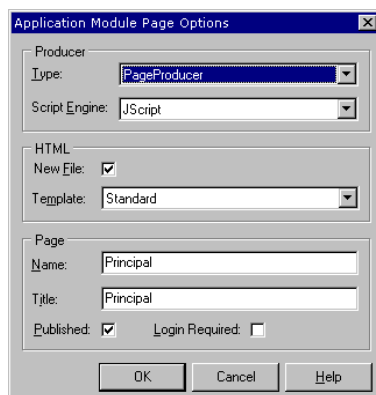


Ya conocemos el funcionamiento del grupo de botones superior, para indicar el tipo de aplicación que vamos a generar. Como siempre, le recomiendo elegir un ejecutable para el Web App Debugger, y darle un nombre extravagante a la clase COM que se va a generar.

En el siguiente grupo podemos pedirle al asistente que genere inicialmente un módulo de datos a secas, o un módulo que a su vez pueda generar el contenido de una página. Esto último es una de las peculiaridades de WebSnap: si seguimos las recomendaciones de Borland, cada página diferente debe ser generada por un módulo separado. En cualquiera de los dos casos, el asistente colocará una retahíla de com-

ponentes sobre el módulo generado. Más adelante veremos la función de cada uno de los componentes que se añaden, pero por ahora le bastará conocer que puede elegirlos pulsando el botón *Components* y modificando las opciones del cuadro de diálogo que aparece como respuesta (a la izquierda del diálogo inicial, en la imagen anterior).

Elegiremos un *Page Module*, para matar dos pájaros de un tiro. Al hacerlo, se activan los controles del próximo panel que permiten darle un nombre a la página principal, y configurar el mecanismo de generación de contenido para la misma.



También en este caso dejaremos las opciones que sugiere el asistente.



La imagen adyacente corresponde al módulo generado por el asistente. La primera novedad que podemos constatar incluso en este sencillo ejemplo es que los nombres de las rutas soportadas por la aplicación corresponden ahora a los nombres de los módulos del proyecto. Más adelante daré mi opinión sobre este asunto.

Por el momento, es más importante comprender el papel de esta turba de componentes que nos han arrojado encima. El principal de todos ellos es *WebAppComponents*, porque implementa la interfaz *IWebAppServices*, crucial para el desempeño de WebSnap. El componente es como un erizo, porque casi todas sus propiedades publicadas son de tipo puntero a interfaz y hacen referencia a otros componentes en los que delega parte de la funcionalidad de *IWebAppServices*. La siguiente relación de propiedades de *WebAppComponents* nos dará una idea de las posibilidades del componente:

- *AdapterDispatcher*, *DispatchActions*, *PageDispatcher*: Cuando la aplicación recibe una petición HTTP, *WebAppComponents* barre secuencialmente estas tres interfaces hasta que una de ellas dé respuesta a la petición. Cada una de las interfaz utiliza

un algoritmo diferente que reconoce diferentes formatos de URL. En breve daré más detalles.

- *ApplicationAdapter*, *EndUserAdapter*: Los componentes asociados son objetos *adaptadores* a los que podemos hacer referencia posteriormente desde el lenguaje de *script*.
- *LocateFileService*: WebSnap, al igual que WebBroker, utiliza plantillas y otros objetos externos con profusión. El componente asociado a esta propiedad puede utilizarse para controlar dónde se sitúan estos ficheros.
- *UserListService*: Permite implementar una lista de usuarios reconocidos por la aplicación, y otorgarles derechos de acceso diferenciados.
- *Sessions*: El componente que se asocia a esta propiedad debe ayudar a implementar variables de sesión persistentes, al estilo de las utilizadas en ASP.

Muchas de estas propiedades pueden vivir felizmente con referencias nulas. Como comprobará, el asistente de WebSnap solamente proporciona componentes para dos de los componentes de tratamiento de peticiones, *AdapterDispatcher* y *PageDispatcher*, y para uno de los componentes adaptadores: *ApplicationAdapter*. El asistente añade además un *PageProducer*, idéntico al que conocimos al estudiar WebBroker, pero esta vez con la posibilidad de interpretar instrucciones de *script* dentro de sus plantillas.

## Módulos de páginas y módulos de datos

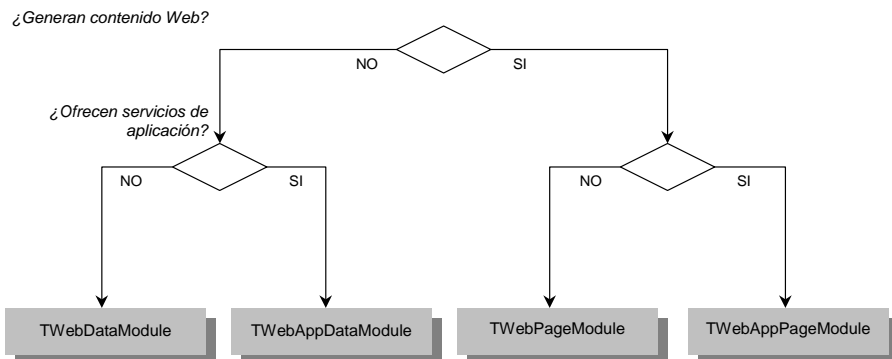
Si vamos al código fuente, comprobaremos que el módulo creado pertenece a una clase nueva en Delphi 6, *TWebAppPageModule*. Es así porque en el asistente habíamos elegido tener un *Page Module*, en vez de un *Data Module*. Pero si hubiéramos elegido el segundo, tendríamos una instancia de la clase *TWebAppDataModule*; recuerde que WebBroker utilizaba *TWebModule* a secas.

```
type
  TPrincipal = class(TWebAppPageModule)
    PageProducer: TPageProducer;
    WebAppComponents: TWebAppComponents;
    ApplicationAdapter: TApplicationAdapter;
    PageDispatcher: TPageDispatcher;
    AdapterDispatcher: TAdapterDispatcher;
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

En realidad, tenemos dos clases de módulos adicionales: *TWebPageModule* y *TWebDataModule*. Estas han perdido la partícula *App* de su nombre, porque se crean cuando ya hay un módulo “principal” dentro de la aplicación, que contiene los componentes necesarios para los servicios de aplicación.

Hay varios motivos para ser generosos y crear tantas clases. Uno de ellos es que WebSnap, en contraste con WebBroker, está más orientado al trabajo con interfaces

que con referencias de objetos. Estos nuevos módulos implementan varios tipos de interfaces. Por ejemplo, las dos clases de módulos “principales”, las que tienen *App* en sus nombres, implementan *IGetWebAppServices*; los módulos de páginas se hacen cargo de *IPageResult* e *IGetProducerComponent*, que no son implementadas por los módulos estrictamente de datos.



Los módulos de WebSnap tienen otra peculiaridad. Cada formulario o módulos de datos “normal” va asociado a una variable global que se define inmediatamente después de la declaración de la clase, y que sirve para que Delphi almacene ahí la referencia al objetos cuando se produce la creación automática de instancias. En cambio, esto es lo que tienen los nuevos módulos en lugar de la variable:

```
function Principal: TPrincipal;
```

No es algo descabellado, porque en Delphi no hay distinciones al nivel sintáctico entre variables y funciones sin parámetros (y eso es bueno, adoradores de C++ y Java!). Si no nos hubiéramos percatado de que la variable había sido sustituida por una función global sin parámetros, Delphi nos habría dejado seguir tranquilamente ... hasta el momento en que intentásemos asignar algo a la pseudo variable.

La implementación de la función es la siguiente:

```
function Principal: TPrincipal;
begin
  Result := TPrincipal(WebContext.FindModuleClass(TPrincipal));
end;
```

Para rematar, el código de inicialización contiene unas misteriosas instrucciones:

```
initialization
  if WebRequestHandler <> nil then
    WebRequestHandler.AddWebModuleFactory(
      TWebAppPageModuleFactory.Create(TPrincipal,
        TWebPageInfo.Create([wpPublished], '.html'), caCache));
end.
```



Con todos los elementos anteriores, es más fácil explicar y entender lo que sucede. Estos módulos pueden residir en un CGI ejecutable, que solamente atiende un hilo por proceso, pero es también probable que pertenezcan a una extensión ISAPI, NSAPI o Apache, con el potencial de admitir varias peticiones en distintos hilos. WebSnap crea una instancia derivada de la clase *TAbstractWebContext* para cada uno de los hilos concurrentes, y la utiliza como punto de partida para localizar los objetos asociados al hilo, y por lo tanto, a la petición. La función global *WebContext* siempre devuelve el puntero a la instancia que corresponde al hilo activo.

Por cada unidad que contiene un módulo Web, se ejecutan instrucciones de inicialización similares a la de nuestro ejemplo, que sirven para registrar el *tipo de clase de módulo* implementado en la unidad. Posteriormente, cuando en algún lugar de la aplicación se llama a la función que suponemos que nos devolverá una instancia del módulo (la función *Principal* en nuestro ejemplo), en realidad se hace una búsqueda en el contexto Web correspondiente al hilo, a ver si ya existe o no esa instancia. En caso negativo se crea, por supuesto.

Gracias a este mecanismo algo complejo, no se crea ningún módulo inútil al iniciarse la petición, sino que cada módulo se crea justo en el momento en que lo necesitamos. Sepa también que el mecanismo de inicialización de componentes desde un recurso *dfm* utiliza también las funciones de instancias. Supongamos que cargamos un módulo que contiene a su vez otro componente, y que una propiedad de este último hace referencia a un componente ubicado en otro módulo. Digamos que la referencia es algo como *ModuloDatos.TablaClientes*. Cuando se termina la lectura de valores desde el recurso *dfm*, se efectúa la tradicional pasada de resolución de referencias. El puntero a la tabla de clientes se obtiene automáticamente llamando a la función global *ModuloDatos*; aquí podría crearse el módulo por primera vez o encontrarnos con una instancia ya creada. Finalmente, con el puntero obtenido se buscaría el componente *TablaClientes*, y la referencia al mismo se sustituiría en la propiedad que no habíamos terminado de inicializar.

## Generación de contenido

Hasta el momento, me he limitado a explicar el esqueleto generado por el asistente de WebSnap. Pero ya es hora de explicar también cómo se realiza el tratamiento y distribución de las peticiones y cómo generamos una respuesta HTML.

Como dije hace poco, hay tres algoritmos de despacho de peticiones en WebSnap, y son ejecutados por los componentes que se asocian a las propiedades *AdapterDispatcher*, *DispatchActions* y *PageDispatcher* del *TWebAppComponents* que se implementa los servicios Web de la aplicación. Cuando llega una petición a la aplicación, se ensaya cada componente, si está presente, en el siguiente orden:

- 1 *AdapterDispatcher*: Veremos dentro de poco que los adaptadores son componentes VCL a los que podemos hacer referencia desde el lenguaje de *script*. Estos

componentes pueden definir *acciones*; no las confunda con las acciones de WebBroker, que aunque comparten nombre, representan distintos conceptos. Podemos tener, por ejemplo, un adaptador que represente un conjunto de datos. Entonces una de las acciones posible sería *DeleteRow*, eliminar una fila. En el *script* podríamos hacer que la acción generase un botón de un formulario. Cuando pulsemos ese botón para pedir el borrado de la fila activa, la petición que se genera debe ser tratada por la acción del adaptador correspondiente. Este tipo de peticiones son detectadas por *AdapterDispatcher* y encaminadas al componente que genera la respuesta.

- 2 *DispatchActions*: El tipo de esta propiedad es *IWebDispatchActions* ... y da la casualidad de que el componente *TWebDispatcher*, que ya vimos con WebBroker, la implementa. Esto quiere decir que podemos realizar el tratamiento de algunas peticiones igual que lo hacíamos en WebBroker. No obstante, el asistente no genera un componente *TWebDispatcher* con las opciones por omisión. De todos modos, es tranquilizante saber que las viejas acciones “siguen ahí”...
- 3 *PageDispatcher*: Si una petición sobrevive a los componentes anteriores sin ser tratada, va a parar al despachador de páginas. Abra bien los ojos, porque este sistema será el utilizado con mayor frecuencia para pedir páginas a la aplicación.

Veamos los detalles del funcionamiento de *PageDispatcher*. Cuando recibe una petición, extrae de la misma la *ruta o path info*, al igual que sucedía en WebBroker. Pero en vez de recorrer una lista de acciones *TWebActionItem*, el nuevo componente recorre todos los módulos registrados en la aplicación. Recuerde que el registro se producía al ser ejecutado el código de inicialización de las unidades, y que consistía en una llamada al método *AddWebModuleFactory* de cierta colección global. Al registrar el tipo de clase, se le asociaba un objeto de la clase *TWebPageInfo*, que se creaba con la ayuda del siguiente constructor:

```
type
  TWebPageAccessFlags = (wpPublished, wpLoginRequired);
  TWebPageAccess = set of TWebPageAccessFlags;

constructor TWebPageInfo.Create (
  AAccess: TWebPageAccess = [wpPublished];
  const APageFile: string = '.html';
  const APageName: string = '';
  const ACaption: string = '';
  const ADescription: string = '';
  const AViewAccess: string = '');;
```

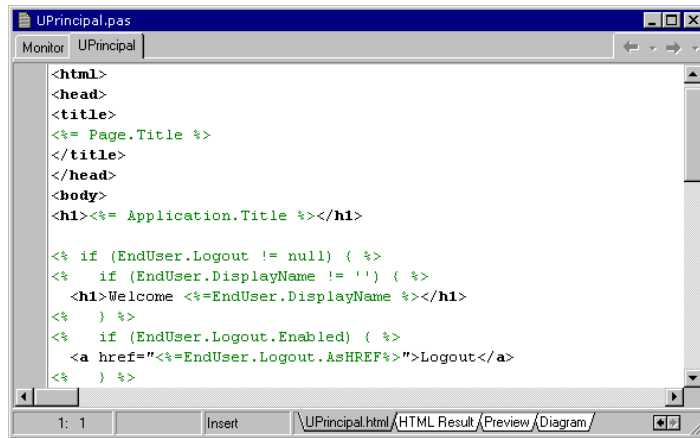
Normalmente, el nombre de la página asociada a un módulo es el mismo nombre (propiedad *Name*) del módulo, pero podemos cambiar ese nombre al registrar la clase del módulo. Si no se encuentra un módulo apropiado, la aplicación puede registrar una página por omisión.

¿Qué sucede cuando se localiza el módulo correcto? Los módulos de página tienen una propiedad *PageProducer*, del tipo *IProducerContent*, el cual puede apuntar a cual-

quiera de los componentes productores de contenido que ofrece WebSnap. Lo que queda entonces es pedirle a ese componente que genere la respuesta a la petición.

## Productores de páginas con superpoderes

Seleccione el módulo del ejemplo que hemos generado con el asistente, y salte al código fuente con la tecla F12. Una vez que tenga delante el Editor de Código de Delphi, observe las pequeñas solapas de la parte inferior de la ventana de edición:



¡Esto es nuevo! De la misma forma en que podemos asociar un fichero *dfm* a una unidad, ahora también podemos asociarle un fichero HTML. De hecho, el código fuente de la unidad *UPrincipal* contiene la siguiente línea al principio de la sección de implementación:

```
{ $R *.dfm } { *.html }
```

Note, sin embargo, la diferencia: el fichero no se incluye como recurso dentro de la unidad, sino que simplemente se advierte a la unidad de la existencia de ese fichero. Este último queda fuera de la aplicación, incluso en tiempo de ejecución. En cierto modo, es como si hubiéramos asignado el nombre del fichero en la propiedad *HTMLFile* del productor de páginas, pero sin preocuparnos esta vez por el directorio concreto desde donde se ejecuta la extensión HTTP.

¿Quiere esto decir que los componentes de WebSnap van siempre a buscar el fichero de plantilla en el directorio de la aplicación? No: esto sucede así sólo por omisión, pero podemos cambiar este comportamiento con la ayuda de un componente que implemente la interfaz *ILocateFileService...* como *TLocateFileService*.

Por ejemplo, podemos querer que las plantillas estén en un subdirectorio del directorio de la aplicación, de nombre *plantillas*. Para conseguirlo, basta con añadir un componente *TLocateFileService* al módulo de servicios de aplicación, es decir, al módulo que contiene el componente *TWebAppComponents*. Este último, por su parte, recono-

cerá automáticamente la presencia del nuevo componente y modificará su propiedad *LocateFileService* para que apunte a él. Debemos entonces interceptar su evento *OnFindTemplateFile*:

```
procedure TmodData.LocateFileServiceFindTemplateFile (
  ASender: TObject; AComponent: TComponent;
  const AFileName: string; var AFoundFile: string;
  var AHandled: Boolean);
begin
  AFoundFile := 'plantillas\' + AFileName;
  AHandled := True;
end;
```

También podemos aprovechar esta posibilidad para implementar páginas en varios idiomas. Una parte importante del trabajo la resolveríamos creando varios subdirectorios de plantillas, uno para cada idioma soportado. La aplicación debería ser capaz de reconocer en todo momento en qué idioma prefiere trabajar el usuario; más adelante, veremos cómo utilizar las *sesiones* de WebSnap para este objetivo. A partir de la información sobre el idioma preferido, es un juego de niños hacer que el servicio de localización de archivos indique el subdirectorio de búsqueda apropiado.

### CONSEJO

Mientras está ejecutando una aplicación WebSnap con la ayuda del Web App Debugger, puede modificar las plantillas que haya cargado dentro del editor de Delphi. Para ver los cambios, basta con que actualice el contenido del navegador. Esta técnica será la que utilizaremos en la siguiente sección para presentar el lenguaje de *scripting*.

## Scripts en WebSnap

Al inicio del capítulo comenté que WebSnap permitía utilizar lenguajes de *script* dentro de las plantillas. Cuando una plantilla es procesada por un productor de páginas o cualquier otro generador de contenido, además de expandir las etiquetas transparentes al estilo WebBroker, se utiliza simultáneamente el motor de Active Scripting para evaluar las instrucciones que se encuentren dentro de los delimitadores de código utilizados por ASP: `<% y %>`. En ese momento, mencioné algunas de las ventajas:

- 1 Mayor compatibilidad con los editores HTML más populares.
- 2 Más potencia, al permitir el uso de condicionales y bucles.

Pero si las ventajas se limitasen a los dos puntos anteriores, quizás no merecería la pena utilizar un sistema tan complejo como WebSnap. Eso quiere decir que hay más en juego...

Aclaremos antes un malentendido frecuente:

*“No es lo mismo Active Scripting que ASP”*

La relación entre ellos consiste en que ASP está basado en el motor de *scripts* genérico que es Active Scripting, pero una parte muy importante de ASP es la definición de su biblioteca de clases y objetos. En ASP, por ejemplo, tenemos una variable global llamada *Form*, que nos permite acceder al contenido de una petición POST. *Form* corresponde a un objeto definido por ASP, no por Active Scripting. Y eso quiere decir que no podemos utilizar esa variable, sin más, en un *script* escrito para WebSnap.

¿Cuáles son, entonces, las variables globales que estarán siempre disponibles dentro de un *script* de WebSnap?

- *Response*  
Funciona como en ASP. Normalmente nos limitaremos a usar su método *Write*. También se pueden abreviar las llamadas a *Response.Write* incluyendo una igualdad inmediatamente después del inicio del bloque de *script*: Las dos instrucciones siguientes son equivalentes:

```
<% Response.Write(Application.Title) %>
<%= Application.Title %>
```

- *Application*  
Este objeto corresponde al componente *ApplicationAdapter* del módulo de servicios de aplicación, y tiene muy pocas propiedades predefinidas. Acabamos de ver una de ellas: *Title*, que se extrae de la propiedad *ApplicationTitle* del componente adaptador. Poco uso se le puede dar al título de la aplicación, pero más adelante veremos que podemos añadirle campos a este objeto manipulando el componente correspondiente dentro de Delphi. Es cierto que podemos hacer lo mismo con un componente *TAdapter*, pero *Application* tiene la ventaja de ser visible desde todas las páginas, sin necesidad de calificar la referencia.
- *Request*  
No se entusiasme demasiado, porque solamente tiene tres propiedades: *Host*, *PathInfo* y *ScriptName*, que corresponden a fragmentos de la URL de la petición.

### ¿CURIOSIDAD?

Usted se preguntará cómo puedo estar seguro de que la documentación no miente, o si estoy seguro de que no hay más propiedades o métodos “ocultos”. Bueno, hay una forma muy sencilla de averiguarlo; mientras el código fuente no mienta, claro. Puede buscar las llamadas al método *RegisterScriptClass*, que Delphi utiliza para registrar las clases que describen los tipos de datos accesibles desde una plantilla. Con esta información y con un poco de paciencia...

- *Modules, Pages*  
Permiten acceder a todos los módulos y páginas de la aplicación. Pueden utilizarse de dos formas diferentes. La primera requiere que utilicemos un objeto *enumerador*. Por ejemplo, el siguiente código muestra el nombre de cada uno de los módulos de la aplicación.

```

<%
    var s = "Modulos: ";
    for (var e = new Enumerator(Modules); !e.atEnd(); e.moveNext()) {
        Response.Write(s + e.item().Name_);
        s = ", ";
    }
%>

```

Los métodos de un *Enumerator* son siempre los mismos. Lo que varía en cada caso es el tipo de objeto asociado a su función *item*. En este ejemplo, cada elemento representa un módulo, con las propiedades *Name\_*, *ClassName\_* y *Objects*. Con este último podemos también utilizar un enumerador. Veremos más ejemplos a lo largo de este capítulo.

La otra forma de utilizar *Modules* y *Pages* es acceder a uno de sus elementos individuales utilizando su nombre como si de una propiedad se tratase. Supongamos que una de las páginas de la aplicación se llama *Principal*. El siguiente *script* crearía un enlace a dicha página:

```

<a href="<%=Pages.Principal.HREF%>"><%=Pages.Principal.Title%></a>

```

Este otro ejemplo utiliza una instrucción **with** de JScript para evitar repetir el camino hasta el objeto, pero es equivalente al anterior:

```

<% with (Pages.Principal) { %>
<a href="<%=HREF%>"><%=Title%></a>
<% } %>

```

- *Page, Producer*

Estas variables hacen referencia a la página que estamos generando, y al generador de contenido asociado a la misma. Del productor, en particular, se utiliza su método *Write*, porque permite expandir etiquetas transparentes en el caso en que las hayamos definido.

- *Session, EndUser*








La primera permite el acceso a variables de sesión, como en ASP. Y la segunda suministrar información sobre el usuario final: si ya se ha identificado, cuál es su identificador, etc. En teoría, no todas las aplicaciones están obligadas a implementar estas características.

## Adaptadores

No obstante, la potencia de WebSnap reside en la posibilidad de registrar de manera relativamente sencilla más objetos, para que puedan ser manipulados desde el lenguaje de *script*. Si estamos creando una tienda virtual, es muy probable que necesitemos objetos como *Cesta* (de la compra), *Cliente* y *Producto*. La técnica con que contamos para la creación de estos objetos es el uso de componentes *adaptadores* en los módulos WebSnap.

Desde el punto de vista de Delphi, un adaptador es un objeto que pertenece a alguna clase derivada del componente *TCustomAdapter*; el objeto *ApplicationAdapter* que crea automáticamente el asistente de WebSnap es precisamente un tipo especial de adaptador. Desde el punto de vista del lenguaje de *script*, cada instancia de estos componentes se percibe como un objeto.

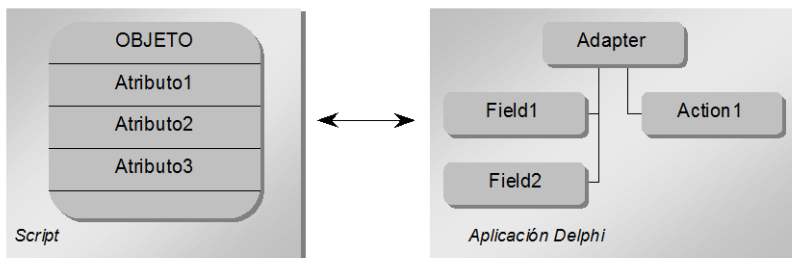
No voy a presentar una jerarquía de herencia para los adaptadores, principalmente porque lo más importante para un adaptador es la implementación de ciertas interfaces, más que si desciende de la casa de Lancaster o de la de York. Estas son las clases de adaptadores definidas por WebSnap:

Clase	Propósito
<i>TAdapter</i> 	Es el adaptador “básico”. No hace nada por sí mismo, pero podemos interceptar eventos suyos y de algunos de sus subcomponentes para que sirva de algo.
<i>TPagedAdapter</i> 	Es un adaptador de carácter general que permite dividir una lista de elementos entre varias páginas. La funcionalidad concreta también se define a golpe de manejadores de eventos.
<i>TDataSetAdapter</i> 	¡Por fin un adaptador que sabe lo que tiene que hacer! Como puede imaginar, permite manipular un conjunto de datos desde un fichero <i>script</i> .
<i>TLoginFormAdapter</i> 	Permite manejar un formulario de identificación. Veremos en breve que en WebSnap es aconsejable utilizar adaptadores para manejar los datos de formularios.
<i>TApplicationAdapter</i> 	Este adaptador sirve por omisión para muy pocas cosas. Pero el asistente de WebSnap lo añade automáticamente a todos sus proyectos. Antes de traer un <i>TAdapter</i> para entrenarlo mediante eventos, puede echar mano de este componente... que pasaba por allí, por casualidad...
<i>TEndUserAdapter</i> 	Aunque el icono parezca representar a una lombriz mutante adorando a un ídolo de metal, este componente controla los datos básicos de un usuario final en una aplicación que requiera identificación.
<i>TEndUserSessionAdapter</i> 	Es una variante más evolucionada de la anterior lombriz mutante, que permite almacenar más datos sobre el anélido mediante un servicio de sesiones.

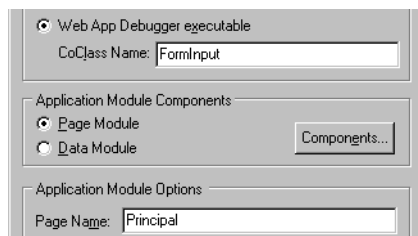
## Campos y acciones de adaptadores

Los “objetos” suelen tener *atributos* y *métodos*. Los métodos de un objeto de *script* vienen predeterminados por la implementación del correspondiente objeto en Delphi, pero es posible ampliar el conjunto de atributos o propiedades disponibles en el

lenguaje de *script*. Para este propósito, todo adaptador publica dos propiedades de tipo colección: *Data* y *Actions*:



Aclaremos una equivocación frecuente: tanto los campos de adaptadores como las acciones se ven como propiedades en el lenguaje de *script*; es falso que las acciones correspondan a métodos. Sin embargo, la confusión viene originada porque las acciones de adaptadores sirven realmente para disparar métodos... pero pertenecientes al objeto adaptador definido por Delphi dentro de la aplicación WebSnap. Veámoslo con un ejemplo:



Inicie una nueva aplicación WebSnap, preferiblemente como un ejecutable para el Web App Debugger. Llame a la nueva clase *FormInput*, utilice un módulo de página para colocar los componentes de aplicación, y llame *Principal* a la página que se va a generar. Guarde el proyecto con el nombre de *FormInput*; de esta manera, siempre que esté activo el Web App Debugger, podrá ejecutar la aplicación desde su explorador de Internet tecleando la siguiente URL:

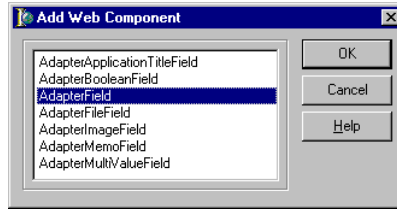
`http://localhost:1024/FormInput.FormInput`

Seleccione el componente *ApplicationAdapter* y haga doble clic sobre él, para que aparezca el editor de su propiedad *Fields*. Inicialmente, la colección estará vacía. Aquí sucede lo mismo que con los objetos de acceso a campos de un conjunto de datos, que se generan automáticamente si no los definimos en tiempo de diseño. Cada clase de adaptador tiene una serie de campos predefinidos. En el caso de un adaptador de aplicación, hay sólo uno: el que representa el título de la aplicación.

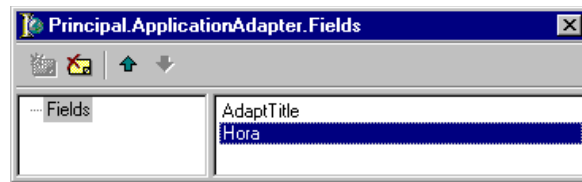
Si queremos crear campos adicionales, debemos primero hacer explícitos los campos predefinidos, ejecutando el comando *Add all fields* del menú de contexto del editor.



Luego hay que el comando *New component*. Aparecerá un diálogo en el que debemos elegir una de las clases de campos disponibles para ese adaptador:



Como el objetivo de un campo es almacenar e incluso permitir la modificación de una propiedad de un objeto, la diferencia entre las clases tiene que ver con el tipo de datos de la propiedad. Seleccionaremos un *AdapterField*, que es apropiado para cualquier tipo de dato representable mediante una cadena de caracteres. Cambie entonces el nombre del nuevo campo para que sea *Hora*:



El siguiente paso consiste en implementar dos eventos del nuevo componente, *OnGetValue* y *OnGetDisplayText*. En el primero asignaremos que tendrá la propiedad *Value* del campo en el lenguaje de *script*, y en el segundo, el de *DisplayText*:

```

procedure TPrincipal.HoraGetValue(Sender: TObject;
    var Value: Variant);
begin
    Value := Now;
end;

procedure TPrincipal.HoraGetDisplayText(Sender: TObject;
    var Value: string);
begin
    Value := FormatDateTime('hh:nn:ss', Now);
end;

```

Para probar nuestro nuevo campo de adaptador, seleccione la plantilla en el editor del Entorno de Desarrollo, y añada estas tres líneas en donde más le guste:

```

Hora: <%= Modules.Principal.ApplicationAdapter.Hora.Value %><br>
Hora: <%= Application.Hora.Value %><br>
Hora: <%= Application.Hora.DisplayText %><br>

```

En la primera de las líneas, he querido mostrar la ruta de acceso completa hasta el campo: comenzamos por la variable global *Modules*, que es una colección; hacemos referencia a uno de sus elementos por su nombre, *Principal*; dentro de este módulo elegimos un componente adaptador, *ApplicationAdapter*. Y, finalmente, dentro del

adaptador elegimos el campo *Hora* y su propiedad *Value*. Naturalmente, es más sencillo si aprovechamos que *Modules.Principal.ApplicationAdapter* puede abreviarse como *Application*, a secas.

## Principal

Hora: 24/11/01 22:03:26

Hora: 24/11/01 22:03:26

Hora: 22:03:26

Observe que en la tercera línea he mencionado la propiedad *DisplayText*, en vez de *Value*. La diferencia salta a la vista, ¿verdad?

## Acciones y formularios

Aprovecharé la misma aplicación para mostrar qué es una acción y cómo funciona. Vamos a añadir a la página de prueba un formulario con un cuadro de edición y un botón de envío, con el título *Grabar*. Queremos que el usuario introduzca en el formulario para enviar un mensaje de texto al servidor, y que el servidor almacene esos mensajes dentro de un fichero plano.

Hay que añadir dos componentes dentro de *ApplicationAdapter*:

- 1 El primero será un campo, al que llamaremos *Mensaje*.
- 2 El otro será una acción y la añadiremos, como es natural, dentro de la propiedad *Actions*. La llamaremos *Grabar*. No hace falta crear manejadores de eventos todavía, ni para la acción ni para el campo.

El escenario del conflicto se traslada a la plantilla. Vaya al final de la misma y, si no es mucho pedirle, teclee algo parecido al siguiente texto:

```
<form name="Grabacion" method="post">
  <input type="hidden" name="__act">
  <input type="text"
    name="<%=Application.Mensaje.InputName%>"
    value="<%=Application.Mensaje.EditText%>">
  <input type="submit" value="Grabar mensaje"
    onclick="Grabacion.__act.value='<%=Application.Grabar.
      LinkToPage(Page.Name, Page.Name).AsFieldValue%>'">
</form>
```

¡Cálmese por favor, que no cunda el pánico! Esto es sólo un entrenamiento; en la vida real nadie le exigirá nunca que diseñe sus formularios de esta manera tan espantosa. ¿Le confieso la verdad?, copié el formulario de otro ejemplo. Y más adelante veremos cómo utilizar el componente *TAdapterPageProducer* para que sea Delphi, y no nosotros, quien se encargue de conectar todos los elementos como en el formulario de la discordia.

De todos modos, analizaremos cada una de las partes del formulario. Comencemos con la cabecera:

```
<form name="Grabacion" method="post">
```

Lo principal es que no hay un atributo `ACTION`; eso significa que cuando pulsemos el botón de *Grabar* será la propia página *Principal* la que, además de generar el formulario, se ocupe de la grabación. El nombre del formulario es arbitrario; da lo mismo *Grabacion* que *OdioWebSnap*.

```
<input type="hidden" name="__act">
```

El nombre de este campo oculto *sí* que es importante, por lo que aclaro que comienza con dos subrayados consecutivos. Podríamos asignarle su valor directamente, pero Delphi prefiere que lo hagamos al pulsar el botón de grabar:

```
<input type="submit" value="Grabar mensaje"
  onclick="Grabacion.__act.value='<%=Application.Grabar.
  LinkToPage(Page.Name, Page.Name).AsFieldValue%>'">
```

La flecha retorcida es una adición mía, para indicar que tuve que romper la línea. Lo que hace la instrucción anterior es pedir, al objeto que representa a la acción en JavaScript, que ejecute su función *LinkToPage* para calcular una dirección de enlace, y que nos la devuelva en forma de cadena. Esto último se debe a que utilizamos la propiedad *AsFieldValue* del enlace; la alternativa sería utilizar *AsHRef*, que podría utilizarse como una URL “normal”. Para que satisfaga su curiosidad, este es el resultado de *AsFieldValue*:

```
__p.9.Principal__id.35.Principal.ApplicationAdapter.Grabar
```

La jerigonza anterior es tan clara como el evangelio... para Delphi, por supuesto, que la traduce más o menos como una lista de parámetros:

```
__p=Principal
__id=Principal.ApplicationAdapter.Grabar
```

Observe que esos números misteriosos, el 9 y el 35, indican simplemente el número de caracteres utilizados para representar el valor de cada parámetro.

Además del campo oculto, tenemos el cuadro de edición donde el usuario debe teclear su mensaje. Para garantizar que no se estropee el formulario si cambiamos el nombre del componente, utilizamos la propiedad *InputName* para el nombre del control:

```
<input type="text"
  name="<%=Application.Mensaje.InputName%>"
  value="<%=Application.Mensaje.EditText%>">
```

¿Qué sucede cuando la página recibe de vuelta los datos del formulario? En ese momento entra en escena el componente *AdapterDispatcher*, que busca dentro de los campos de la petición, para ver si existe uno llamado \_\_act; con dos subrayados, eso es. Si lo encuentra, utiliza el valor asociado para localizar la página y el componente exacto que tiene que ocuparse de la acción. Entonces se dispara el evento *OnExecute* de la acción. Pero antes, Delphi se encarga de recorrer todos los campos de la petición, para asignar las propiedades *ActionValue* de cada campo del adaptador. Esto es muy importante, porque es ahí donde recibiremos el mensaje que tenemos que grabar:

```

procedure TPrincipal.GrabarExecute(Sender: TObject;
  Params: TStrings);
var
  T: TextFile;
begin
  AssignFile(T, ChangeFileExt(ParamStr(0), '.txt'));
  try Append(T); except Rewrite(T); end;
  try
    WriteLn(T, Mensaje.ActionValue.Values[0]);
    // ¡En este ejemplo no se utiliza Params!
  finally
    CloseFile(T);
  end;
end;

```

Lo único interesante del manejador de eventos anterior es la propiedad donde se busca el valor tecleado por el usuario:

```
Mensaje.ActionValue.Values[0]
```

¿Por qué un vector? Pues porque existen campos que pueden almacenar varios valores simultáneamente. En un formulario HTML estos campos se editarían dentro de un `<SELECT>` con múltiple selección, o mediante un grupo de casillas, todas con el mismo nombre. En ese caso, nos interesaría también la siguiente propiedad, para averiguar cuántos valores ha seleccionado el usuario.

```
Mensaje.ActionValue.ValueCount
```

#### NOTA

Observe que en el manejador de eventos no hemos utilizado el parámetro *Params*, que a primera parece más apropiado. De esto nos ocuparemos en la siguiente sección.

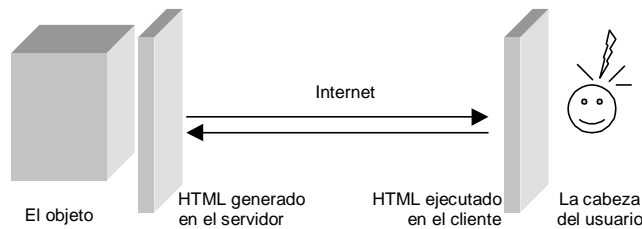
## Interpretación del concepto de acción

El ejemplo anterior nos ha permitido presentar una acción, para que le perdamos el miedo. Pero hay montones de detalles que se nos han quedado fuera. Para presentarlos, precisemos antes cómo debemos entender el papel de las acciones dentro de WebSnap.

Primera premisa: el objetivo final de las instrucciones en JScript de las plantillas es generar elementos visuales en HTML, a partir de ciertos objetos que maneja el servidor, y que representamos mediante *adaptadores*. Seguramente estará harto de ver repetida la metáfora que presenta a un objeto como una caja negra; las propiedades serían paneles de visualización LCD o LED, y los métodos corresponderían a botones en la superficie de la caja. Idealmente, para modificar cualquier atributo del objeto deberíamos pulsar alguno de esos botones. Tenga *muuy* presente que estoy hablando ahora del objeto adaptador de Delphi, programado en Object Pascal. Bajo estas suposiciones:

- 1 Los campos son “propiedades” del objeto adaptador.
- 2 Las acciones corresponden a los “métodos”, en un sentido amplio, que permiten la modificación del estado del objeto, y en consecuencia, de sus propiedades.

Pero en WebSnap, esa caja negra se encuentra oculta dentro de una cámara acorazada de seguridad. Si el usuario necesita saber el valor que muestra un panel o quiere pulsar un botón tiene que hacerlo mediante una pantalla sensible al tacto:



Porque poniéndonos metafóricos, en la página HTML generada por WebSnap lo que se ve son las “sombras” del objeto real. Esa imagen de sombras se genera, con la ayuda del ya famoso JScript, en el propio servidor. En JScript, el objeto se ve representado por una variable. Podemos pedir el valor de una propiedad a través de ella, pero: ¿no tiene sentido alguno que ejecutemos métodos sobre la variable! Al menos, esa ejecución no va a poder modificar el estado del objeto, sino de su “sombra”. Por este motivo es que las acciones *no corresponden a métodos* en JScript, sino a otras propiedades, como ya he mencionado.

Volvamos al objeto adaptador, dentro de Delphi. He dicho antes que una acción simula un método que se ejecuta sobre ese objetos. Al generarse la “imagen” HTML, dentro de un formulario, los campos se representan mediante controles de edición, y las acciones mediante botones; al menos, en nuestro ejemplo. Como teníamos una sola acción, usamos un único botón, pero podríamos crear varias acciones y poner un botón por cada una de ellas. Por ejemplo, si un adaptador representa la cesta de la compra, podríamos tener acciones, y botones, para actualizar las cantidades de cada producto, para eliminar determinado producto o para facturar. Cuando el usuario pulsa uno de esos botones, se envía una petición al servidor, indicando cuál fue la acción elegida por el usuario.

Normalmente los métodos van acompañados de parámetros. En este modelo, podríamos interpretar como tales los valores que haya tecleado el usuario en los controles del formulario. Ya hemos visto que, al ejecutar el código asociado a la acción, podemos buscar el valor de esos controles en la propiedad *ActionValue* de cada campo. Pero podemos necesitar parámetros “fijos”, que el usuario no pueda modificar. Por ejemplo, cuando vamos a facturar el contenido de la cesta de la compra, tenemos que indicar a la acción cuál es el identificador del usuario conectado. La solución más inmediata, con los recursos que conocemos, sería utilizar “propiedades ocultas”, que se representen en HTML mediante campos ocultos.

No obstante, WebSnap nos permite utilizar una técnica adicional, definiendo parámetros específicos para la acción. Cuando se genera la página con el formulario, podemos especificar esos parámetros por medio del evento *OnGetParams* de la acción:

```
procedure TPrincipal.GrabarGetParams(Sender: TObject;
  Params: TStrings);
begin
  Params.Add('HORA=' + TimeToStr(Now));
end;
```

En este caso, estamos asociando a la acción la hora en que se ha generado el formulario. El resultado de la función *LinkToPage* sería similar al siguiente:

```
HORA.8.18:02:19__p.9.Principal__id.35.☞
Principal.ApplicationAdapter.Grabar
```

Podríamos recuperar ese parámetro dentro del evento *OnExecute*, para calcular, por ejemplo, el tiempo que ha tardado el usuario en responder:

```
S := (Frac(Now) - StrToTime(Params.Values['HORA'])) * 86400;
WriteLn(T,
  Mensaje.ActionValue.Values[0] +
  ' [Tiempo: ' + FormatFloat('0.000', S) + ' seg]');
```

## Acciones, botones y redirección

Todavía no hemos acabado con las acciones. Veamos ahora el método de JScript que nos ha ayudado en la generación del botón:

```
LinkToPage(url1, url2)
```

Hay dos parámetros en *LinkToPage*: el primero es el nombre de la página que debe mostrarse si la ejecución de la página transcurre sin problemas, y el segundo es el nombre de la página que hay que mostrar cuando se produce un error. En nuestro ejemplo hemos indicado que ambas páginas sean la misma que ha generado el formulario inicialmente:

```
LinkToPage(Page.Name, Page.Name)
```

La forma en que se envía la nueva página al cliente depende del valor de la propiedad *RedirectOptions* de la acción, que almacena un conjunto de dos opciones: *roRedirectHTTPPost* y *roRedirectHTTPGet*. Ya hemos discutido este problema antes, en el contexto de WebBroker: si una petición ejecuta cambios persistentes en el servidor e inmediatamente devuelve una página HTML al cliente, podemos tener problemas si el cliente decide releer la página, porque se volvería a ejecutar la acción. La solución consistía en sustituir la respuesta HTML por un comando de redirección a una nueva página. Ahora vemos que WebSnap incorpora este mismo truco en sus acciones.

Resumiendo: podemos implementar el ciclo de envío de formulario/tratamiento de datos/información de respuesta en WebSnap utilizando un máximo de tres páginas: la que genera el formulario, la que genera la respuesta para el usuario cuando todo va bien, y la que genera esa misma respuesta si hay problemas. No hace falta habilitar ninguna página o “acción” (en el sentido de WebBroker) adicional: WebSnap envía al servidor peticiones especiales que las acciones de un adaptador pueden manejar sin ayuda nuestra.

Claro, esas tres páginas se pueden reducir a una sola, como en el ejemplo que he mostrado. La página *Principal* es la que crea el formulario en el que el usuario teclea su mensaje. Cuando el usuario envía los datos, la grabación tiene lugar dentro del evento *OnExecute* de la acción que hemos diseñado con este objetivo. En cualquier caso, vaya bien o mal la grabación, la respuesta consiste en volver a mostrar la misma página. Si quiere, puede probar fuerzas diseñando un par de páginas adicionales para redirigir la respuesta de la acción a ellas.

Solamente quiero añadir un pequeño detalle al ejemplo. Tal como funciona hasta ahora, el cuadro de edición del mensaje siempre aparece vacío. Analicemos entonces la siguiente respuesta al evento *OnGetValue* del campo *Mensaje*:

```

procedure TPrincipal.MensajeGetValue(Sender: TObject;
    var Value: Variant);
var
    S: string;
begin
    if Mensaje.ActionValue = nil then
        Value := ''
    else
        begin
            S := Mensaje.ActionValue.Values[0];
            if (S <> '') and (S[Length(S)] <> '?') then S := S + '?';
            Value := S;
        end;
    end;
end;

```

Compruebe si su funcionamiento concuerda con la idea que se ha formado sobre los campos y acciones. Por ejemplo, ¿para qué comprobamos si *ActionValue* es **nil**? Recuerde que *ActionValue* contiene los valores que el usuario ha editado dentro de los controles de un formulario. El evento *OnGetValue* del campo se dispara al generar la página, y existen dos formas de activar esa generación: cuando tecleamos la URL de

la página en el explorador de Internet, por primera vez, o cuando pulsamos el botón de grabar y el servidor nos envía el formulario de vuelta. En el primer caso, *Action-Value* no tiene sentido alguno, y no se asocia a objeto alguno. Por este motivo es que devolvemos una cadena vacía, para que el control aparezca en blanco en la primera petición.

A partir de ese momento, el valor que haya tecleado el usuario se envía de vuelta... con un pequeño detalle: se le añade un signo de interrogación al final. Así, cuando el usuario grabe *Hola*, volverá a aparecer el formulario con la pregunta *Hola?* en su interior. Y ahora, la prueba de fuego: ¿qué pasaría si activásemos las dos opciones de redirección en la acción *Grabar*? No le voy a decir la respuesta, pero no siga leyendo hasta que no comprenda por qué sucede lo que sucede.

## Producción orientada a adaptadores

¿Recuerda el contenido del formulario que hemos utilizado en el ejemplo?

```
<form name="Grabacion" method="post">
  <input type="hidden" name="__act">
  <!-- ... más HTML ... -->
</form>
```

La pregunta lógica sería: ¿cómo he logrado averiguar que hace falta un campo oculto, llamado *\_\_act*, para que funcione todo el sistema de WebSnap? La mejor forma de familiarizarse con las exigencias del guión... de acuerdo, suena demasiado cinematográfico... con las peculiaridades del *scripting* en WebSnap es dejar que un amigo invisible nos haga el trabajo sucio, tal como hacen los niños pequeños. Nuestro amigo invisible, que no imaginario, se llama *TAdapterPageProducer*, y es un productor de contenido HTML alternativo a *TPageProducer*.

Más que limitarnos a mostrar su uso, voy a enseñarle cómo sustituir un *TPageProducer* por el nuevo *TAdapterPageProducer*. Inicie una nueva aplicación WebSnap, y duplique los pasos seguidos en el ejemplo anterior. Si lo desea, llame también *FormInput* a la clase COM que creará el asistente; como única diferencia inicial, nombre *FormInput01* al proyecto, para que el identificador de programa de la clase sea diferente. Resumo para usted el resto de los pasos:

- 1 Añada al *ApplicationAdapter* el campo *Title*, con el comando *Add all fields* del menú de contexto de este componente.
- 2 Cree un nuevo campo llamado *Mensaje*, dentro del adaptador de aplicación. No hace falta que intercepte su evento *OnGetValue*, para este nuevo ejemplo. Tampoco es necesario que cree un campo *Hora*.
- 3 Entre en la propiedad *Actions* del adaptador de aplicación y cree una nueva acción grabar.
- 4 Finalmente, intercepte el evento *OnExecute* de la nueva acción.

```
procedure TPrincipal.GrabarExecute(Sender: TObject;
```



```

    Params: TStrings);
var
  T: TextFile;
begin
  AssignFile(T, ChangeFileExt(ParamStr(0), '.txt'));
  try Append(T); except Rewrite(T); end;
  try
    WriteLn(T, Mensaje.ActionValue.Values[0]);
  finally
    CloseFile(T);
  end;
end;
end;

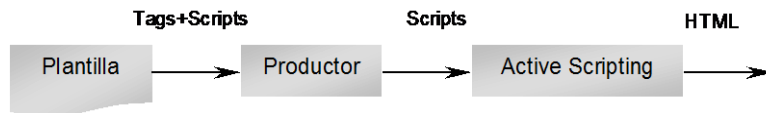
```

Al llegar a este punto, nos encontramos con una página que genera su contenido a partir de un *TPageProducer*. Realmente, podríamos haber indicado al asistente que generase la página con un *TAdapterPageProducer*. Pero vamos a suponer que nos hemos equivocado, y elegido el componente equivocado. Los pasos para sustituir el componente son:

- 1 Elimine, por supuesto, el componente *PageProducer*, y traiga un *TAdapterPageProducer*. No hace falta que le cambie el nombre.
- 2 Observe cómo el módulo Web se cambia inmediatamente al bando vencedor: inicialmente, su propiedad *PageProducer* apuntaba al componente eliminado, pero ahora pasa a apuntar a *AdapterPageProducer1*.
- 3 Por último, seleccione la plantilla *UPrincipal.html* en el Editor. Vaya al final de la página y añada las siguientes etiquetas, dentro del cuerpo de la plantilla:

```
<#STYLES><#WARNINGS><#SERVERSCRIPT>
```

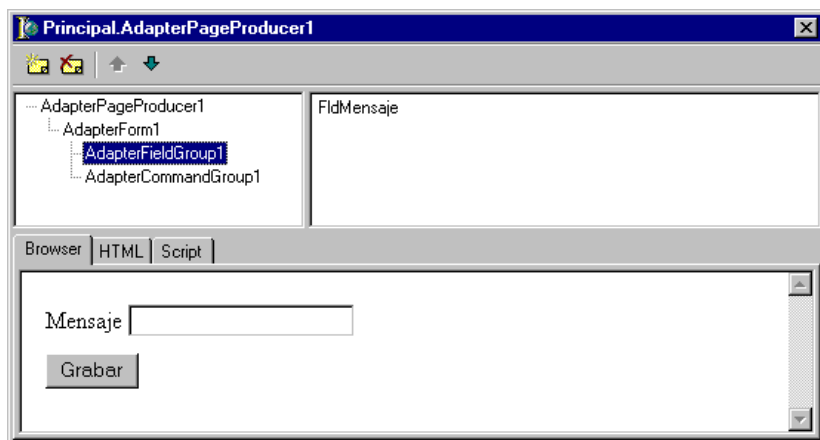
Puede que estas tres etiquetas le desconcierten inicialmente. ¿No habíamos quedado en que íbamos a utilizar JavaScript en el servidor en vez de etiquetas transparentes? Si ha pensado tal cosa, está equivocado: WebSnap utiliza JavaScript y etiquetas transparentes, pero con el siguiente orden de evaluación:



Es decir, que para generar una página HTML, primero se filtra la plantilla a través del productor que esté asociado al módulo que representa a la página; la responsabilidad del productor consiste en sustituir las etiquetas “transparentes” que pueda contener la plantilla original. Un *TPageProducer* utilizará el código que escribamos en respuesta a su evento *OnHTMLTag* para expandirlas, pero un *TAdapterPageProducer* sabe, además, cómo sustituir *SERVERSCRIPT* y sus compañeras por código JavaScript. ¡Note que es posible que ya exista código JavaScript en la plantilla! Finalmente, es el módulo quien invoca al motor de Active Scripting para evaluar las instrucciones JavaScript o VBScript resultantes.

Volvamos la atención al *TAdapterPageProducer*. Existe una gran semejanza entre este componente y *TInetXPageProducer*: ambos generan una mezcla de código *script* con HTML, y en los dos casos, el origen es un árbol de componentes que se almacena dentro de la aplicación. La diferencia: el código *script* generado por Internet Express se evalúa en el lado cliente, mientras que el de un *TAdapterPageProducer* es evaluado dentro de la misma aplicación, en el lado servidor. Y el conjunto de componentes generadores es diferente en cada caso, como veremos en breve.

Haga doble clic sobre el componente productor. Vamos a añadir componentes hasta que el editor tenga el siguiente aspecto:



Los pasos necesarios para llegar a ese resultado son:

- 1 Inicialmente, sólo tendrá el nodo principal, llamado *AdapterPageProducer1*.
- 2 Pulse la tecla INS y añada un *AdapterForm*. La alternativa, el *LayoutGroup*, sirve para distribuir el contenido varias columnas.
- 3 Dentro del *AdapterForm* añada un *AdapterFieldGroup* y un *AdapterCommandGroup*. Dentro del primero colocaremos controles que corresponderán a campos del adaptador, y dentro del segundo se utilizarán botones o enlaces en correspondencia con las acciones del adaptador.
- 4 Observe que aparecerán varias advertencias, que vamos a resolver. Primero, seleccione el *AdapterFieldGroup*, vaya al Inspector de Objetos y modifique la propiedad *Adapter* de este componente, para que apunte al adaptador de aplicación del módulo principal. Luego seleccione el grupo de comandos, para modificar su propiedad *DisplayComponent*, que deberá apuntar al grupo de campos anterior.
- 5 Por omisión, los grupos de acciones y campos crearán automáticamente sus propios subcomponentes. Por ejemplo, en el grupo de campos aparecerán controles para los dos campos existentes en el adaptador: el título y el mensaje. Además de que el título es innecesario, no es bueno dejar los controles por omisión, por algunos *bugs* pendientes. Por lo tanto, seleccione el *AdapterFieldGroup* y

utilice el comando *Add fields* para añadir el campo *Mensaje* al grupo. Finalmente, añada en el grupo de comandos la acción *Grabar*.

Cuando haya configurado todos los componentes, podrá seleccionar la página *Script*, en el panel inferior, para ver el código que genera el productor; en realidad, el texto con el que sustituye la etiqueta `<#SERVERSCRIPT>`. Y si selecciona la página *HTML*, verá el resultado final, incluyendo la evaluación del *script*. Por supuesto, es en *Script* donde he encontrado el fragmento que aproveché en el anterior ejemplo:

```
<form name="AdapterForm1" method="post">
  <input type="hidden" name="__act">
  <table>
  ...
```

Encontraremos además otras instrucciones que no generan código alguno en este ejemplo. Como las que incluyo a continuación:

```
<% if (vApplicationAdapter.HiddenFields != null)
{
  vApplicationAdapter.HiddenFields.WriteFields(Response)
} %>
<% if (vApplicationAdapter.HiddenRecordFields != null)
{
  vApplicationAdapter.HiddenRecordFields.WriteFields(Response)
} %>
```

Porque debe saber que no siempre el componente generará el código más compacto; en este contexto, compacto y eficiente casi siempre quieren decir lo mismo.

## Listas de errores

Aprovechando que tenemos el *AdapterPageProducer* conectado, vamos a añadir un último componente de generación a la sopa. Seleccione el *AdapterForm1* en el árbol de componentes, ejecute el comando *New component* y añada un *AdapterErrorList*. No va a estremecerse el suelo, ni aparecerán trompetas en el cielo, pero comprobará que en la página *Script* aparecerán las siguientes líneas de JScript, al final del código generado:

```
<% {
  var e = new Enumerator(
    Modules.Principal.ApplicationAdapter.Errors)
  for (; !e.atEnd(); e.moveNext())
  {
    Response.Write("<li>" + e.item().Message)
  }
  e.moveFirst()
} %>
```

Note, en primer lugar, que el adaptador ya contaba con una propiedad *Errors*, disponible desde el lenguaje de *script*, que corresponde a una propiedad con el mismo

nombre de la clase *TCustomAdapter*. Para poder utilizarla en nuestro ejemplo, modificaremos la respuesta al evento *OnExecute* de la acción *Grabar* en la siguiente forma:

```

procedure TPrincipal.GrabarExecute(Sender: TObject;
  Params: TStrings);
var
  S: string;
  T: TextFile;
begin
  try
    S := VarToStr(Mensaje.ActionValue.Values[0]);
    if S = '' then
      raise Exception.Create('Mensaje vacío');
    AssignFile(T, ChangeFileExt(ParamStr(0), '.txt'));
    try Append(T); except Rewrite(T); end;
    try
      WriteLn(T, S);
    finally
      CloseFile(T);
    end;
  except
    on E: Exception do
      ApplicationAdapter.Errors.AddError(E);
    end;
  end;

```

He encerrado el núcleo de la respuesta dentro de un gran bloque **try/except**. Si se produjese alguna excepción durante su ejecución, ya sea prevista o imprevista, en la cláusula **except** estamos impidiendo la propagación de la misma. Una aplicación WebSnap no es una aplicación interactiva, en la que podríamos dejar flotar las excepciones hasta el bucle de mensajes. Como efecto secundario, pero importante, el mensaje asociado a la excepción se almacena en la lista de errores del adaptador.

El evento anterior se ha disparado durante la respuesta a una petición asociada a una acción. ¿Recuerda que al generar el botón que dispara la acción hemos utilizado una llamada a *LinkToPage*?

```

<% if (vApplicationAdapter_Grabar.Visible)
{ %>
<td><input type="submit" value="Grabar"
onclick="AdapterForm1.__act.value=␣
'<%=vApplicationAdapter_Grabar.LinkToPage(␣
Page.Name, Page.Name).AsFieldValue%>'></td>
<% } %>

```

Si utilizamos un *TAdapterPageProducer*, los parámetros de *LinkToPage* se extraen de las propiedades *PageName* y *ErrorPageName* del componente *TAdapterActionButton* que se trae en representación de la acción. En este ejemplo, hemos dejado vacías ambas propiedades, por lo que se utiliza la propia página, tanto para el caso en que la acción se ejecute sin problemas como para el caso en que se produzca un error. Esto último es muy conveniente: si se produce un error, mostramos al usuario la lista de errores que hemos detectado... pero dentro de la misma página en la que debe corregirlos.

¿Qué pasaría si la acción tuviese activa las opciones de redirección? Si la página de errores se mostrase mediante una petición de redirección, correríamos el riesgo de perder la lista de errores: la lista de errores se llenaría durante la petición original, pero si hay redirección, la página con la respuesta final se generaría en una petición posterior e independiente, en la que la lista de errores estaría vacía.

No obstante, no hay por qué preocuparse: las opciones de redirección de las acciones solamente se aplican cuando la ejecución transcurre sin problemas. Si desea probarlo, active las opciones de redirección de la acción *Grabar*, ejecute la aplicación e intente grabar un mensaje en blanco. Verá que el error se muestra correctamente. Y si revisa el registro del Web App Debugger, comprobará que solamente se lanza la redirección cuando el mensaje se graba sin problemas.

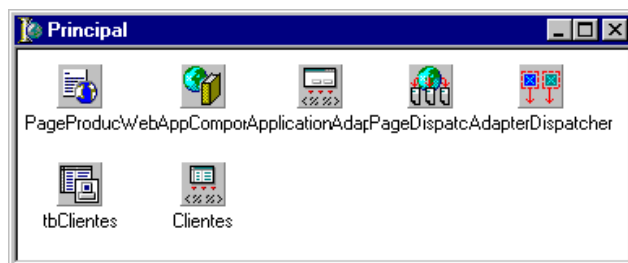


## WebSnap: conjuntos de datos

ESTE ES EL PUNTO EN EL QUE COMIENZAN casi todas las explicaciones de WebSnap: usted añade controles a una página Web, como si estuviese creando un formulario. Conecta estos controles a un misterioso componente adaptador, sobre el que es mejor no preguntar demasiado, y mágicamente todo parece funcionar. Nadie le explicará, por supuesto, que la técnica que divide un listado largo en páginas es lo más ineficiente que pueda imaginar, ni que la modificación de un registro trae aparejada muchas otras operaciones innecesarias sobre la base de datos. Eso deberá descubrirlo usted en la práctica, y esta vez sí que se trata de una cara oculta... de acuerdo, no de Delphi, pero sí de WebSnap.

### Adaptadores para conjuntos de datos

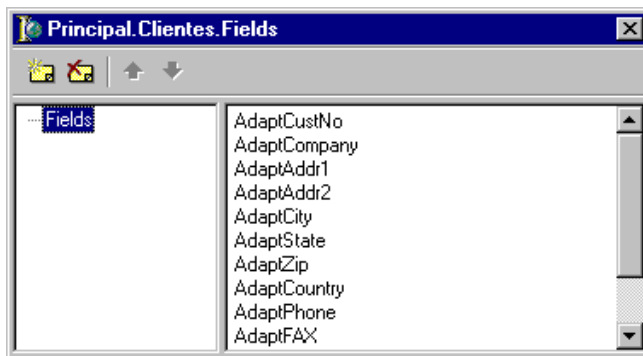
El principal responsable de lidiar con los conjuntos de datos es un componente que se llama *TDataSetAdapter*, y se encuentra, como era de esperar, en la página *WebSnap* de la Paleta de Componentes. Para probar su funcionamiento, vamos a crear una nueva aplicación WebSnap que se pueda ejecutar dentro del Web App Debugger. Para los ejemplos del CD, he llamado *Clientes* al nuevo proyecto, porque vamos a utilizar una tabla de empresas. La clase ha recibido el mismo nombre, de modo que el identificador de programas es *Clientes.Clientes*. Para la página he reservado el exclusivo nombre de *Principal*, y he pedido un módulo de página para que WebSnap coloque sus componentes de aplicación. Da lo mismo el componente productor de contenido que cree ahora.



Sí necesitaremos un *TClientDataSet*, al que llamaremos *tbClientes*. En el directorio de la aplicación copiaremos el fichero *customer.xml*, de los ejemplos de MyBase, y haremos

que la propiedad *FileName* de *tbClientes* haga referencia al mismo. Añada al conjunto de datos todos los componentes de acceso a campos y cambie, al menos, el *Display-Label* de cada uno de ellos. Finalmente, abra paso al protagonista: traiga un *TDataSetAdapter* al módulo, llámelo *Clientes* y modifique su propiedad *DataSet* para que apunte a la tabla de clientes.

El siguiente paso es añadir al adaptador su propio conjunto de *campos de adaptador*. Podríamos confiar en los campos de adaptador que el componente crea implícitamente, pero perderíamos la posibilidad de configurarlos a nuestro antojo. Debemos hacer doble clic sobre el adaptador, o editar su propiedad *Fields*, y ejecutar el comando *Add fields* del menú de contexto del editor de la colección:



Por el momento, sólo vamos a configurar una propiedad de un único campo: seleccione el campo *AdaptCustNo* y añada la opción *ffInKey* en su propiedad *FieldFlags*. No es que sea estrictamente necesario hacerlo ahora, pero no quiero que se nos olvide.

El único adaptador que hemos visto hasta el momento ha sido el *ApplicationAdapter* de los ejemplos del capítulo anterior, que en realidad, es un caso particular de la clase más general *TAdapter*. El nuevo *TDataSetAdapter*, sin embargo, se diferencia de estos adaptadores más simples en que representa potencialmente a más de un registro. En consecuencia, debe permitir el uso de enumeradores en JScript. Como ejemplo, el siguiente fragmento de código sirve para recorrer todos los registros de clientes dentro de una plantilla (¡y no hacer nada, por el momento!):

```
<%
var e = new Enumerator (Clientes.Records);
while (! e.atEnd())
    e.moveNext();
%>
```

O utilizando una variante más “compacta”:

```
<%
for (var e = new Enumerator (Clientes.Records);
    ! e.atEnd(); e.moveNext()) ;
%>
```



**ADVERTENCIA**

No destruimos la instancia del enumerador al terminar porque JScript implementa la recogida automática de basura. Tenga mucho cuidado de no ejecutar código JScript estando su jefe cerca, que puede perderlo para siempre. Asegúrese, en cualquier caso, de que deja pagada las nóminas.

En el lenguaje de *script*, los campos se consideran como si fueran propiedades de los adaptadores. Añada este pequeño fragmento dentro de la plantilla de la página, para generar una lista sencilla con los nombres de clientes. No hará falta que ejecute la aplicación; bastará con que seleccione la vista *Preview* en el editor de Delphi.

```
<%
    var e = new Enumerator(Clientes.Records);
    while (! e.atEnd()) {
        Response.Write(Clientes.Company.DisplayText + '<br>');
        e.moveNext();
    }
%>
```

¡Observe que he utilizado la propiedad *DisplayText* de *Company*! Las otras propiedades similares soportadas en JScript son *EditText*, para el texto en edición, y *Value*, que representa el valor “real” del campo. Deberíamos siempre utilizar *DisplayText* para mostrar valores que no estén dentro de controles HTML de edición. En cambio, dentro de los controles lo apropiado es usar *EditText*. Finalmente, para cálculos internos, la propiedad adecuada es *Value*.

Con el nombre de la compañía no hay diferencias entre las propiedades mencionadas, pero pongamos por caso que queremos mostrar, además del nombre de la compañía, la tasa de impuestos aplicable, *TaxRate*:

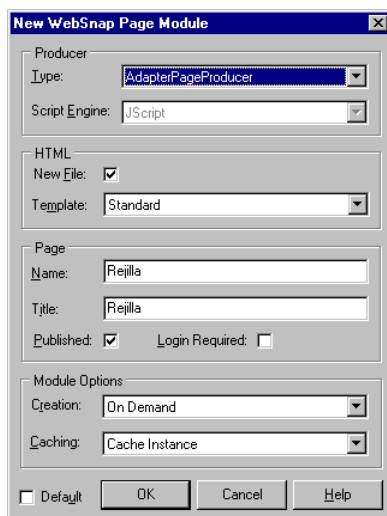
```
Response.Write(Clientes.Company.DisplayText +
    ': ' + Clientes.TaxRate.DisplayText + '<br>');
```

Juegue un poco con la última línea, sustituya *DisplayText* por *EditText* primero, y luego por *Value*, y compare los resultados. Observará algo un poco extraño, a primera vista: si modifica el formato de visualización, *DisplayFormat*, para el campo de los impuestos, *tbClientesTaxRate*, el formato se propagará a la propiedad *DisplayText* del campo del adaptador. En cambio, da lo mismo lo que hagamos con *EditFormat*: el adaptador no se fiará de nuestras buenas intenciones, y mostrará el contenido de *EditText* formateado a su gusto. Asigne la cadena *0.00* dentro de la propiedad *EditFormat* de *tbClientesTaxRate*, para que aparezcan dos decimales en *EditText*... y comprobará que JScript se empeña en no añadir los decimales en el resultado.

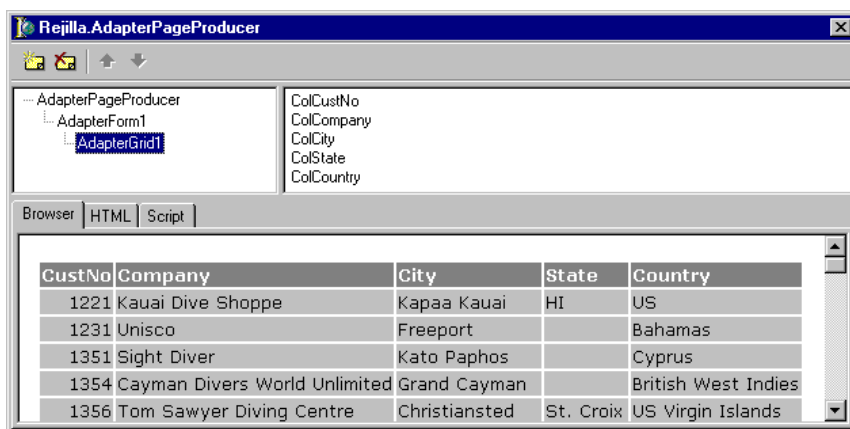
Es más, aunque el componente para el campo de adaptador dentro de Delphi, llamado *AdaptTaxRate* en nuestro ejemplo, soporta los eventos *OnGetDisplayText* y *OnGetValue* para que personalizemos la visualización o incluso el valor que se aparecerán en JScript, no hay un evento equivalente para modificar el formato del texto en edición.

## Rejillas en HTML

Complicuemos las cosas añadiendo una nueva página a nuestra aplicación de prueba. Para ello, debemos pulsar sobre el botón *New WebSnap Page Module*, en la barra de herramientas de WebSnap; si está visible, claro. En caso contrario, podemos ir a la página *WebSnap* del Almacén de Objetos, y pulsar sobre el icono correspondiente. Este es el asistente para la creación de nuevas páginas:



He llamado *Rejilla* a la nueva página, porque vamos a mostrar el contenido de la tabla de clientes sobre el equivalente de una rejilla de datos de Delphi; bueno, en realidad sobre una tabla HTML. Es importante que especifiquemos un *TAdapterPageProducer* como componente generador de contenido.



Haga doble clic sobre *AdapterPageProducer1*, y añada el consabido componente inicial *AdapterForm* al nodo raíz. Para variar, traiga entonces un *AdapterGrid* dentro del for-

mulario. Aparecerá una advertencia: hay que darle un valor a la propiedad *Adapter* de la rejilla. El adaptador *Clientes* se encuentra dentro del módulo *Clientes*, por lo que hay que añadir la unidad *UPrincipal* a la cláusula **uses** de la unidad *URejilla* antes de complacer a WebSnap. Cuando lo haya hecho, verá que todos los campos de la tabla de clientes aparecerán de súbito en el panel de la vista previa. ¿Motivo? Ya lo conoce: el componente *AdapterGrid1* creará implícitamente una columna para cada campo, si no lo impedimos. Por lo tanto, debemos seleccionar *AdapterGrid1* y ejecutar su menú de contexto *Add columns*, para seleccionar unas pocas columnas.

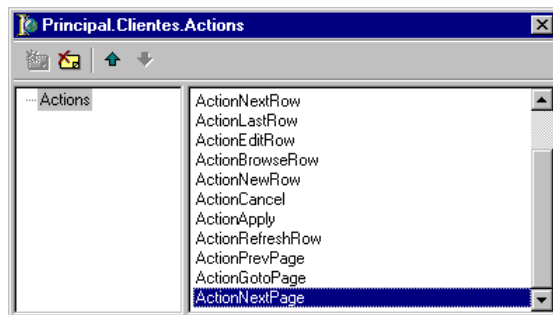
### UN PAR DE CONSEJOS

Hay pocas cosas más feas que una tabla HTML con los bordes por omisión visibles. Si quiere que su página tenga una apariencia soportable, lo primero que debe hacer es seleccionar la propiedad *TableAttributes* del *AdapterGrid1*, y asignar un enorme cero en la subpropiedad *Border*. A partir de ese momento, tendrá que volverse un mago en el uso de estilos para que la tabla mejore un poco visualmente. Que tenga suerte, y no pierda la paciencia...

## Paginación

Le advierto que encontrará una dosis de muy mala leche en las conclusiones de esta sección. Voy a presentarle el mecanismo que ofrece WebSnap para dividir un conjunto de datos grandes en trozos más manejables. La tabla de clientes es muy pequeña, pero nos servirá de ilustración. Seleccione el adaptador *Clientes*, en la página principal de la aplicación. Solamente tendrá que modificar su propiedad *PageSize*; asígnele 10, para mostrar solamente grupos de 10 registros en la rejilla.

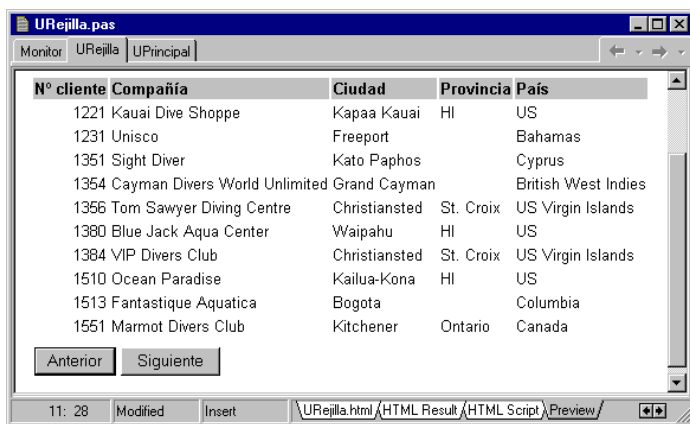
Si ahora ejecuta la aplicación, comprobará que la rejilla se detiene en el décimo registro; el listado sencillo de la página de inicio hace lo mismo, aunque no hayamos previsto medidas especiales. Claro, el siguiente paso debe consistir en permitir que el usuario navega, al menos, a la página siguiente y a la anterior.



La navegación se logra con acciones predefinidas para que acompañen al adaptador de conjuntos de datos. No es obligatorio, pero sí prudente que editemos la propiedad *Actions* del adaptador *Clientes* para añadir todas las acciones predefinidas mediante el comando de menú *Add all actions*. Son muchas las acciones, como verá, pero ahora solamente nos ocuparemos de las tres últimas: *PrevPage*, para la página anterior,

*NextPage*, para la siguiente, y *GotoPage* para ir a una página concreta. Las dos primeras que he mencionado tienen una propiedad *DisplayLabel*, en la que podemos traducir el título de cada una de ellas.

Para incluir estas acciones en la página, debemos editar el *AdapterPageProducer1*, de la página *Rejilla*, y añadir un *AdapterCommandGroup* dentro de *AdapterForm1*. Inicialmente aparecerán representadas todas las acciones del adaptador, pero ejecutando el comando *Add commands* añadimos solamente aquellas acciones que nos interesan. La siguiente imagen muestra la vista preliminar de la rejilla, una vez que se añaden los comandos *PrevPage* y *NextPage*, cambiando además la propiedad *DisplayType* de los dos comandos a *ctButton*:



Para comprender cómo funciona el sistema de paginación de WebSnap, echemos un vistazo al código HTML generado. Esta es la etiqueta del botón *Anterior*, cuando nos hemos desplazado al segundo grupo de registros:

```
<input type="submit" value="Anterior"
onclick="AdapterForm1.__act.value='_cp.1.1_1CustNo.4.1560'
__p.7.Rejilla_id.27.Principal.Clientes.PrevPage">
```

Observe que la acción *PrevPage* ha sido definida por WebSnap con un *parámetro* (¿recuerda *OnGetParams*?) llamado como la clave primaria de la tabla asociada. En este caso, el valor asignado al campo oculto indica que el grupo generado tiene como primer registro a aquel cuya clave primaria vale 1.560. Con sólo esta información, WebSnap puede seleccionar el grupo de registros anterior y el siguiente. Con total transparencia para nosotros.

Entonces, ¿de qué me quejo? Pues de que no todo es de color rosa, y que nadie nos advierte acerca de ello. Como soy muy desconfiado, seleccioné el conjunto de datos *tbClientes*, y creé un manejador para su evento *AfterScroll*:

```

procedure TPrincipal.tbClientesAfterScroll(DataSet: TDataSet);
begin
    OutputDebugString(PChar('Registro #' +
        IntToStr(tbClientes.RecNo)));
end;

```

El procedimiento *OutputDebugString*, del API de Windows, envía la cadena que le pasamos como parámetro a una estructura interna de Windows, que podemos examinar desde Delphi mediante la ventana que aparece al ejecutar el comando de menú *View|Debug Windows|Event list*. Haga usted mismo el experimento, y descubrirá algo asombroso: cada vez que pedimos un grupo de registros, ¡WebSnap empieza a contarlos desde el primero! Esto significa que, a la medida que vamos avanzando hacia el final de una tabla grande, tardaremos cada vez más en generar la página siguiente.

Hay una justificación para ello: a no ser que usemos consultas especialmente preparadas para ello, sabemos que no existe una forma eficiente para ir directamente a un registro del que conocemos su posición. No me molesta entonces que esta primera implementación sea tan mala. Pero sí me cabrea que nadie advierta sobre este hecho, a pesar de toda la tinta que ha corrido sobre WebSnap desde que apareció Delphi 6.

#### UN AMARGO COMENTARIO

Imagine que está en uno de los bordes del Gran Cañón del Colorado, y que debe cruzar urgentemente al otro lado. ¿Sabe qué es mucho peor que la ausencia de un puente? Pues que venga un cantamañanas, tire una cuerda de un lado a otro del barranco y le diga que tenga fe y que camine sobre ella...

## Búsqueda y presentación

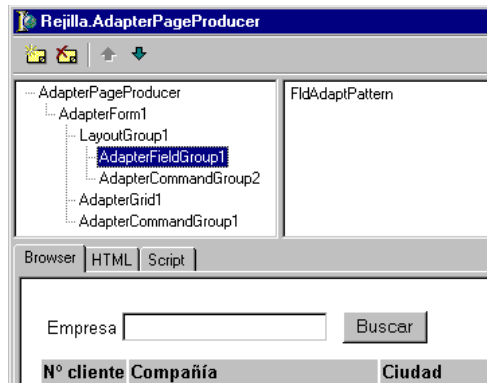
Otra de mis quejas acerca de WebSnap: nadie escribe una aplicación para Internet en la que se muestren tropecientos cincuenta mil registros para que el usuario elija uno y lo edite. Lo normal es que el usuario comience definiendo un criterio de búsqueda que acote el número de registros disponibles, y luego haga con el resultado lo que tenga que hacer. Sin embargo, esta tarea tan básica no se ha incluido en ninguno de los ejemplos de Delphi 6; tampoco en la documentación escrita. Personalmente, lo pasé muy mal hasta que descubrí cómo hacerlo... con la menor cantidad de código posible, que es algo muy importante, porque demuestra si realmente se dominan los conceptos sobre los que se basa WebSnap.

Como estamos utilizando una tabla MyBase, no podemos utilizar una consulta para demostrar esta técnica de búsqueda y presentación. Pero sí podemos recurrir a los filtros de MyBase que, como sabemos, son muy flexibles. Permitiremos que el usuario teclee una cadena de búsqueda en un formulario que ubicaremos dentro de la misma página de la rejilla. Cuando pulse el botón de búsqueda, en la rejilla sólo aparecerán aquellas empresas cuyos nombres comiencen con los caracteres tecleados por el usuario.

¿He dicho “botón”? Si ha leído el capítulo anterior, sabrá entonces que necesitamos una nueva acción. ¿He mencionado un control de edición HTML? Eso tiene pinta de campo de adaptador. Entramos en la propiedad *Fields* del adaptador *Clientes*, y añadimos un nuevo campo de tipo *TAdapterField*. Cambiamos su *DisplayLabel* para que diga *Empresas*, y lo nombramos *AdaptPattern*, para que su nombre no desentone demasiado. Luego entramos en *Actions*, y creamos un nuevo *TAdapterAction*; la bautizamos *ActionFilter* y cambiamos su *DisplayLabel* a *Buscar*.

Muy importante: deje sin activar las dos opciones de redirección de la acción. Al ejecutar la acción de filtrado en el servidor queremos que, acto seguido, se genere la página de la rejilla. Si hubiera una redirección por medio, sería demasiado complicado propagar la condición del filtro de una petición a la otra.

A continuación, vamos a añadir los nuevos elementos a la página. Añada un *Layout-Group* dentro del formulario ya existente, y muévelo a la primera posición, antes de la rejilla; luego, asigne un 2 en su propiedad *DisplayColumns*.



Dentro del mismo, añada un grupo de campos y otro de comandos. En el grupo de campos, añada explícitamente el campo *AdaptPattern*, y en el de comandos, el nuevo *ActionFilter*. Ya estamos listos para interceptar el evento *OnExecute* de la acción:

```

procedure TPrincipal.ActionFilterExecute(Sender: TObject;
  Params: TStrings);
var
  S: string;
begin
  S := AnsiUpperCase(VarToStr(
    AdaptPattern.ActionValue.Values[0]));
  if S = '' then
    tbClientes.Filtered := False
  else
    begin
      tbClientes.Filter := 'UPPER(COMPANY) LIKE ' +
        QuotedStr(S + '%');
      tbClientes.Filtered := True;
    end;
end;

```

Recuerde que las acciones se ejecutan antes de que se genere el contenido de la página. Examinamos el valor que ha tecleado el usuario en el campo del patrón. Si no ha tecleado nada, eliminamos cualquier posible filtro que esté activo en *tbClientes*. En caso contrario, le añadimos el comodín y creamos una expresión basada en el operador **like** de SQL. No olvide que sólo es posible utilizar **like** en un filtro cuando se trata de un conjunto de datos cliente.

Es el momento de ocuparnos del evento *OnGetValue* de *AdaptPattern*:

```
procedure TPrincipal.AdaptPatternGetValue(Sender: TObject;
    var Value: Variant);
begin
    if AdaptPattern.ActionValue <> nil then
        Value := VarToStr(AdaptPattern.ActionValue.Values[0]);
end;
```

La primera vez que se genera la página, la propiedad *ActionValue* del campo contiene un puntero vacío; sólo contendrá un objeto cuando la petición llegue del formulario de la propia página. En el segundo caso, el valor que se mostrará en el campo de búsqueda es el mismo que ha tecleado el usuario para generar la nueva página. De este modo, se propaga la cadena de búsqueda de una petición a la siguiente.

## Modos y edición

Cree una nueva aplicación WebSnap, y llámela *EdicionClientes*; use el mismo nombre, si no le importa, para el nombre de la clase. El generador de contenido de la página principal debe ser un *TAdapterPageProducer*. Y traiga un *TClientDataSet*, como en los ejemplos anteriores, llámelo *Clientes* y conéctelo a una copia del fichero *customer.xml*. Porque ahora vamos a mostrar cómo editar su contenido.

Las actualizaciones sobre conjuntos de datos en WebSnap se basan en el uso combinado de acciones, que ya conocemos, y una variable del estado interno del adaptador, conocida como *modo del adaptador*, y representada en la propiedad *Mode*. Hay cuatro modos: *amBrowse*, *amEdit*, *amInsert* y *amQuery*. A pesar de la creencia popular, el modo no determina el contenido de las propiedades de los campos del adaptador. Pero su valor puede ser consultado para generar un tipo u otro de *script*.

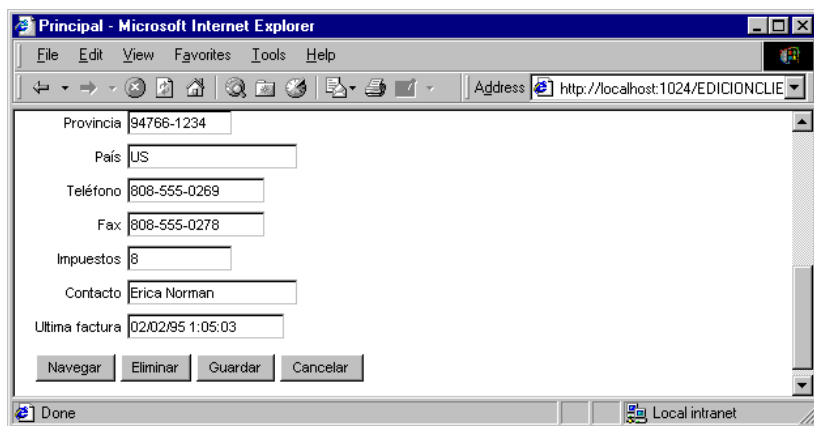
Para demostrarlo, traiga un *TDataSetAdapter* a la nueva aplicación y enlázelo a la tabla de clientes. Cree explícitamente todos los campos y acciones predefinidos del adaptador; no olvide seleccionar *AdaptCustNo* para indicar que se trata de la clave primaria, en su propiedad *FieldFlags*. Haga doble clic sobre *AdapterPageProducer1*, y añada los siguientes componentes:

- 1 Un *AdapterForm*, para que contenga a los restantes componentes.
- 2 Un primer *AdapterCommandGroup* en el que solo añadiremos cuatro comandos de navegación: los correspondientes a las acciones *FirstRow*, *PrevRow*, *NextRow* y

*LastRow*. No quiero poner todos los comandos dentro de un mismo grupo, para no abrumar al usuario.

- 3 Un *AdapterFieldGroup*, para añadir todos los campos del adaptador en su interior.
- 4 Un segundo *AdapterCommandGroup*, en el que añadiremos los comandos restantes del adaptador.

El resultado será similar al de la siguiente imagen, sobre todo si modifica un poco los estilos y las etiquetas de campos y acciones:



Además, comprobará que no hay que añadir una línea adicional de código para que podamos actualizar la tabla base. Quiero añadir una sola mejora, en relación con las acciones del adaptador *Cientes*. Seleccione, por ejemplo, la acción *ActionEditRow* y observe el valor de su propiedad *ActionModes*. Deje solamente los modos *amBrowse* y *amQuery*. Al hacerlo, evitamos que el botón de edición aparezca en la página cuando el adaptador ya se encuentre en alguno de los modos de edición. Repita este paso con las restantes acciones, hasta que esté satisfecho con el resultado.

Y bien, Mr. Grumpy, ¿tiene alguna queja? Pues sí: este sistema es estupendo... para MyBase, Access o Paradox, no para bases de datos SQL. Imagine que estamos editando una consulta SQL; y digamos que es una consulta con un único registro, para no complicar más las cosas. Esta es una descripción detallada de los acontecimientos:

- 1 Hay una primera petición para generar el formulario HTML con los datos iniciales del registro. Esto es inevitable, sea cual sea el sistema utilizado.
- 2 En el lado cliente, el usuario modifica los valores y pulsa el botón de grabar. Se genera entonces una segunda petición HTTP.
- 3 Esto es lo que pasa con el sistema de WebSnap: primero el adaptador reabre la consulta. Si la consulta es actualizable, esto puede implicar la lectura de metadatos. Primera lectura innecesaria.
- 4 Luego hay que localizar y leer los datos del registro a modificar. Incluso en nuestro ejemplo, con un solo registro, esta es una segunda lectura innecesaria.



- 5 Los cambios se *aplican* sobre la fila activa de la consulta. Se genera automáticamente, o con nuestra ayuda, la correspondiente instrucción **update**. Este paso es inevitable.
- 6 Pero es muy probable que la interfaz de acceso a datos con la que trabajamos haga algún tipo de relectura del registro, si existen *triggers* sobre la tabla base, valores por omisión o campos autoincrementales. Probablemente, una tercera lectura innecesaria.

Dos o tres lecturas adicionales gratuitas. Pero, ¿respecto a qué alternativa? El método que se suele utilizar en las aplicaciones *profesionales* (pongo las cursivas con saña) comparte los dos primeros pasos con el algoritmo anterior. A partir de ese momento:

- 3 Los parámetros de la petición se utilizan para llamar a un procedimiento almacenado que se encarga de la actualización. Eso es todo.

### INTERNET EXPRESS

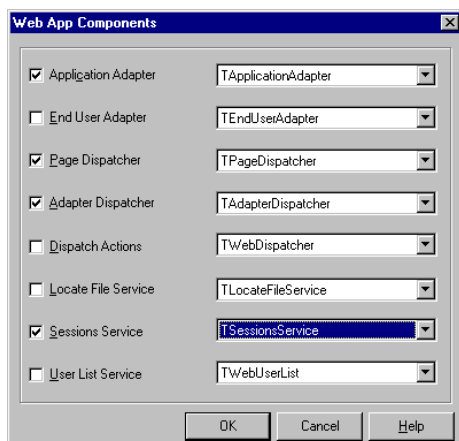
En ese sentido, Internet Express deja más oportunidades para modificar el algoritmo de grabación. El productor de páginas *TInetXPageProducer* puede utilizarse en una aplicación WebSnap. Pero, como dije en el capítulo correspondiente, Internet Express depende excesivamente de las capacidades del navegador, en el lado cliente.

## Sesiones

Sabemos ya que lo más complicado para una aplicación que utiliza HTTP es mantener la continuidad entre peticiones independientes entre sí. ¿Qué mecanismo propone WebSnap para lograrlo? Nada más y nada menos que las denostadas *cookies*. Al igual que hace ASP, WebSnap monta sobre las *cookies* un sistema de variables globales agrupadas por sesión. Ahora veremos qué hace falta para poder utilizar sesiones en WebSnap.

Con el asistente apropiado, cree una nueva aplicación WebSnap. Esta vez es sumamente importante que el módulo creado sea una DLL, ya sea ISAPI, NSAPI o colmulgue con el modelo Apache. O, en caso contrario, una aplicación para el Web App Debugger. En ningún caso puede ser una aplicación CGI; luego veremos el porqué de esta restricción. Le sugiero que elija el modelo del Web App Debugger, que llame *Tienda* al proyecto, y que utilice el mismo nombre para la clase COM

En el asistente de WebSnap debe pedir que los componentes globales de aplicación se sitúen sobre un módulo de datos, más bien que dentro de un módulo de páginas. Así podremos separar nítidamente el acceso a datos de la generación HTML. Y esto es imprescindible: pulse el botón *Components*, para marcar la casilla *Session services*. De esta manera, Delphi añadirá un componente *TSessionsService* al nuevo módulo, para que se encargue del mantenimiento de las sesiones.



Cuando la aplicación contiene un *TSessionsService*, para cada petición que recibe se analizan las *cookies* enviadas desde el cliente, en busca de un identificador de sesión. Lo normal es que la petición inicial enviada por un usuario no contenga tal identificador. WebSnap se encarga entonces de generar uno. En todo caso, a la petición se le asocia un *objeto de sesión*, perteneciente a algún descendiente de la clase *TAbstractWebSession*. La sesión asociada a la petición activa puede localizarse así:

```
WebContext.Session
```

La propiedad *SessionID* de este objeto contiene el identificador de sesión, representado como una cadena de caracteres. Al terminar de generar la respuesta a una petición, WebSnap llama al siguiente método del objeto de sesión, para asociar a la respuesta la *cookie* de retorno con el identificador de sesión, ya sea nuevo o recibido de una petición anterior:

```
procedure UpdateResponse (AResponse: TWebResponse);
```

En la presente versión de Delphi, los identificadores de sesión están formados por 16 caracteres alfanuméricos, con distinción de mayúsculas y minúsculas. Cada carácter se elige aleatoriamente de un mapa de caracteres ordenado también de forma aleatoria. No hay ningún tipo de dígito o letra de control; la seguridad se basa en la poca probabilidad de adivinar un identificador existente, dado el enorme dominio de valores posibles.

### IMPORTANTE

¡No debe confundir el servicio de sesiones con la identificación de usuarios! Las sesiones “ensartan” varias peticiones como las perlas de un collar, y nos ayudan a identificar aquellas que pertenecen a un mismo usuario... pero no nos dicen quién es éste. Aprovechando el servicio de sesiones, se puede añadir la posibilidad de que el usuario se identifique, quizás para reclamar determinados privilegios de acceso a algunas operaciones o páginas.

## Variables de sesión

Por una parte, el servicio de sesiones permite identificar y agrupar varias peticiones dentro de una misma sesión. Pero la principal aplicación de este recurso es la gestión de variables de sesión. Cada sesión mantendrá un diccionario, que a un nombre de variable asociará un valor de tipo *Variant*. Por ejemplo: tarta/chocolate, CPU/AMD, ineficiente/Java, etc. Estos diccionarios deben ser independientes entre sí.

Para acceder a las variables de una sesión debemos utilizar la propiedad vectorial *Values* del objeto de sesión asociado. Por ejemplo, en algún momento de la respuesta a una petición podríamos realizar una asignación similar a esta:

```
WebContext.Session.Values['Apellido'] := 'Marteens';
```

En cualquier petición posterior que pertenezca a la misma sesión, podemos preguntar por el valor de la “variable” *Apellido*; como es de esperar, recibiremos el valor que asignamos inicialmente.

WebSnap almacena la información sobre las sesiones existentes en una variable global de colección llamada *Sessions*. Para cada sesión mantiene, también en memoria, una colección de pares nombre/valor. Y aquí empiezan los problemas: para que el servicio de sesiones funcione, es necesario que la aplicación no se descargue de la memoria del servidor después de cada petición, como mencionaba en la sección precedente. Tenemos también el problema del consumo de memoria, que será proporcional al número total de sesiones que maneje el servidor.

Para paliar un poco las dificultades, el componente *TSessionsService* tiene dos propiedades: *DefaultTimeout*, que es la cantidad de minutos que puede estar inactiva una sesión, y *MaxSessions*, el número máximo de sesiones que permitiremos. No confunda este último número con el de conexiones concurrentes. Si cada petición se maneja con la suficiente rapidez, puede que un servidor llegue a acumular miles de sesiones en poco tiempo y, sin embargo, haber manejado concurrentemente sólo una o dos peticiones. Esta última propiedad es la que se limita mediante la propiedad *MaxConnections* del objeto global de aplicación.

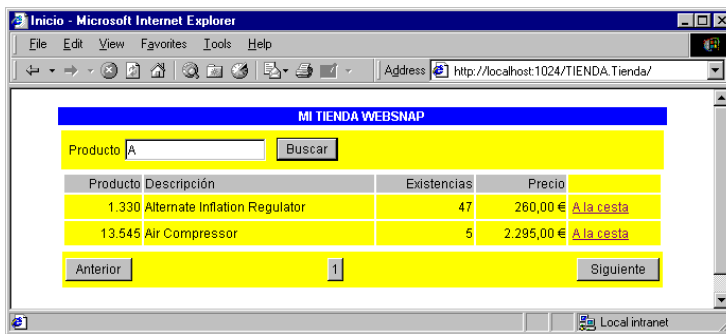
## Una tienda con WebSnap

Como demostración sencilla, vamos a simular el proceso de selección de productos en una tienda para añadirlos al carro de la compra. Haremos que la cesta de la compra se almacene en una variable global de sesión, a la que llamaremos *basket*. En el módulo de datos del proyecto WebSnap que hemos iniciado dejaremos caer un componente *TClientDataSet* y le cambiaremos el nombre a *tbProductos*. Para alimentarlo, utilizaremos una copia del fichero *parts.xml*. Después de añadir los campos al conjunto de datos, traeremos un *TDataSetAdapter*, que bautizaremos *Productos* y asociaremos a *tbProductos*. Debemos entonces añadir los campos de adaptador, sin olvidar

retocar la propiedad *FieldFlags* de la clave primaria: el campo *PartNo*. De las acciones solamente necesitaremos a *PrevPage*, *NextPage* y *GotoPage*.

Para implementar la búsqueda de productos por su descripción añadiremos al adaptador un campo, *AdaptFilter*, y una acción *ActionFilter*. Ya hemos visto cómo configurarlos para aplicar un filtro al conjunto de datos, y no creo que sea necesario repetir los pasos. Aproveche este momento y añada una segunda acción, *ActionBasket*, que utilizaremos para añadir un producto, desde el resultado de la consulta, a la cesta.

Como hemos puesto los componentes de aplicación en un módulo de datos, necesitaremos al menos una página adicional, que deberá utilizar un *TAdapterPageProducer* para generar su contenido. Llámela *Inicio*. Su aspecto final será parecido al siguiente:



No hay mucho misterio en el diseño de la página, excepto que he añadido, dentro del componente de rejilla, un *AdapterCommandColumn*, que es una columna preparada para mostrar uno o más comandos. Dentro de la columna hay que añadir solamente la acción *ActionBasket*, la segunda que hemos creado. Por omisión, el comando aparecería como un botón, pero he cambiado su propiedad *DisplayType* a *ctAnchor*, para que muestre un enlace.

¿Cómo podemos representar la cesta o carro de la compra dentro de una variable de tipo *Variant*? Crearemos una clase *TBasket*, que almacene la lista de productos y sus cantidades, y la dotaremos de una propiedad que “comprima” todos sus datos dentro de un *Variant*. Esta es la interfaz pública de *TBasket*, declarada e implementada en la unidad *UBasket*, en el CD del libro:

```
type
  TBasket = class(TComponent)
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;

    property AsVariant: Variant;
    property ItemCount: Integer;
    property Items[I: Integer]: string; default;
    property Qty[I: Integer]: Integer;
    property Units[const Item: string]: Integer;
  end;
```

Básicamente, una cesta contiene una cantidad variable de productos, *ItemCount*. *Items* devuelve el nombre de un producto dada su posición en la cesta, y *Qty* contiene el número de unidades para ese producto. Ambas propiedades trabajan con la posición, y son útiles para generar representaciones visuales de la cesta. Por otra parte, la propiedad vectorial *Units* acepta como índice la descripción del producto, y devuelve también la cantidad de unidades que hay en la cesta. Si el producto no existe, *Units* devuelve 0; de este modo, tienen sentido instrucciones como la siguiente:

```
// Si no hay Coca Cola, se añadirá una unidad
FBasket.Units['Coca Cola'] := FBasket.Units['Coca Cola'] + 1;
```

Pero lo más interesante de *TBasket* es la propiedad *AsVariant*. Cuando leemos su valor, recibimos una representación lineal del contenido de la cesta, dentro de una cadena de caracteres. Por el contrario, si asignamos en esa propiedad un variante que contenga cadena correctamente formada, el contenido de la cesta se ajusta al valor codificado. Adicionalmente, las constantes especiales *Unassigned* y *Null* se interpretan como cestas vacías.

Por último, crearemos una instancia de la cesta dentro del módulo de datos. Mi objetivo es evitar la creación desenfrenada de objetos temporales de cestas, cada vez que sea necesario operar con una de ellas. Esto puede evitarnos algo de fragmentación de la memoria del servidor:

```
type
  TmodDatos = class(TWebAppDataModule)
    // ...
  private
    FBasket: TBasket;
    // ...
  end;
```

La variable se inicializa durante la construcción del módulo de datos:

```
procedure TmodDatos.WebAppDataModuleCreate(Sender: TObject);
begin
  FBasket := TBasket.Create(Self);
end;
```

Y no hace falta destruirla; al ser un componente y especificar que su propietario es el módulo de datos, se destruye automáticamente cuando este último dice adiós.

## Añadir a la cesta

Ahora nos ocuparemos de la acción de añadir a la cesta. Debe tener presente que *ActionBasket* va siempre ligada a una fila concreta del adaptador. Para asociarle un código de producto utilizaremos los parámetros de la acción, que se definen durante la respuesta al evento *OnGetParams*:

```
procedure TmodDatos.ActionBasketGetParams(Sender: TObject;
  Params: TStrings);
```

```
begin
  Params.Add('PartNo=' + tbProductosPartNo.Text)
end;
```

Ese parámetro se recupera durante la respuesta a *OnExecute*, si el usuario decide pulsar sobre el enlace o botón de comando vinculado a *ActionBasket*:

```
procedure TmodDatos.ActionBasketExecute(Sender: TObject;
  Params: TStrings);
var
  AItem: string;
  SaveActive: Boolean;
begin
  FBasket.AsVariant := WebContext.Session.Values['BASKET'];
  SaveActive := tbProductos.Active;
  tbProductos.Open;
  try
    AItem := tbProductos.Lookup('PARTNO',
      Params.Values['PartNo'], 'DESCRIPTION');
  finally
    tbProductos.Active := SaveActive;
  end;
  FBasket.Units[AItem] := FBasket.Units[AItem] + 1;
  WebContext.Session.Values['BASKET'] := FBasket.AsVariant;
end;
```

Es ahí donde está la enjundia: el contenido actual de la cesta para esa sesión se encuentra dentro de una de las variables de sesión, comprimido dentro de un *Variant*. Aprovechamos el objeto *FBasket* del módulo para restaurar la cesta en un formato más manejable, y añadimos entonces una unidad del producto seleccionado a la misma. Luego volvemos a comprimir su contenido y lo volvemos a almacenar en la variable de sesión.

De nada sirve poder añadir productos a una cesta invisible. El siguiente método público del módulo de datos crea una lista HTML muy sencilla a partir del contenido de la cesta:

```
function TmodDatos.DescribeBasket: string;
var
  I: Integer;
begin
  FBasket.AsVariant := WebContext.Session.Values['BASKET'];
  Result := '';
  for I := 0 to FBasket.ItemCount - 1 do
    Result := Format('%s'#13#10'<li>%s: %d unidades</li>',
      [Result, FBasket[I], FBasket.Qty[I]]);
  if Result = '' then
    Result := '&nbsp;';
  else
    Result := '<ul>' + Result + '</ul>';
  end;
end;
```

Un poco más difícil será buscar la forma de insertar el contenido de la cesta dentro de la página... pero sólo si nos empeñamos en utilizar el *TAdapterPageProducer* para ello. Todo se vuelve fácil si definimos una nueva etiqueta transparente, #BASKET, que

incluiremos dentro de la plantilla. Dedíquese un vistazo al contenido final de la plantilla, porque he incluido un par de trucos adicionales para mejorar la presentación:

```
<html>

  <head>
    <title><%= Page.Title %></title>
  </head>

  <body>
    <#STYLES>
    <#WARNINGS>
    <table width=90% align=center>
      <tr>
        <td bgcolor=#90a0e0 align=center>
          <font color=white><b>MI TIENDA WEBSNAP</b></font>
        </td>
      </tr>
      <tr><td><#SERVERSCRIPT></td></tr>
      <tr>
        <td bgcolor=#90a0e0 align=center>
          <font color=white><b>MI CESTA DE LA COMPRA</b></font>
        </td>
      </tr>
      <tr><td bgcolor=#f0f8ff align=left><#BASKET></td></tr>
    </table>
  </body>
</html>
```

Tenga en cuenta que podemos incluso generar código JScript en la sustitución de una etiqueta transparente, porque la evaluación del *script* tiene lugar como último paso de la respuesta.

Por último, tenemos que interceptar el evento *OnHTMLTag* de *AdapterPageProducer1*:

```
procedure TInicio.AdapterPageProducerHTMLTag(Sender: TObject;
  Tag: TTag; const TagString: string; TagParams: TStrings;
  var ReplaceText: string);
begin
  if SameText(TagString, 'BASKET') then
    ReplaceText := modDatos.DescribeBasket;
end;
```

## NOTA

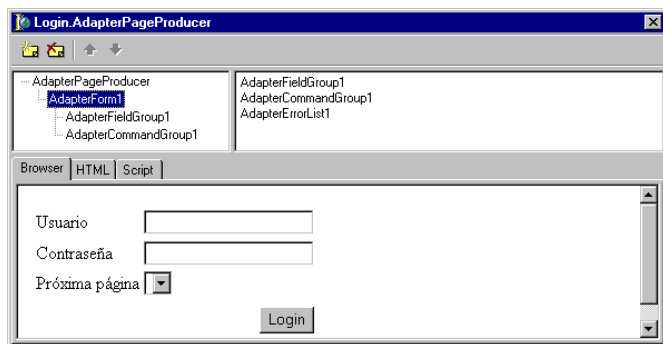
Una solución más elaborada sería crear una nueva clase de adaptador para representar una cesta de la compra. De este modo podríamos acceder a la cesta desde la plantilla de página, utilizando JScript, para poder cambiar el formato de la cesta sin necesidad de realizar modificaciones en Delphi (como sucede ahora). El proceso requeriría que registrásemos las clases de la cesta y de cada elemento de la cesta, y que definiésemos y registrásemos un enumerador para recorrer el contenido de una cesta. Pero eso va más allá del alcance de este libro.

## Usuarios

En una aplicación para la Web es importante saber quién se encuentra al otro lado de la red, manejando el explorador. Aunque no es obligatorio, es muy conveniente activar este mecanismo una vez que se ha implementado el servicio de sesiones. De este modo, no hay que recurrir a campos ocultos ni trucos parecidos para mantener esta información de una petición a la siguiente.

WebSnap ofrece componentes especiales para el control de usuarios y de sus permisos de acceso. Enumeremos los pasos necesarios para configurar estos componentes:

- 1 Primero, debemos asignar un componente apropiado en la propiedad *EndUserAdapter* del componente *WebAppComponents*. Hay dos de estos componentes actualmente; el que nos interesa es *TEndUserSessionAdapter*, que utiliza el servicio de sesiones para “recordar” a la aplicación cuál usuario está conectado en cada momento. Si es que hay alguno, claro. El adaptador de usuario final tiene campos y acciones; es recomendable crearlos explícitamente.
- 2 No es obligatorio, pero sí muy conveniente traer también un *TWebUserList*, para almacenar la lista de usuarios con permisos de acceso.
- 3 Si queremos que los usuarios se identifiquen, necesitaremos un formulario de identificación, ¿verdad? Pero en WebSnap, controles de edición equivale a campos de adaptador, y botones de formulario se pronuncia “acciones”. Hay que traer entonces un componente *TLoginFormAdapter*, que contiene los campos y acciones necesarios. No olvide editar sus propiedades *Actions* y *Data* para crear explícitamente estos componentes.
- 4 Hay que crear entonces la página de identificación, basada en el componente *LoginFormAdapter1* que acabamos de traer al módulo de aplicación. Finalmente, regresamos al componente *EndUserSessionAdapter1*, para asignar el nombre de la nueva página, que para ponernos de acuerdo llamaremos *Login*, en su propiedad *LoginForm*.

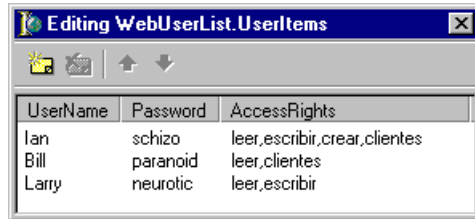


La creación del formulario de identificación no presenta mayor dificultad. No debemos olvidar añadir un *AdapterErrorList*, para mostrar los errores detectados durante la validación. Observe también que existe un campo para seleccionar la página a la



que queremos ir tras pulsar el botón de aceptar. No hace falta incluir este campo en el formulario, sin embargo.

Para rematar la faena, haga doble clic sobre el componente *WebUserList1*:



Es aquí donde WebSnap espera que tecleemos los nombres de usuarios, sus contraseñas y sus permisos de acceso. ¿Leer, escribir, crear? Tenemos total libertad de llamar como se nos ocurra a los permisos de acceso que establezcamos. Para poder realizar un par de pruebas en común, le sugiero que añada los usuarios que aparecen en la imagen. Tenga cuidado, que los nombres de usuario y las contraseñas son sensibles a mayúsculas y minúsculas.

### ¡NO SE PREOCUPE!

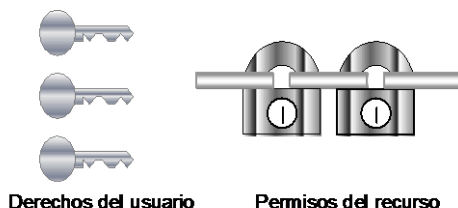
Cada vez que he explicado a alguien el papel de *TWebUserList* he visto la misma reacción: una palmada en la frente y un bufido. Es lógico, porque este sistema de usuarios “estáticos” está muy bien para páginas personales, pero una página comercial tendrá que manejar, como regla general, desde miles a cientos de miles usuarios registrados, cada uno con un nombre y contraseña diferente. No debe preocuparse, sin embargo, porque más adelante mostraré cómo ampliar este sencillo esquema de identificación aprovechando varios eventos de WebSnap.

## Permisos de acceso y acciones

¿Para qué queremos identificar a los usuarios? En primer lugar, para saber a quién tenemos que enviarle la colección completa de muñecas Barbie que algún chalado acaba de comprarnos. No obstante, ahora nos interesa más saber cómo conceder o denegar el acceso a diversos recursos de nuestra página. Estas son las posibilidades que tenemos:

- 1 Exigir que el usuario se haya identificado antes de que pueda navegar a determinadas páginas. Esto funciona en la práctica como un filtro binario: si no estás en la lista de invitados, no saldrás en la foto.
- 2 Lograríamos un nivel superior de control si exigiésemos, además, que el usuario posea determinados permisos de acceso para ver ciertas páginas. Es decir, que aunque estés en la lista de invitados, no te invitaré al ático para mostrarte mi colección de sellos... a no ser que seas una chica guapa, de grandes credenciales.
- 3 Por último, podemos llevar el control a un nivel más detallado, asociando permisos a acciones individuales. No quiero añadir un símil esta vez, porque ninguno de los que se me ocurren pasan por la prueba de lo políticamente correcto.

Paradójicamente, la tercera opción es la más sencilla de implementar en WebSnap. Todas las acciones cuentan con una propiedad llamada *ExecuteAccess*, en la que podemos asignar una lista de permisos separados por comas, o puntos y comas. ¿Cuál es el algoritmo que WebSnap utiliza para comparar los derechos que posee un usuario con los derechos requeridos por un recurso? Póngale un poco de imaginación y buena voluntad al asunto, y observe la siguiente pintura rupestre, realizada con colorantes ecológicos (¡el plomo es un producto “natural”!):



Cada permiso que le otorgamos a un usuario es (además de un error por nuestra parte) una llave. Cada permiso que asignamos a una acción es un candado con la que la protegemos. Pero cuando utilizamos varios candados, en realidad estamos uniéndolos en “serie”, como si se tratase de eslabones en una cadena. Basta con abrir uno de ellos para tener acceso al recurso.

Añada a la aplicación actual una página, a la que llamaremos *Clientes*. En el módulo de componentes de aplicación añada también la tabla XML correspondiente, con su adaptador. Y no olvide crear todos los campos y acciones de éste último. Luego entre en las acciones del adaptador y modifique las propiedades *ExecuteAccess* de las tres acciones de actualización: *ActionDeleteRow*, *ActionNewRow* y *ActionEditRow*. Digamos, por ejemplo, que para editar se requiere el permiso *escribir*, y para añadir o eliminar, el de *crear*. Luego vaya al *AdapterPageProducer* de la nueva página, cree un formulario de navegación y edición, y añada botones para las acciones de actualización.

Ejecute entonces la aplicación. Mientras no se haya identificado, WebSnap considerará que no tiene usted privilegio alguno. Por lo tanto, si entra en la página de clientes, tendrá vedado el uso de los botones de actualización. No obstante los botones “estarán ahí”. El problema se producirá cuando intente ejecutar la acción: WebSnap le mostrará una página con el inevitable regaño.

¿No sería mejor, en cambio, evitarle al usuario la tentación, escondiendo la fruta prohibida? La clave está en la propiedad *HideOptions* de los botones asociados a comandos: tenemos que activar la opción *bboHideOnNoExecuteAccess*. Haga las modificaciones pertinentes y vuelva a ejecutar la aplicación.

## Verificación de acceso a páginas

Veamos ahora cómo exigir que un usuario se identifique antes de poder acceder a una página determinada. No se trata en modo alguno de un secreto: cada vez que

creamos una página para WebSnap, nos salta a la cara una opción dentro del asistente para activar la identificación:

¿Qué pasaría si se nos olvidase activar esta opción? Pues que tendríamos que ir a la sección de inicialización de la unidad de la página, y modificar la instrucción de registro que WebSnap genera. Esta es la versión original:

```
initialization
  if WebRequestHandler <> nil then
    WebRequestHandler.AddWebModuleFactory(
      TWebPageModuleFactory.Create(TClientes,
        TWebPageInfo.Create([wpPublished {, wpLoginRequired}],
          '.html'), crOnDemand, caCache));
  end.
```

Como verá, nos lo han puesto muy fácil: solamente hay que quitar los comentarios que esconden la opción *wpLoginRequired* dentro del constructor de *TWebPageInfo*. Observe de paso las restantes opciones que pueden modificarse en esta instrucción. En particular, esta es la declaración del constructor de *TWebPageInfo*:

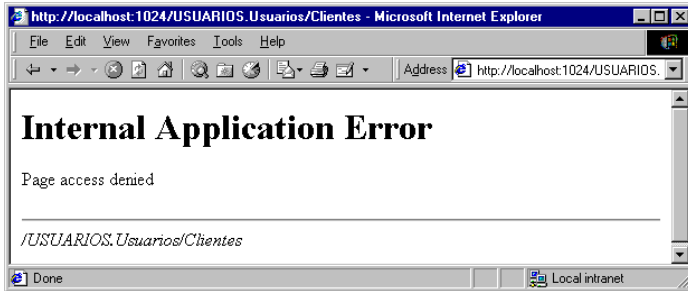
```
constructor Create(
  AAccess: TWebPageAccess = [wpPublished];
  const APageFile: string = '.html'; const APageName: string = '';
  const ACaption: string = ''; const ADescription: string = '';
  const AViewAccess: string = '');
```

¡Hay cuatro parámetros para los que estamos dejando sus valores por omisión! En realidad, el más útil de ellos es el último: *AViewAccess*, que establece permisos obligatorios para poder utilizar una página. Esta es una restricción más potente que verificar simplemente que el usuario se ha identificado. Modifique entonces los parámetros del constructor de *TWebPageInfo*: después del parámetro que contiene la extensión de la plantilla, añada tres cadenas vacías, y una cadena con el permiso necesario para ver la página. En este caso, pruebe con el permiso *'clientes'*:

```
initialization
  if WebRequestHandler <> nil then
    WebRequestHandler.AddWebModuleFactory(
      TWebPageModuleFactory.Create(TClientes,
        TWebPageInfo.Create([wpPublished {, wpLoginRequired}],
          '.html', '', '', '', 'clientes'),
          crOnDemand, caCache));
  end.
```

Después de esta pequeña modificación, la aplicación ya puede verificar los permisos de acceso a las páginas protegidas. Pero nos queda por resolver un pequeño contra-

tiempo. Cuando un usuario intenta acceder a una página prohibida, recibe este horrible pantallazo:



¿Podemos mostrar nuestra propia página? Por supuesto, siempre que estemos dispuestos a escribir un poco de código en respuesta al evento *OnPageAccessDenied* del componente *PageDispatcher*, uno de los componentes del módulo de aplicación:

```
procedure TInicio.PageDispatcherPageAccessDenied(
  Sender: TObject; const PageName: string;
  Reason: TPageAccessDenied; var Handled: Boolean);
begin
  if Reason = adCantView then
  begin
    DispatchPageName('Denegado', Response, []);
    Handled := True;
  end;
end;
```

En este caso, *Denegado* es una página especialmente diseñada, con un mensaje desalentador para el intruso; naturalmente, no debe ser una página publicada, ni requerir identificación. Hay dos razones para denegar el acceso a una página: cuando no hay permisos suficientes, el parámetro *Reason* contiene *adCantView*. El otro valor posible es *adLoginRequired*, para cuando el usuario no se ha identificado todavía. Tome nota del uso del procedimiento *DispatchPageName*, para enviar una página directamente como respuesta al usuario.

## Verificación sobre una base de datos

Para que el sistema de identificación sea realmente útil, debemos tener la posibilidad de almacenar la lista de usuarios en una base de datos. Pero, para no atarnos a un formato rígido de representación, WebSnap resuelve el problema mediante el uso de eventos. En concreto, el evento *BeforeValidateUser* del componente *TWebUserList*. Analice el siguiente ejemplo:

```
procedure TInicio.WebUserListBeforeValidateUser(Strings: TStrings;
  var UserID: Variant; var Handled: Boolean);
begin
  tbEmpleados.Open;
  try
    if SameText(VarToStr(tbEmpleados.Lookup(
```

```

        'LastName', Strings.Values['UserName'], 'FirstName')),
        Strings.Values['Password']) then
    begin
        UserID := Strings.Values['UserName'];
        Handled := True;
    end;
finally
    tbEmpleados.Close;
end;
end;
end;

```

Para este ejemplo, he traído al módulo de aplicación un conjunto de datos XML con la lista de empleados que utilizamos habitualmente. Vamos a utilizar su columna *LastName* como si se tratase de nombres de usuarios, y para las contraseñas usaremos *FirstName*. Al dispararse el evento de validación, abrimos la tabla temporalmente, y buscamos una fila cuya columna *LastName* contenga el nombre del usuario. El nombre del usuario y la contraseña que ha tecleado vienen en el parámetro *Strings*, en este formato:

```

UserName=Nelson
Password=Roberto

```

Si *Lookup* encuentra la fila, devuelve el nombre, que interpretamos en este ejemplo como la contraseña; en caso contrario, devuelve *Null*. Por este motivo es que he utilizado *VarToStr*, que transforma los nulos en cadenas vacías. Como la comparación se realiza con *SameText*, no importan las mayúsculas o minúsculas en la contraseña.

Observe además que sólo asignamos *True* en el parámetro *Handled* cuando encontramos el registro y coincide la contraseña. En caso contrario, el algoritmo sigue su ejecución habitual, y se busca el usuario en la propia lista de *TWebUserList*.

Para la comprobación de permisos debemos utilizar otro evento, llamado *BeforeCheckAccessRights* y disparado por el mismo componente:

```

procedure TInicio.WebUserListBeforeCheckAccessRights (
    UserID: Variant; Rights: string;
    var HasRight, Handled: Boolean);
begin
    tbEmpleados.Open;
    try
        if tbEmpleados.Locate('LastName', UserID, []) then
            begin
                HasRight := (Rights = '') or SameText(Rights, 'clientes');
                Handled := True;
            end;
        finally
            tbEmpleados.Close;
        end;
    end;
end;

```

En el parámetro *UserID* nos dicen de qué usuario se trata; tenga cuidado, porque pueden pasarnos la constante *Unassigned*, cuando no se ha realizado todavía la identificación. En nuestro ejemplo, comprobamos primero si se trata de uno de *nuestros*

usuarios o de alguno de los usuarios predefinidos dentro de la lista (*Ian, Bill, Larry* y el resto de la orquesta). Porque si es de los nuestros y no respondemos asignando *True* a *Handled*, se mostrará un mensaje de error incómodo: *UserID not found*. Tome también nota de que nos pasarán también la cadena vacía en el parámetro *Rights*; eso sucederá con recursos que no están protegidos con permisos de acceso.

### ADVERTENCIA

La verificación de permisos se realizará con bastante frecuencia. Ponga un punto de ruptura en ese evento, y comprobará que se dispara, por ejemplo, cada vez que se va a generar un botón de comando en la página de datos de clientes. Quizás sea conveniente, en consecuencia, almacenar la información de acceso en las variables de sesión, como si se tratase de una caché.

## La plantilla estándar

Ahora que tenemos todos los elementos en la mano, quiero que repasemos el contenido de la plantilla que WebSnap utiliza por omisión para las páginas nuevas. En vez de mostrar todo el listado de golpe, vamos a analizar cada fragmento por separado. Por ejemplo, el fragmento inicial no tiene misterio alguno. En él intervienen solamente el adaptador de aplicación global y la página activa:

```
<html>
<head>
  <title><%= Page.Title %></title>
</head>
<body>
<h1><%= Application.Title %></h1>
```

La siguiente sección ya es más interesante, porque utiliza el adaptador de usuarios, visible desde JScript con el nombre de *EndUser*, para las tareas básicas de identificación y de paso al anonimato (*logout*):

```
<% if (EndUser.Logout != null) { %>
<%   if (EndUser.DisplayName != '') { %>
  <h1>Welcome <%=EndUser.DisplayName %></h1>
<%   } %>
<%   if (EndUser.Logout.Enabled) { %>
    <a href="<%=EndUser.Logout.ASHREF%">Logout</a>
<%   } %>
<%   if (EndUser.LoginForm.Enabled) { %>
    <a href="<%=EndUser.LoginForm.ASHREF%">Login</a>
<%   } %>
<% } %>
```

Hay solamente un problema con el código anterior: la insistencia en mostrar el *DisplayName* del usuario conectado. Para *TEndUserSessionAdapter*, por algún motivo que no acabo de entender, el valor de *DisplayName* es el mismo *UserID* que se almacena en variables de sesión. Normalmente, el *UserID* es un valor que funciona como una clave primaria; entre otras cosas, mientras más compacto sea, mejor. Todo lo contrario del *DisplayName*. ¿Es grave? No mucho. Siempre podremos añadir campos adi-

cional al adaptador, y utilizar también las variables de sesión para almacenar el nombre completo del usuario.

A continuación viene la sección de código más larga de la plantilla. Su objetivo es mostrar una lista de enlaces a las páginas publicadas por la aplicación. He añadido algunos puntos y comas, prescindibles para JScript, pero que ayudan a la lectura:

```
<h2><%= Page.Title %></h2>
<table cellpadding="0" cellspacing="0">
<td>
<%
    e = new Enumerator(Pages);
    s = ''; c = 0;
    for (; !e.atEnd(); e.MoveNext()) {
        if (e.item().Published) {
            if (c>0) s += '&nbsp;|&nbsp;';
            if (Page.Name != e.item().Name)
                s += '<a href="' + e.item().HREF + '">' +
                    e.item().Title + '</a>'
            else
                s += e.item().Title;
            c++;
        }
    }
    if (c>1) Response.Write(s);
%>
</td>
</table>
```

Aquí también conviene realizar cambios. En particular, deberíamos retocar, dentro del bucle, el código que decide si mostrar o no el enlace a una página. Esta es la condición actual:

```
if (e.item().Published) { /* ... */ }
```

Deberíamos modificarla de esta manera:

```
if (e.item().Published && e.item().CanView) { /* ... */ }
```

De esta manera, sólo se mostrarían los enlaces a las páginas en las que puede entrar el usuario activo.





## Servicios Web

COMO SUELE OCURRIR, SIEMPRE HAY UN FETICHE a mano en el mundo de la Informática. El último grito de la moda son los *servicios Web*, una forma bastante sencilla de automatización remota independiente del lenguaje e incluso de la plataforma. Y como de costumbre, es el entusiasmo de sus adoradores lo que hace que el objeto de culto parezca más grande e importante de lo que es en realidad.

Pero la sencillez no quita utilidad al invento. Veamos para qué sirven estos servicios y cómo Borland pone su desarrollo al alcance de todos. Hasta de los abogados.

### Simple Object Access Protocol

Estrictamente hablando, un *Web Service* es cualquier tipo de servicio remoto basado en objetos que esté disponible en Internet. Se debe exigir, por supuesto, que se pueda acceder con cierta facilidad al servicio sin importar el lenguaje con el que se creen las aplicaciones clientes. Un servidor de DataSnap, por ejemplo, cumpliría el primer requisito, pero no el segundo; aunque se podrían escribir los clientes con Delphi, C++ Builder o JBuilder, dudo mucho de que se pueda hacer lo mismo con cualquier de los lenguajes de programación de Microsoft.

Desde un punto de vista más pragmático, la palabreja se refiere en concreto a servicios que utilizan HTTP como protocolo de transporte, y una especificación llamada *SOAP* (*Simple Object Access Protocol*, o Protocolo de Acceso a Objetos Simples) para el formato de los mensajes que intercambian el cliente y su servidor. SOAP se basa en XML para definir el formato de sus mensajes; no voy a entrar en los detalles sintácticos porque no nos harán falta. Lo único que debemos saber es que uno de los objetivos de este protocolo de comunicación es que los mensajes sean lo más cortos y sencillos que sea posible.

¿Cuál es el modelo de objetos de SOAP? Resulta que SOAP no tiene modelo de objetos, a pesar de la O mayúscula en la sigla. El cliente envía un mensaje a un ordenador a través de Internet. El mensaje es recibido por un servidor HTTP, por lo que el mensaje se encapsula dentro de una petición HTTP. En concreto, se utiliza una petición POST, marcada con una cabecera especial, y el texto XML que describe el mensaje se pasa como contenido de la petición.

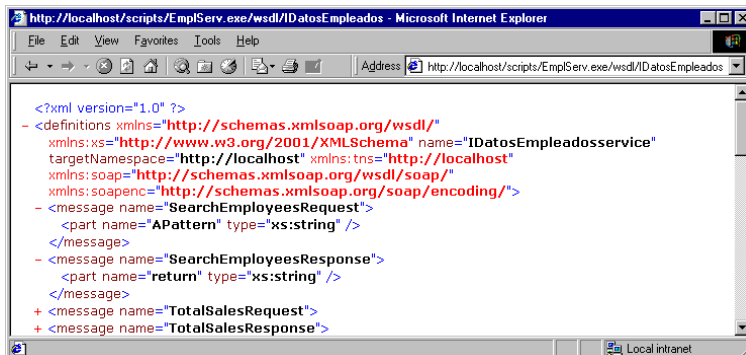
Se supone que la URL mencionada en la petición corresponde a una extensión CGI o similar que es la que implementa el servicio. El propósito de la extensión es devolver al cliente una respuesta HTTP que encapsule un mensaje XML de retorno. Como podrá entender, de esta manera se pueden simular llamadas remotas. En realidad, SOAP es más parecido a RPC que a DCOM o CORBA; esta es la primera decepción que se llevará.

Al utilizar HTTP, además, nos estamos atando a una limitación muy importante de este protocolo: en HTTP la iniciativa siempre corre a cargo del cliente. ¿En qué nos afecta? Pues en que no podemos pedirle al servidor que nos envíe notificaciones o eventos asíncronos. Segunda decepción.

¿Por qué entonces se ha montado tanto revuelo alrededor de SOAP? Hay varias razones, más o menos válidas:

- 1 SOAP utiliza XML, y XML está de moda. Y no me haga decir lo que pienso de la moda XML, por favor.
- 2 SOAP funciona sobre Internet, que también está de moda.
- 3 Cuando SOAP se implementa sobre HTTP, puede atravesar con facilidad cualquier cortafuegos. Aquí tiene una ventaja sobre la implementación tradicional de DCOM y CORBA.
- 4 SOAP no tiene una compañía predominante detrás, así que parece “cosa de todos” y por lo tanto es políticamente correcto.
- 5 SOAP va a ser implementado en la plataforma .NET de Microsoft.
- 6 Las personas para las que el punto 4 es importante, cierran los ojos y fingen desconocer la existencia del punto 5. Las personas a las que le importan el punto 5, se hacen los de la vista gorda cuando le mencionan el punto 4.
- 7 SOAP deja muy bien definidos los formatos de mensajes y la forma de representación de los tipos de datos admitidos.
- 8 SOAP va acompañado de un lenguaje de descripción muy preciso de las capacidades de un servidor: el *Web Services Description Language*, o *WSDL*.

La siguiente imagen muestra una descripción en WSDL de un servicio muy sencillo, cargado en el Internet Explorer:



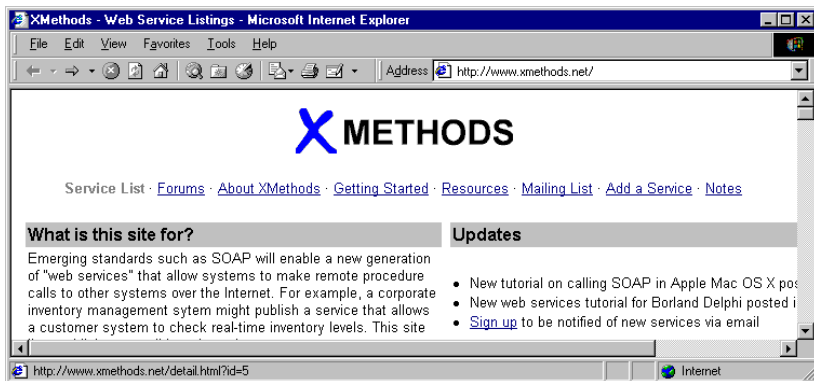
En resumen, SOAP es atractivo por dos motivos principales: por su independencia de la plataforma y porque la precisión de su especificación garantiza la compatibilidad entre lenguajes y plataformas. Pero tampoco es la panacea. En parte porque la especificación actual no incluye un modelo de objetos. Es cierto que podríamos definir uno... pero claro, ya estaríamos saliéndonos del carril. Por otro lado, cuando se vincula a HTTP se le contagian todos los malos modales de este protocolo, en particular, su carácter unidireccional. También es cierto que se puede utilizar otro protocolo para el transporte... pero violaríamos otra vez las reglas del juego.

## Cientes SOAP

Veamos ahora una demostración práctica de la potencia y sencillez de los servicios Web. Supongamos que hay que desarrollar una aplicación para el sector financiero, y que tenemos que tener acceso a las tasas de cambio entre varias monedas. Y queremos que la información más fresca posible.

En otra época, hubiéramos tenido que contactar con algún banco o entidad estatal para que nos prestase ese servicio. Nos habrían dado un manual dos veces más grande que este libro, jeso si tuviéramos suerte!, y tendríamos que aprendernos los detalles de implementación del servicio según se le ocurrió a algún profeta no reconocido de la entidad. Ahora en cambio, buscamos en Internet, a ver si hay algún servidor que ofrezca ese servicio, y si es posible, que sea gratuito. Podemos mirar, por ejemplo, en la siguiente URL:

<http://www.xmethods.net>



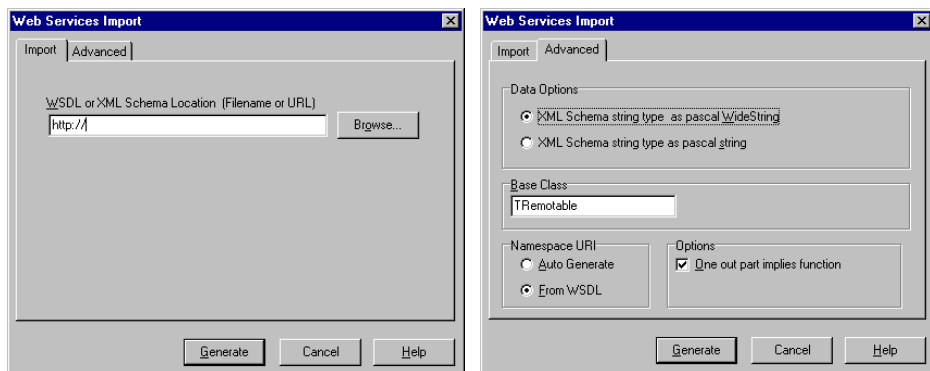
XMethods es una organización fundada por Tony y James Hong con el propósito de promover el desarrollo y uso de servicios Web. Entre esos servicios, se encuentra un directorio de servicios que actualmente están en funcionamiento. En particular, la propia XMethods ha implementado algunos servicios de prueba, entre los que se encuentra precisamente un servicio de información sobre tasas de cambio. Cuando seleccionamos el servicio en el directorio, aparece una página de detalles. El detalle que más nos interesa aparece bajo la etiqueta *WSDL URL*:

<http://www.xmethods.net/sd/2001/CurrencyExchangeService.wsdl>

Esto es lo único que necesitamos saber, de momento. Iniciamos entonces una nueva aplicación que actuará como cliente de ese servicio. No hay restricción alguna en el tipo de cliente que programemos. Por ejemplo, podríamos implementar un servicio Web que utilizase a otros servicios a su vez. Pero en este caso se trata de una aplicación GUI normal. Vaya entonces al Almacén de Objetos, a la página titulada *WebServices*, y ejecute el asistente *Web Services Importer*. ¡Es muy importante que tenga activa en ese momento su conexión a Internet!



El asistente contiene dos páginas, y sólo necesitaremos modificar la primera de ellas:



En el cuadro de edición *WSDL or XML Schema Location* debemos teclear la URL antes mencionada. Si lo desea, pruebe a teclearla también en un navegador de Internet. Comprobará que recibe un fichero XML con lo que parece ser la descripción del servicio. Puede guardar ese fichero y utilizarlo posteriormente para la importación del servicio, en vez de usar una URL como hemos hecho inicialmente.

Cuando cierre el asistente, Delphi habrá añadido una nueva unidad al proyecto activo, con la siguiente declaración de tipo de interfaz:

```
type
  CurrencyExchangePortType = interface(IInvokable)
    ['{04CA9562-063F-11D6-8943-00C026263861}']
```

```
function getRate(const Country1: WideString;
               const Country2: WideString): Single; stdcall;
end;
```

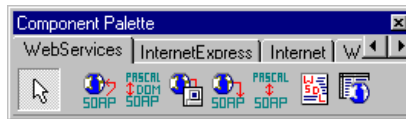
Como puede ver, el servicio es muy sencillo: hay que pasar el nombre de dos países como parámetros de un método llamado *getRate*, y éste nos devolverá la tasa de cambio entre las monedas de los respectivos países. Si mira al final de la unidad, encontrará el siguiente código de inicialización:

```
initialization
  InvRegistry.RegisterInterface (
    TypeInfo(CurrencyExchangePortType),
    'urn:xmethods-CurrencyExchange', '');
end.
```

Esta instrucción aparentemente inocente disfraza un avance notable introducido en Delphi 6: ahora las declaraciones de tipos de interfaz pueden generar información para tiempo de ejecución, o RTTI; antes, sólo las clases tenían esa prerrogativa. Para que un tipo de interfaz genere RTTI debe ser compilado con la opción `{$M+}`, aunque lo más sencillo es que herede de la interfaz *Invokable*, porque los herederos de una interfaz con RTTI también generan esa información.

## Interfaces invocables

Una vez que hayamos registrado la interfaz del servidor, el siguiente paso consiste en traer, a cualquier módulo del proyecto, un componente *THHTTPRIO*, de la página *WebServices* de la Paleta de Componentes; el primero, comenzando por la izquierda:



El nombre lleno de mayúsculas del componente significa *Remote Invokable Object*, es decir, objeto invocable remoto... sí, al cual se accede a través de HTTP. Es un nombre más largo que el título oficial del emperador japonés.

Un *THHTTPRIO*, al que llamaremos *RIO* a partir de este momento, proporciona una interesante implementación al método *QueryInterface* de *IUnknown*. Cuando se le pide la implementación de un tipo de interfaz determinado, busca el identificador de interfaz en una estructura controlada por el objeto global *InvRegistry*. Si hemos registrado antes el tipo de interfaz, *RIO* crea al vuelo un *proxy* cuyo formato corresponda al definido por el tipo. Recuerde que COM es quien realiza la creación de *proxies* cuando accedemos a un servidor COM; *RIO* está suplantando esta función en el lado cliente.

Pero una *v-table* contiene punteros a código. ¿Adónde apuntan las entradas de la *v-table* generada? Como es lógico, a rutinas que implementan que utilizan HTTP para

solicitar la ejecución remota de cada método definido por la interfaz. Los detalles exactos no están disponibles, pues Borland no ha incluido el código fuente de la unidad *Rio*, que es donde se define el ancestro inmediato de *THHTPRIO*.

Ahora que ya sabemos el papel de *HTHPRIO1*, debemos configurar sus propiedades. Hay tres de estas propiedades, y deben asignarse en el mismo orden en que aparecen en la siguiente tabla, para que los editores de propiedades nos echen una mano:

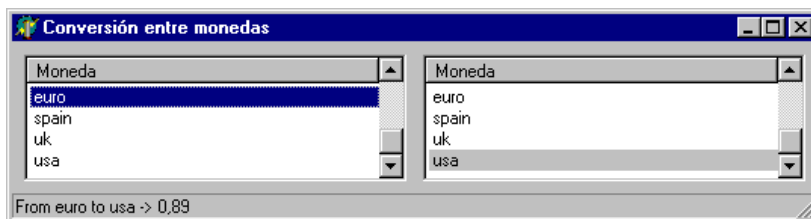
Propiedad	Valor
<i>WSDLLocation</i>	La misma URL que utilizamos en la importación
<i>Service</i>	<i>CurrencyExchangeService</i>
<i>Port</i>	<i>CurrencyExchangePort</i>

La primera propiedad, como hemos visto, indica la URL donde se encuentra la definición en WSDL del servicio Web. En vez de utilizar una URL basada en HTTP, podríamos indicar la ubicación de un fichero que contenga esa misma descripción. Un servidor Web puede ofrecer uno o más *servicios*; un servicio, a su vez, se define como una colección de *puertos*. Y un puerto equivale a un grupo de llamadas, más o menos como los tipos de interfaz de Delphi. Por este motivo es que debemos indicar al *RIO* qué servicio y puerto queremos que represente.

Definamos entonces un método en la clase del formulario similar al siguiente:

```
function TwndMain.GetRate(const Country1, Country2: string): Double;
var
    Port: CurrencyExchangePortType;
begin
    Port := HTHPRIO1 as CurrencyExchangePortType;
    Result := Port.getRate(Country1, Country2);
end;
```

Es aquí donde disparamos la ejecución del método remoto. Primero forzamos la conversión de tipo del componente al tipo de interfaz implementado por el puerto. Recuerde que el operador *as* se implementa, cuando hay tipos de interfaz por medio, llamando al método *QueryInterface*. A partir del momento en que tenemos una variable de tipo *CurrencyExchangePortType*, podemos ejecutar sus métodos como si estuvieran implementados localmente.



En la aplicación del CD se muestran dos listas de “países”; el euro no es un país, al menos de momento, pero lo han incluido. Para averiguar la tasa de cambio entre sus

monedas debemos arrastrar el primero y dejarlo caer sobre el segundo. Como se trata de un servicio Web sencillo, no hay un método que devuelva la lista de países aceptados. Me he limitado a extraer unos cuantos nombres de los documentados en la página Web.

## Servidores SOAP

¿Qué tal si creamos nuestro propio servicio Web? Nos vamos al Almacén de Objetos, y hacemos doble clic sobre el icono *Soap Server Application*:



La interfaz visual del asistente es casi idéntica a la del asistente de WebBroker, excepto por el título del diálogo, precisamente porque un servicio Web debe implementarse dentro de una extensión HTTP. Las consideraciones sobre los modelos disponibles son las mismas: el objetivo final debe ser la creación de una DLL, antes que un ejecutable. Y para depurar, lo adecuado es un ejecutable para el *Web App Debugger*.

Cuando pulsamos el botón de aceptar, sin embargo, veremos que el asistente ha configurado el proyecto de forma diferente. Nos encontraremos con un módulo de datos Web, descendiente de *TWebModule*, con los siguiente componentes:



Antes de explicar el papel de esos componentes, debo aclararle que, excepto modificaciones puntuales, no tendremos necesidad de añadir, quitar o reconfigurar componentes en este módulo. La funcionalidad que implementemos dentro del proyecto se ubicará en otros módulos y unidades.

Ahora las explicaciones: los tres componentes del módulo se dividen en dos grupos separados. Por una parte, tenemos a *WSDLHTMLPublish1* (otro título nobiliario japonés), que como su nombre indica, se encarga de publicar la descripción del servicio en WSDL. Cada vez que una petición HTTP arriba a la aplicación, se analiza su información de ruta, o *path info*, al igual que sucede en WebBroker. El componente WSDL se encarga entonces de responder a todas las peticiones cuya ruta comience con el prefijo *wsdl*.

¿De dónde extrae *WSDLHTTPPublish* la información para publicar la descripción del servicio? Nuevamente se utiliza la variable *InvRegistry*, donde deben haberse registrado de forma distribuida todas las interfaces que implementa el servicio. El componente realiza un barrido sobre esas interfaces y aprovecha la RTTI para crear el documento WSDL.

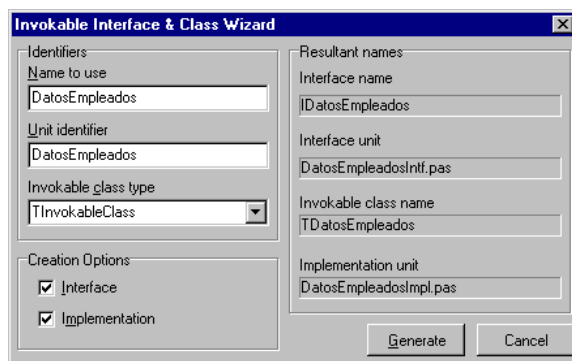
Los otros dos componentes se encargan de implementar el servicio *en sí*. El componente *HTTPSoapDispatcher1* vigila las peticiones, para ocuparse de aquellas cuya ruta comience con *soap*. Cada vez que detecta una, pasa la petición a *HTTPSoapPascalInvoker* (qué manía de poner nombres largos). Este último es el que decodifica la petición y realiza las llamadas a la interfaz registrada más apropiada.

## Creación de interfaces para SOAP

Ya tenemos una plantilla lista para recibir su funcionalidad. Necesitamos dos elementos por cada servicio que añadamos al proyecto:

- 1 Una definición de interfaz, que será registrada dentro de *InvRegistry*.
- 2 Una clase que implemente la interfaz anterior. También será añadida al registro.

En teoría, podríamos programar desde cero ambas cosas. Pero podemos ahorrar algo de tiempo si utilizamos un asistente que Borland “regala” a los usuarios registrados de Delphi 6, llamado *Invokable Wizard*. Hay que entrar en la página de Borland, en el área de usuarios registrados, y descargar el código fuente desde allí. Luego hay que compilarlo e instalarlo dentro del Entorno de Desarrollo.





La imagen anterior corresponde al asistente en plena ejecución. Como puede comprobar, es muy fácil de utilizar. Tenemos que elegir, en primer lugar, el nombre que va a usar como base para crear los nombres de la interfaz y de la clase que la implementa, en *Name to use*. Luego, el nombre que se utilizará como base para las unidades que se van a generar (*Unit Identifier*). En realidad, podríamos mezclar la interfaz con la clase en una misma unidad, pero ya sabe que sería una chapuza horrible: mejor separarlas en dos unidades. En *Invokable class type* debemos indicar la clase base de la que derivaremos la clase que implementa el servicio. Y, finalmente, tenemos la posibilidad de generar solamente la declaración de la clase o de la interfaz.

Al terminar, nos encontraremos con dos nuevos ficheros dentro del proyecto. Vamos a abrir primeramente el que contiene la definición de la interfaz. Modifíquela, añadiéndole un par de métodos como los que muestro a continuación:

```
type
  IDatosEmpleados = interface(IInvokable)
    ['{55DB5F40-05E8-11D6-8943-00C026263861}']
    procedure SearchEmployees(
      const APattern: WideString;
      out IDs: TIntegerDynArray;
      out Names: TStringDynArray); stdcall;
    function TotalSales(AEmpNo: Integer): Double; stdcall;
  end;
```

El primer método recibe como entrada un patrón de búsqueda para localizar empleados en una base de datos. Como parámetros de salida, devolvemos una lista de identificadores de empleados, y una lista paralela del mismo tamaño con los nombres completos de esos mismos empleados. El segundo método recibe un identificador de empleado y calcula el importe total de sus ventas.

Hay que ser muy cuidadosos con los tipos de datos que se utilicen dentro de la interfaz, porque deben ser tipos que Delphi sepa convertir a cadenas de caracteres. Casi todos los tipos escalares predefinidos son aceptables, y también lo son los vectores dinámicos con elementos escalares. Por ejemplo, *TIntegerDynArray* está declarado en la unidad *Types*:

```
type
  TIntegerDynArray = array of Integer;
```

Si desea estar completamente seguro sobre los tipos permitidos, consulte en el código fuente de la unidad *InvokeRegistry*, la función *InitBuiltIns*. Más adelante explicaré cómo añadir nuevas clases o tipos al registro de tipos remotos.

## Implementación de la interfaz remota

Busque ahora la unidad *DatosEmpleadosImpl*, que ha sido creada también por el asistente. En su interior se define una clase que da cuerpo a la interfaz remota *IDatosEmpleados*:

```

type
  TDatosEmpleados = class (TInvokableClass, IDatosEmpleados)
    procedure SearchEmployees (
      const APattern: WideString;
      out IDs: TIntegerDynArray;
      out Names: TStringDynArray); stdcall;
    function TotalSales (AEmpNo: Integer): Double; stdcall;
  end;

```

Al final, en el código de inicialización de la unidad, encontrará una instrucción que registra la clase de implementación. Es necesario que *HTTPSoapPascalInvoker* sepa con qué clases cuenta para responder a las peticiones remotas:

```

initialization
  InvRegistry.RegisterInvokableClass (TDatosEmpleados);
end.

```

Los problemas aparecen cuando hay que implementar los métodos. Excepto en los ejemplos más elementales, un servicio Web debe sintetizar sus respuestas consultando una base de datos, o recurriendo a otros servicios; es irreal que alguien se tome la molestia de realizar una llamada remota para saber cuál es el algoritmo neperiano de la constante de Plank dividida entre dos veces  $\pi$ . Por lo tanto, la clase de implementación necesitará manejar componentes de acceso a datos, o clientes de otros servicios. Estos se pueden crear y configurar al vuelo... pero no es la forma tradicional y productiva de trabajar en Delphi.

En el ejemplo del CD, lo he resuelto de una manera bastante burda, sobre todo para no complicar el primer ejemplo que iba a presentar. Simplemente añadí un módulo de datos “normal” al proyecto. En el nuevo módulo añadí los componentes de datos necesarios, e implementé la misma interfaz *IDatosEmpleados*. Este es el cuerpo de *BuscarEmpleados* en la clase *TmodTablas*, que es como llamé al nuevo módulo:

```

procedure TmodTablas.BuscarEmpleados (
  const Patron: string;
  out IDs: TIntegerDynArray; out Names: TStringDynArray);
var
  I: Integer;
begin
  Empleados.Filter := Format(
    'upper(firstname) like %0:s or upper(lastname) like %0:s',
    [QuotedStr (UpperCase (Patron) + '%')]);
  Empleados.Filtered := True;
  Empleados.Open;
  try
    SetLength (IDs, Empleados.RecordCount);
    SetLength (Names, Empleados.RecordCount);
    I := Low (IDs);
    while not Empleados.Eof do begin
      IDs[I] := Empleados.EmpNo.Value;
      Names[I] := Empleados.FirstName.Value + ' ' +
        Empleados.LastName.Value;
      Inc (I);
      Empleados.Next;
    end;
  end;

```

```

    finally
        Empleados.Close;
    end;
end;

```

Y la implementación de *TotalVentas* es la siguiente:

```

function TmodTablas.TotalVentas (AEmpNo: Integer): Double;
begin
    Result := 0;
    Empleados.Filter := 'EmpNo = ' + IntToStr(AEmpNo);
    Empleados.Filtered := True;
    Empleados.Open;
    try
        if not Empleados.Eof then begin
            Pedidos.Open;
            try
                while not Pedidos.Eof do begin
                    Result := Result + Pedidos.ItemsTotal.Value;
                    Pedidos.Next;
                end;
            finally
                Pedidos.Close;
            end;
        end;
    finally
        Empleados.Close;
    end;
end;

```

Regresando a *DatosEmpleadosImpl*, la implementación de sus métodos procede por simple delegación. Por ejemplo:

```

procedure TDatosEmpleados.SearchEmployees (
    const APattern: WideString;
    out IDs: TIntegerDynArray; out Names: TStringDynArray);
begin
    with TmodTablas.Create(nil) do
        try
            BuscarEmpleados(APattern, IDs, Names);
        finally
            Free;
        end;
    end;
end;

function TDatosEmpleados.TotalSales (AEmpNo: Integer): Double;
begin
    with TmodTablas.Create(nil) do
        try
            Result := TotalVentas(AEmpNo);
        finally
            Free;
        end;
    end;
end;

```

No obstante, insisto: es una implementación chapucera. ¿Por qué? Pues porque hay que crear un módulo nuevo para cada petición. Es cierto que en nuestro ejemplo trabajamos directamente con ficheros MyBase, pero en un caso real cada módulo

necesitaría una conexión a una base de datos SQL o a un servidor de capa intermedia. No pierda de vista que nuestro objetivo final es que el servicio se implemente dentro de una DLL.

¿Podemos usar una caché de módulos? Bueno, ¡ya existe una caché de módulos! La creación de instancias para el módulo principal del proyecto, el que contiene los tres componentes de *WebServices*, es controlada por WebBroker. Cuando el módulo habita dentro de una CGI, cada petición recibe un módulo nuevo, pero en cuanto lo pasamos a una DLL, se activa la caché de módulos. Podríamos colocar los componentes de acceso a datos en el mismo módulo inicial, y realizar las llamadas desde la clase de implementación... El jarro de agua fría es que Delphi no nos ha dejado técnica alguna para acceder a la instancia del módulo principal desde una instancia de la clase de implementación del servicio.

Se me ocurren varias ideas, pero tendríamos que probarlas antes de darlas por buenas. Por ejemplo, podríamos crear un manejador para el evento *OnCreate* del módulo, y guardar el puntero *Self* en una variable *de hilo*:

```
threadvar           // ¡ES SOLO UNA SUGERENCIA!
    Modulo: TmodDatos;

procedure TmodDatos.WebModuleCreate(Sender: TObject);
begin
    Modulo := Self;
end;
```

Las variables declaradas en una sección **threadvar** crean una copia independiente sin inicializar para cada hilo que hace referencias a ellas. Si se cumple la hipótesis de que toda petición sea manejada de principio a fin en un hilo separado, la instancia de *TDatosEmpleados* podría consultar la variable *Modulo* para saber cuál es el módulo que la ha creado.

Mejor todavía, ¿sabe que no es necesario que la clase de implementación descienda de *TInvokableClass*? Podríamos hacer que el propio módulo implementase la interfaz del servicio. Lo único que habría que cambiar es la forma de llama a *RegisterInvokableClass*, suministrando un segundo parámetro con un procedimiento que “fabrique” una instancia de la clase:

```
initialization
    InvRegistry.RegisterInvokableClass(TmodDatos, DevolverInstancia);
end.
```

El procedimiento *DevolverInstancia* sería similar al siguiente:

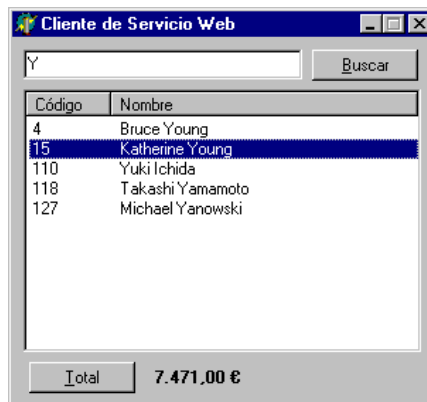
```
procedure DevolverInstancia(out Obj: TObject);
begin
    Obj := Modulo;
end;
```

**MUY IMPORTANTE**

Tenga *muy* presente que esto valdría solamente si se cumpliera la condición que he mencionado antes: un hilo único e independiente para satisfacer cada petición HTTP. De todos modos, si quiere apostar sobre seguro, podría implementar en *DevolverInstancia* una caché separada para las clases de implementación.

**Poniendo a prueba el servicio**

Es un juego de niños crear un cliente para probar el servicio anterior. La única diferencia respecto al primer ejemplo es que no necesitamos importar la interfaz del servicio, sino que podemos reutilizar la unidad *DatosEmpleadosImpl* para saltarnos un paso.



Vea, por ejemplo, cómo se realiza la llamada a *SearchEmployees* y cómo se añaden los empleados encontrados a un componente *TListView*:

```

procedure TwndPrincipal.bnBuscarClick(Sender: TObject);
var
    I: Integer;
    IDs: TIntegerDynArray;
    Names: TStringDynArray;
begin
    (HTTPIO1 as IDatosEmpleados).SearchEmployees (
        edPatron.Text, IDs, Names);
    ListView1.Clear;
    for I := Low(IDs) to High(IDs) do
        with ListView1.Items.Add do begin
            Caption := IntToStr(IDs[I]);
            SubItems.Add(Names[I]);
            Data := Pointer(IDs[I]);
        end;
end;

```

Cuando un *THTTPIO* accede a un servicio Web implementado en Delphi, no es necesario configurar sus tres propiedades *WSDLLocation*, *Service* y *Port*. Basta con llenar URL con la dirección donde se encuentra el servidor, con la ruta SOAP al final.

## Nuevas clases transmisibles

En el ejemplo anterior nos interesaban solamente dos campos del registro de empleados, y pudimos traer la lista de empleados utilizando un par de vectores paralelos. En casos reales, sería mejor transmitir cada empleado como un registro, y que la lista de empleados sea un vector dinámico de ese tipo de registro.

Exactamente como registro de Pascal, o **record**, no es posible. Pero podemos crear una clase derivada de *TRemotable* para que represente a un empleado. Los campos que queramos transmitir deben ser definidos como propiedades publicadas:

```
type
  TSoapEmployee = class(TRemotable)
  private
    FEmpNo: Integer;
    FFirstName, FLastName: string;
    FSalary: Double;
  published
    property EmpNo: Integer read FEmpNo write FEmpNo;
    property FirstName: string read FFirstName write FFirstName;
    property LastName: string read FLastName write FLastName;
    property Salary: Double read FSalary write FSalary;
  end;
```

Necesitamos también registrar la clase para poder usarla. La siguiente instrucción debe colocarse, directa o indirectamente, en la sección de inicialización de alguna unidad; preferiblemente, de la misma unidad donde se ha declarado la clase:

```
initialization
  RemClassRegistry.RegisterXSClass(
    TSoapClass, XMLSchemaNamespace, 'employee', False);
end.
```

El último parámetro indica si la clase puede convertirse, como un todo, en una cadena de caracteres. En este ejemplo he indicado que no, que hay que enviar por separado cada uno de los campos.

### NOTA

El último parámetro de *RegisterXSClass* existe para poder recibir y transmitir tipos complejos como si se tratase de tipos básicos. Consulte, por ejemplo, la declaración de la clase *TXSDateTime* en la unidad *XSBuiltIns*. Observe que este tipo de clases descienden de *TXSRemotable*, y que deben redefinir los métodos *XSToNative* y *NativeToXS*.

De todos modos, *TSoapEmployee* representa un solo empleado, y queremos trabajar con listas de empleados. En tal caso, debemos ejecutar otro tipo de datos más y una función de registro adicional:

```
type
  TSoapEmployees = array of TSoapEmployee;
```

```

initialization
    RemClassRegistry.RegisterXSInfo(
        TypeInfo(TSoapEmployees), '', 'employees', False);
end.

```

## Módulos remotos jabonosos

También es posible utilizar SOAP como medio de comunicación entre aplicaciones y servidores de capa intermedia de DataSnap. El proceso de creación del servidor es el siguiente:

- 1 Tenemos que partir de una *Soap Server Application*, que creamos desde el Almacén de Objetos, en la página *WebServices*, mediante el icono del mismo nombre.
- 2 A continuación, en la misma página del Almacén, hay que añadir un *Soap Server Data Module*.
- 3 Solamente debe existir un módulo SOAP por servidor.

### ADVERTENCIA

Quizás me equivoque, pero no estoy convencido de que estos módulos disfruten de algún tipo de caché en el servidor.

Es muy sencillo crear un cliente DataSnap que utilice el servidor anterior. El componente de conexión necesario se llama *TSoapConnection*, y se encuentra en la página *WebServices*. Observe que esta vez no hay *ServerName*, ni *ServerGUID*; esas propiedades estaban vinculadas a que los anteriores tipos de servidores estaban basados en COM. La dirección del servicio se indica directamente en la propiedad *URL*.

Hay una importante limitación en este tipo de conexiones: no se puede utilizar la propiedad *AppServer* para acceder a otros métodos que los definidos por *LAppServer*. Esto se debe al carácter más estático de SOAP, en comparación con COM. Tampoco es posible implementar eventos en el servidor de capa intermedia, esta vez por culpa del protocolo HTTP. ¿Cuál es entonces el posible atractivo de desarrollar la capa intermedia como un servicio Web? Simplemente, que si a usted le hace ilusión utilizar un servidor remoto basado en el sistema del pingüino, puede programar la capa intermedia con Kylix. Recuerde que COM no funciona en el Polo Norte.







## **Leftoverture**

---

- **Impresión de informes con QuickReport**
- **Gráficos**

# **Parte**



## Impresión de informes con QuickReport

AUNQUE LA PROPAGANDA COMERCIAL intente convencernos de las bondades del libro electrónico, nuestros ojos siempre agradecerán un buen libro impreso en papel. Del mismo modo, toda aplicación de bases de datos debe permitir imprimir la información con la que trabaja. Sistemas de creación e impresión de informes para Delphi hay muchos, quizás demasiados. Inicialmente, junto con Delphi 1 se suministraba un producto de Borland, llamado ReportSmith: no era un mal programa, pero casi todos lo odiábamos. ¿La razón?: un *runtime* bastante grande e incómodo que distribuir, demasiado ineficiente en tiempo y espacio (estamos hablando de los tiempos de Windows 3.1, cuando 16 MB en una máquina era bastante memoria) y, sobre todo, una molesta pantalla de presentación con los créditos de Borland, que aparecía cada vez que se imprimía un informe.

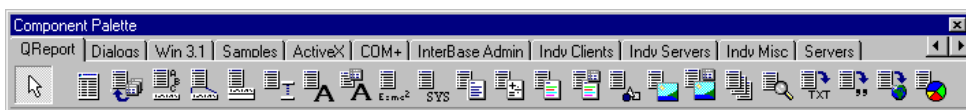
### La filosofía del producto

QuickReport vino al mundo como un pequeño y eficiente sistema de impresión de informes de libre distribución, programado en Noruega. En algún momento, llamó la atención de Borland, que lo incluyó como alternativa a ReportSmith en Delphi 2. A partir de ese momento, ha sido un producto con gafe: *bugs* repetidos, parches a destiempo, parches para los parches, características reclamadas a gritos que nunca se implementan... De todos modos, QuickReport tiene algo muy bueno: como viene incluido con Delphi, te puedes ahorrar una pasta al no tener que comprar un generador de informes realmente profesional. Más no puedo decir a su favor, lo siento.

QuickReport es un sistema de informes basado en *bandas*. Esto quiere decir que durante el diseño del informe no se ve realmente la apariencia final de la impresión, sino un simple esquema, aunque bastante realista. Tomemos un listado simple de una base de datos: la mayor parte de una página estará ocupada con las líneas procedentes de los registros de la tabla. Pues bien, todas estas líneas tienen la misma función y formato y, en la terminología de QuickReport se dice que proceden de una misma banda: la banda de *detalles*. En ese mismo listado se pueden identificar otras bandas: una correspondiente a la cabecera de páginas, la de pie de páginas, etc. Son estas bandas las que se editan y configuran en QuickReport. Durante la edición, la banda

de detalles no se repite, como sucederá durante la impresión. Pero en cualquier momento podemos ver el aspecto final del informe mediante el comando *Preview*.

La otra característica singular es que el proceso de diseño tiene lugar dentro de Delphi, utilizando las propias herramientas de diseño del Entorno de Desarrollo. Los componentes de QuickReport se colocan en un formulario, aunque este formulario solamente sirve como contenedor, y nunca es mostrado al usuario de la aplicación. Como consecuencia, todo el motor de impresión reside dentro del mismo espacio de la aplicación; no es un programa externo con carga independiente. Con esto evitamos las demoras relacionadas con la carga en memoria de un programa de impresión externo. Por supuesto, el código del motor puede utilizarse desde un paquete. Esta es la página de QuickReport en la Paleta de Componentes de Delphi 6:



Otra ventaja de QuickReport es que los datos a imprimir se extraen directamente de conjuntos de datos de Delphi. Una tabla que se está visualizando en una ventana de exploración, sobre la cual hemos aplicado filtros y criterios de ordenación, puede también utilizarse de forma directa para la impresión de un informe. En el listado solamente aparecerán las filas aceptadas por el filtro, en el orden especificado para la tabla. El hecho de que los datos salgan directamente de la aplicación implica que no son necesarias conexiones adicionales durante la impresión del informe. Y esto significa muchas veces, en dependencia del servidor, ahorrar en el número de licencias.

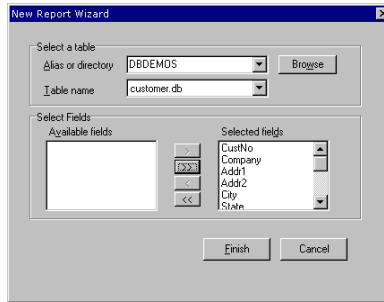
Claro está, también existen inconvenientes para este estilo de creación de informes. Ya hemos mencionado el primero: no hay una retroalimentación inmediata de las acciones de edición. El segundo inconveniente es más sutil. Con un sistema de informes independiente se pueden incluir *a posteriori* informes adicionales, que puede diseñar el propio usuario de la aplicación. Esto no puede realizarse directamente con QuickReport, a no ser que montemos algún mecanismo basado en DLLs o en automatización OLE.

## Plantillas y expertos para QuickReport

QuickReport trae plantillas de formularios para acelerar la creación de informes; las encontrará en la página *Forms* del Almacén de Objetos. Las plantillas son tres: una para listados de una sola tabla, una para informes *master/detail*, y otra para la impresión de etiquetas. No voy a entrar en detalles acerca del trabajo con estas plantillas pues, a partir de la explicación que haré de los componentes de impresión, deducirá fácilmente que se puede hacer con ellas.

Delphi también ofrece un experto para la generación de listados simples, en la página *Business* del Almacén. Cuando ejecutamos el experto, en la primera página debemos

indicar el tipo de informe que queremos generar. En la versión de QuickReport que viene con Delphi, solamente tenemos una posibilidad: *List Report*, es decir, un listado simple. En la siguiente pantalla, debemos seleccionar la tabla cuyos datos vamos a imprimir. Como ve, Quick Report sigue viviendo en la época del BDE:



El resultado es un formulario con una tabla y con los componentes necesarios de QuickReport. Si quiere visualizar el informe generado, pulse el botón derecho del ratón sobre el componente *TQuickRep* y ejecute el comando *Preview*.

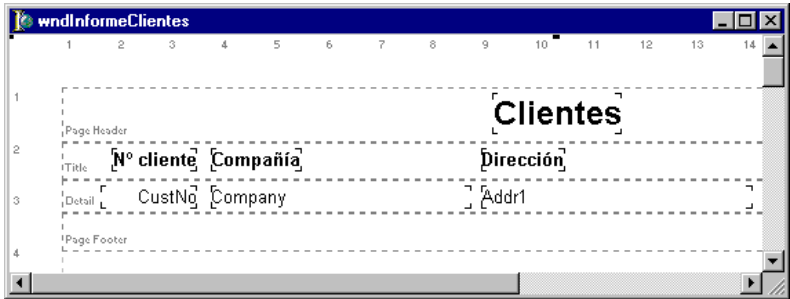
## El corazón de un informe

Ahora estudiaremos cómo montar manualmente los componentes necesarios para un informe. Para definir un informe, traemos a un formulario vacío un componente *TQuickRep*, que ocupará un área dentro del formulario en representación de una página del listado. Luego hay que asignar la propiedad fundamental e imprescindible, que indica de qué tabla principal se extraen los datos que se van a imprimir: *DataSet*. Esta propiedad, como su nombre indica apunta a un conjunto de datos, no a una fuente de datos. Si se trata de un informe *master/detail*, la tabla que se asigna en *DataSet* es la maestra.

La tarea principal de un componente *TQuickRep* es la de iniciar el proceso de impresión, cuando se le aplica uno de los métodos *Print* ó *Preview*. También podemos utilizar el método *PrintBackground*, que realiza la impresión en un proceso en segundo plano. Por ejemplo, el informe generado en la sección anterior se puede imprimir en respuesta a un comando de menú de la ventana principal utilizando el siguiente código:

```
procedure TForm1.Imprimir1Click(Sender: TObject);
begin
    Form2.QuickRep1.Print;
end;
```

La propiedad *ReportTitle* del informe se utiliza como título de la ventana de previsualización predefinida.

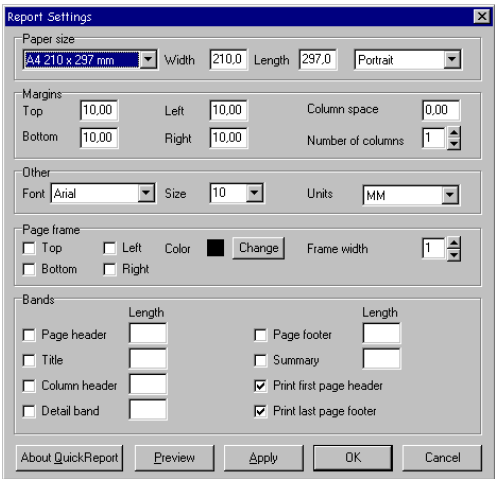


Para obtener la vista preliminar de un informe en tiempo de diseño, utilice el comando *Preview* del menú local del componente. Tenga en cuenta que en tiempo de diseño no se ejecutan los métodos asociados a eventos, así que para ver cualquier opción que haya implementado por código, tendrá ejecutar su aplicación.

La configuración de un informe comienza normalmente por definir las características de la página, que se almacenan dentro de la propiedad *Page*. Esta es una clase con las siguientes propiedades anidadas:

Propiedad	Significado
<i>PaperSize, Length, Width</i>	Tamaño del papel
<i>Orientation</i>	Orientación de la página
<i>LeftMargin, RightMargin</i>	Ancho de los márgenes horizontales
<i>TopMargin, BottomMargin</i>	Ancho de los márgenes verticales
<i>Ruler</i>	Mostrar regla en tiempo de diseño
<i>Columns</i>	Número de columnas
<i>ColumnSpace</i>	Espacio entre columnas

Sin embargo, es más cómodo realizar una doble pulsación sobre el componente *TQuickRep*, para cambiar las propiedades anteriores mediante el siguiente cuadro de edición:



La configuración del tamaño de papel es una de las mayores frustraciones de los programadores de QuickReport, aunque la culpa no es achacable por completo a este producto. En primer lugar, no todos los tamaños de papel son aceptados por todas las impresoras. En segundo lugar, el controlador para Windows de la mayoría de las impresoras no permite definir tamaños personalizados de papel.

Como puede verse en el editor del componente, podemos asignar un tipo de letra por omisión, global a todo el informe. Esta corresponde a la propiedad *Font* del componente *TQuickRep*, y por omisión se emplea la Arial de 10 puntos, que es bastante legible. En el mismo recuadro, a la derecha, se establece en qué unidades se indican las medidas (propiedad *Units*); inicialmente se utilizan milímetros. En el recuadro siguiente se muestran las subpropiedades de la propiedad *Frame*, que sirve para trazar un recuadro alrededor de la página. Pueden seleccionarse las líneas que se van a dibujar, el color, el ancho y su estilo.

## Las bandas

Una *banda* es un componente sobre el cual se colocan los componentes “imprimibles”; en este sentido, una banda actúa como un panel. Sobre las bandas se colocan los *componentes de impresión*, en la posición aproximada en la que queremos que aparezcan impresos. Para ayudarnos en esta tarea, el componente *TQuickRep* muestra un par de reglas en los bordes de la página, que se muestran y ocultan mediante el atributo *Ruler* de la propiedad *Page*.

La propiedad más importante de una banda es *BandType*, y estos son sus posibles valores:

Tipo de banda	Objetivo
<i>rbTitle</i>	Se imprime una sola vez, al principio del informe
<i>rbSummary</i>	Se imprime una sola vez, al terminar el informe
<i>rbPageHeader</i>	Se imprime en cada página, en la cabecera
<i>rbPageFooter</i>	Se imprime en cada página, al final de la misma
<i>rbColumnHeader</i>	Si la página se divide en columnas, al comienzo de cada una
<i>rbDetail</i>	Se imprime para cada registro de la tabla principal
<i>rbSubDetail</i>	Se imprime para cada registro de una tabla dependiente
<i>rbGroupHeader</i>	Se imprime cuando se detecta un cambio de grupo
<i>rbGroupFooter</i>	Se imprime cuando termina la impresión de un grupo
<i>rbChild</i>	Banda hija: se imprime siempre después de su banda madre
<i>rbOverlay</i>	Se sobreimprime sobre cada página

El orden de impresión de los diferentes tipos de bandas, para un listado simple, es el siguiente:

<i>rbPageHeader</i> : En todas las páginas
<i>rbTitle</i> : En la primera página
<i>rbColumnHeader</i> : En cada columna, si las hay

<i>rbDetail</i> : Una banda por cada fila de la tabla
<i>rbSummary</i> : Al final del informe
<i>rbPageFooter</i> : Al final de cada página

Las bandas de tipo *rbChild* se imprimen siempre a continuación de la banda madre, pero podemos ejercer más control sobre ellas mediante eventos. En la siguiente sección veremos un ejemplo.

Las bandas pueden añadirse manualmente al informe. Traemos un componente *TQRBand* desde la Paleta de Componentes, e indicamos el tipo de banda en su propiedad *BandType*. Sin embargo, es más fácil indicar las bandas que necesitamos en el Editor de Componente de *TQuickRep*, en el panel inferior, o mediante la propiedad *Bands* del componente. Tenga en cuenta que los componentes *TQRGroup* y *TQRSubDetail*, que estudiaremos más adelante, son componentes visuales y traen incorporadas sus respectivas bandas. En la primera versión de QuickReport, estos componentes venían separados de sus bandas.

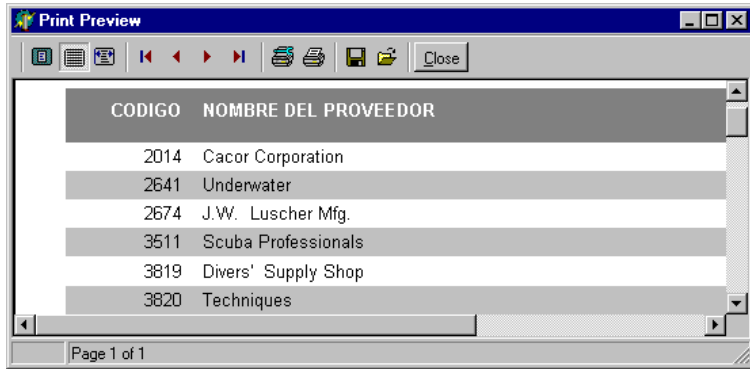
La propiedad *Options* del componente *TQuickRep* permite omitir la impresión de la cabecera en la primera página del listado, y la del pie de página en la última. Esto también puede especificarse en el Editor del componente, en el panel inferior, mediante dos casillas de verificación. De este modo, si definimos una banda de título (*rbTitle*) en el informe, se logra el efecto de tener una cabecera de página diferente para la primera página y para las restantes. Sin embargo, las bandas de resumen (*rbSummary*) se imprimen por omisión justo a continuación de la última banda de detalles. Si queremos que aparezca como si fuera un pie de página, debemos asignar *True* a su propiedad *AlignToBottom*.

## El evento **BeforePrint**

Todas las bandas tienen los eventos *BeforePrint* y *AfterPrint*, que se disparan antes y después de su impresión. Podemos utilizar estos eventos para modificar características de la banda, o de los componentes que contienen, en tiempo de ejecución. Por ejemplo, si queremos que las líneas de un listado simple salgan a rayas con colores alternos, como un pijama, podemos crear la siguiente respuesta al evento *BeforePrint* de la banda de detalles:

```
procedure TForm2.DetailBand1BeforePrint(
  Sender: TQRCustomBand; var PrintBand: Boolean);
begin
  if Sender.Color = clWhite then
    Sender.Color := clSilver
  else
    Sender.Color := clWhite;
end;
```





También podemos impedir que se imprima una banda en determinadas circunstancias. Supongamos que la columna *Direccion2* de la tabla *tbClientes* tiene valores no nulos para pocas filas. Nos interesa mostrar esta segunda línea de dirección solamente cuando vaya a contener algún valor. La solución más elegante es colocar el componente de impresión correspondiente a la columna (ver la siguiente sección) en una banda hija de la banda de detalles. Para evitar la impresión de bandas vacías, se crea el siguiente manejador para el evento *BeforePrint* de la nueva banda:

```
procedure TForm2.ChildBand1BeforePrint(Sender: TQRCustomBand;
  var PrintBand: Boolean);
begin
  PrintBand := not tbClientesDireccion2.IsNull;
end;
```

## Componentes de impresión

Una vez que tenemos bandas, podemos colocar componentes de impresión sobre las mismas. Los componentes de impresión de QuickReport son:

Componente	Imprime...
<i>TQRLabel</i>	Un texto fijo de una línea
<i>TQRMemo</i>	Un texto fijo con varias líneas
<i>TQRRichText</i>	Un texto fijo en formato RTF
<i>TQRImage</i>	Una imagen fija, en uno de los formatos de Delphi
<i>TQRShape</i>	Una figura geométrica simple
<i>TQRSysData</i>	Fecha actual, número de página, número de registro, etc.
<i>TQRDBText</i>	Un texto extraído de una columna
<i>TQRDBRichText</i>	Un texto RTF extraído de una columna
<i>TQRExpr</i>	Una expresión que puede referirse a columnas
<i>TQRExprMemo</i>	Un memo con campos de expresiones en su interior
<i>TQRDBImage</i>	Una imagen extraída de una columna

Todos estos objetos, aunque pertenecen a clases diferentes, tienen rasgos comunes. La propiedad *AutoSize*, por ejemplo, controla el área de impresión en la dimensión

horizontal. Normalmente, esta propiedad debe valer *False*, para evitar que se superpongan entre sí los diferentes componentes de impresión. En tal caso, es necesario agrandar el componente hasta su área máxima de impresión. La posición del texto a imprimir dentro del área asignada se controla mediante la propiedad *Alignment*: a la derecha, al centro o a la izquierda. Ahora bien, el significado de esta propiedad puede verse afectado por el valor de *AlignToBand*. Cuando *AlignToBand* es *True*, *Alignment* indica en qué posición horizontal, con respecto a la banda, se va a imprimir el componente. Da lo mismo, entonces, la posición en que situemos el componente. Todos los componentes de impresión tienen también la propiedad *AutoStretch* que, cuando es *True*, permite que la impresión del componente continúe en varias líneas cuando no cabe en el área de impresión vertical definida.

El componente más frecuentemente empleado es *TQRDBText*, que imprime el contenido de un campo de un conjunto de datos. Hay que configurar sus propiedades *DataSet* y *DataField*. Tiene la ventaja de que aprovecha automáticamente el formato de visualización del campo asociado, algo que no hace el componente alternativo *TQRExpr*, que veremos a continuación. Este componente, además, es capaz de mostrar el contenido de un campo memo, sin mayores complicaciones.

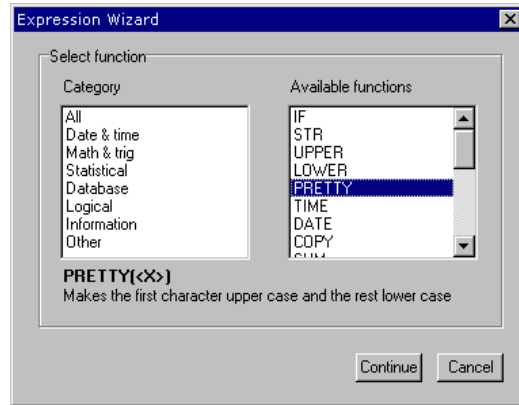
## El evaluador de expresiones

Con QuickReport tenemos la posibilidad de imprimir directamente expresiones que utilizan campos de tablas. Muchas veces, podemos utilizar campos calculados con este propósito, por ejemplo, si queremos imprimir el nombre completo de un empleado, teniendo como columnas bases el nombre y los apellidos por separado. Esto puede ser engorroso, sobre todo si la nueva columna es una necesidad exclusiva del informe, pues tenemos que tener cuidado de que el campo no se visualice accidentalmente en otras partes de la aplicación. Pero también cabe utilizar un componente *TQRExpr*, configurando su propiedad *Expression* como sigue:

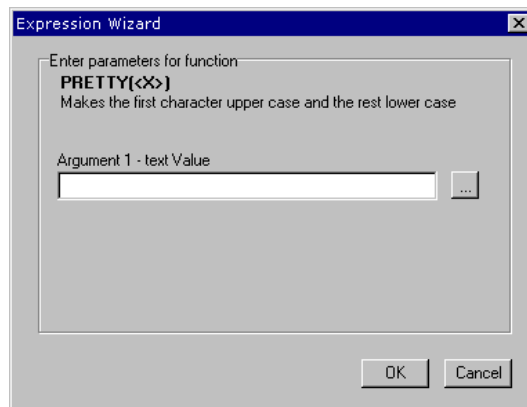
```
tbEmpleados.LastName + ', ' + tbEmpleados.FirstName
```



Para ayudarnos a teclear las expresiones, QuickReport ofrece un editor especializado para la propiedad *Expression*. El editor es un cuadro de diálogo en el cual podemos formar expresiones a partir de constantes, nombres de campos, operadores y funciones. Una expresión como la anterior se compone muy fácilmente con el editor de expresiones. Como vemos en la imagen anterior, podemos utilizar una serie de funciones en la expresión. El siguiente diálogo aparece cuando pulsamos el botón *Function*:



Una vez elegida una función, pulsamos el botón *Continue*, y aparece el siguiente diálogo, para configurar los argumentos:



Si somos lo suficientemente vagos como para pulsar el botón de la derecha, aparece otra instancia del editor de expresiones, para ayudarnos a componer cada uno de los argumentos de la función.

El componente *TQRExpr* también permite definir estadísticas, si utilizamos las funciones de conjuntos de SQL. Si en una banda *rbSummary* quiere que se imprima la suma total de los salarios de la tabla de empleados, utilice la expresión *Sum(Salary)*. La propiedad *ResetAfterPrint* indica si se limpia el acumulador del evaluador después de

imprimir el componente. No olvide configurar también la propiedad *Master* de la expresión, pues en caso contrario la evaluación no es correcta. En un informe simple, debemos indicar aquí el componente *QuickRep* correspondiente, pero en un informe *master/detail* o que está basado en grupos, hay que indicar el componente *TQRSubDetail* ó *TQRGroup* que pertenece al nivel de anidamiento de la función estática.

Podemos tener problemas al tratar de imprimir campos de tablas que, aunque son accesibles desde el formulario del informe, no están siendo utilizadas explícitamente por alguno de sus componentes. En tal caso, hay que añadir manualmente la tabla a la lista *AllDataSets* del informe, preferiblemente en el evento *BeforePrint* del mismo.

El componente *TQRExprMemo* es una variante de *TQRExpr*. Como su nombre indica, contiene un texto con múltiples líneas en su interior, al igual que *TQRMemo*. Ahora bien, es posible situar dentro del texto expresiones encerradas entre llaves. Por ejemplo:

```
Estimado señor o señora {FirstName + ' ' + LastName}:
Siento decirle que su salario actual es de {Salary}.
```

La única propiedad adicional más o menos interesante es *RemoveBlankLines*, que como su nombre indica, permite eliminar las líneas vacías del texto al imprimir.

## Utilizando grupos

Al generar un listado, podemos imprimir una banda especial cuando cambia el valor de una columna o expresión en una fila de la tabla base. Por ejemplo, si estamos imprimiendo los datos de clientes y el listado está ordenado por la columna del país, podemos imprimir una línea con el nombre del país cada vez que comienza un grupo de clientes de un país determinado. De esta forma, puede eliminarse la columna de la banda de detalles, pues ya se imprime en la cabecera de grupo.

Para crear grupos con QuickReport necesitamos el componente *TQRGroup*. Este realiza el cambio controlando el valor que retorna la expresión indicada en la propiedad *Expression*. La siguiente expresión, por ejemplo, provoca que, en un listado de clientes ordenados alfabéticamente, la banda de cabecera de grupo se imprima cada vez que aparezca un cliente con una letra inicial diferente:

```
COPY(tbClientes.Company, 1, 1)
```

Ya hemos mencionado que el componente *TQRGroup* actúa también como banda de cabecera. Los componentes que se deben imprimir en la cabecera del grupo pueden colocarse directamente sobre el grupo. Para indicar la banda del pie de grupo sigue existiendo una propiedad *FooterBand*. Usted trae una banda con sus propiedades predefinidas, y la asigna en esta propiedad. Entonces QuickReport cambia automáticamente el tipo de la banda a *rbGroupFooter*.

Demostraré el uso de grupos con un ejemplo sencillo: quiero un listado de clientes agrupados por totales de ventas: los que han comprado entre 0 y \$25.000, los que han comprado hasta \$50.000, etc. La base del listado es la siguiente consulta, que se coloca en un componente *TQuery*:

```
select Company, sum(ItemsTotal) Total
from Customer, Orders
where Customer.CustNo = Orders.CustNo
group by Company
order by Total desc
```

Es muy importante que la consulta esté ordenada, en este caso por los totales de ventas, en forma descendente. Colocamos un componente *TQuickRep* en un formulario y asignamos el componente *TQuery* que contiene la consulta anterior a su propiedad *DataSet*. Después, con ayuda del editor del informe, añadimos una banda de detalles y una cabecera de página. En la banda de detalles situamos un par de componentes *QRDBText*, asociados respectivamente a cada una de las columnas de la consulta. En este punto, podemos visualizar el resultado de la impresión haciendo clic con el botón derecho del ratón sobre el componente *QuickRep1* y seleccionando el comando *Preview*.

Ahora traemos un componente *TQRGroup* al informe, y editamos su propiedad *Expression*, para asignarle el siguiente texto:

```
INT(Total / 25000)
```

Esto quiere decir que los clientes que hayan comprado \$160.000 y \$140.000 se imprimirán en grupos diferentes, pues la expresión que hemos suministrado devuelve 6 en el primer caso, y 5 en el segundo. Cuando se inicie un nuevo grupo se imprimirá la banda de cabecera de grupo. Traemos un *TQRExpr* sobre el grupo, y tecleamos la siguiente fórmula en su propiedad *Expression*:

```
'Más de ' + FORMATNUMERIC('0,.00', INT(Total / 25000) * 25000)
```

Cada grupo mostrará entonces el criterio de separación adecuado. Para terminar, añada una banda directamente sobre el formulario. Modifique la propiedad *BandType* a *rbGroupFooter*, y *Name* a *PieDeGrupo*. Seleccione el grupo, *QRGroup1*, y asigne *PieDeGrupo* en su propiedad *FooterBand*. Por último, coloque un *TQRExpr* sobre la nueva banda, con la propiedad *ResetAfterPrint* activa y la siguiente expresión:

```
FORMATNUMERIC('0,.00 €', SUM(Total))
```

De este modo, cuando termine cada grupo se imprimirá el total de todas las compras realizadas por los clientes de ese grupo.

Compañía	Total
<hr/>	
Más de 261.575,00 €	
Sight Diver	261.575,80 €
Total	261.575,80 €
<hr/>	
Más de 193.079,00 €	
VIP Divers Club	193.079,75 €
American SCUBA Supply	183.094,40 €
Total	376.174,15 €

La versión 3 de QuickReport introduce la propiedad *RepeatOnNewPage*, de tipo *Boolean*. Cuando esta propiedad está activa, las cabeceras de grupo se repiten al inicio de las páginas, si es que el grupo ocupa varias páginas.

## Eliminando duplicados

Los grupos de QuickReport, sin embargo, no nos permiten resolver todos los casos de información redundante en un listado. Por ejemplo, tomemos un listado de clientes agrupados por ciudades:

Shangri-La Sports Center	Freeport
Unisco	Freeport
Blue Sports	Giribaldi
Cayman Divers World Unlimited	Grand Cayman
Safari Under the Sea	Grand Cayman

Una forma de mejorar la legibilidad del listado es no repetir el nombre de la ciudad en una línea, si coincide con el nombre de ciudad en la línea anterior:

Shangri-La Sports Center	<b>Freeport</b>
Unisco	
Blue Sports	<b>Giribaldi</b>
Cayman Divers World Unlimited	<b>Grand Cayman</b>
Safari Under the Sea	

Si utilizamos el componente *TQRGroup*, agrupando por ciudad, el nombre de la ciudad no puede aparecer en la misma banda que el nombre de la compañía. Sin embargo, es muy fácil lograr el efecto deseado, manejando el evento *OnPrint* del componente que imprime el nombre de la ciudad. Primero debemos declarar una variable *UltimaCiudad* en la sección **private** del formulario:

```
private
    UltimaCiudad: string;
```

Vamos a inicializar esa variable en el evento *BeforePrint* del informe:

```
procedure TForm2.QuickRep1BeforePrint(  
    Sender: TCustomQuickRep; var PrintReport: Boolean);  
begin  
    UltimaCiudad := '';  
end;
```

Suponiendo que el componente que imprime el nombre de ciudad sea *QRDBText2*, interceptamos su evento *OnPrint*:

```
procedure TForm2.QRExpr2Print(Sender: TObject; var Value: string);  
begin  
    if UltimaCiudad = Value then  
        Value := ''  
    else  
        UltimaCiudad := Value;  
end;
```

## Informes *master/detail*

Existen dos formas de imprimir datos almacenados en tablas que se encuentran en relación *master/detail*. La primera consiste en utilizar una consulta SQL basada en el encuentro natural de las dos tablas, dividiendo el informe en grupos definidos por la clave primaria de la tabla maestra. Por ejemplo, para imprimir un listado de clientes con sus números de pedidos y las fechas de ventas necesitamos la siguiente instrucción:

```
select C.CustNo, C.Company, O.OrderNo, O.SaleDate, O.ItemsTotal  
from Customer C, Orders O  
where C.CustNo = O.CustNo
```

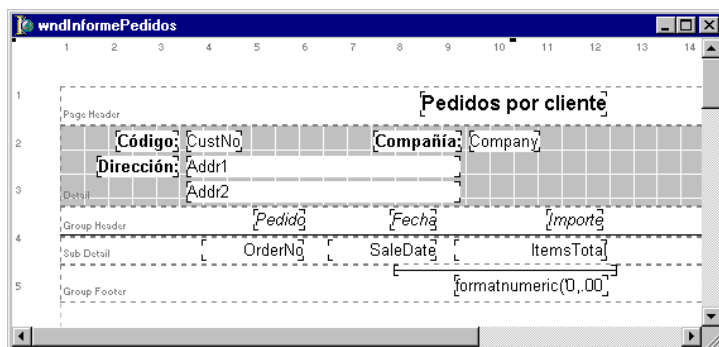
El componente de grupo estaría basado en la columna *CustNo* del resultado. En la cabecera de grupo se imprimirían las columnas *CustNo* y *Company*, mientras que en las filas de detalles se mostrarían *OrderNo*, *SaleDate* y *ItemsTotal*.

La alternativa es emplear el componente *TQRSubDetail* y dejar que QuickReport se ocupe de la gestión de las relaciones entre tablas. Este componente es también una banda, del mismo modo que *TQRGroup*, y sus propiedades principales son:

Propiedad	Significado
<i>Master</i>	Apunta a un <i>TQuickReport</i> o a otro <i>TQRSubDetail</i>
<i>DataSet</i>	El conjunto de datos de detalles
<i>Bands</i>	Permite crear rápidamente la cabecera y el pie
<i>HeaderBand</i>	Una banda de tipo <i>rbGroupHeader</i>
<i>FooterBand</i>	Una banda de tipo <i>rbGroupFooter</i>

Tomemos como ejemplo el informe descrito anteriormente, acerca de clientes y pedidos, y supongamos que esta vez tenemos dos tablas, *tbClientes* y *tbPedidos*, enlazadas en relación *master/detail*. Para crear el informe correspondiente, añada un compo-

nente *TQuickRep* a un formulario vacío y cambie su propiedad *DataSet* a *tbClientes*. Cree ahora una banda de detalles, *rbDetail*, que es donde colocaremos los componentes de impresión correspondientes a la tabla de clientes. Ahora coloque un *TQRSubDetail*, con su propiedad *DataSet* igual a *tbPedidos*. Encima de éste se colocarán los componentes de impresión de los pedidos. Ajuste la propiedad *Master* del enlace de detalles a *QuickRep1*; la propiedad *Master* puede apuntar también a otro componente *TQRSubDetail*, permitiendo la impresión de informes con varios niveles de detalles.



Para crear las bandas de cabecera y pie de grupo, expanda la propiedad *Bands* del *TQRSubDetail*. La clase a la que pertenece *Bands* contiene las propiedades *HasHeader* y *HasFooter*. Basta con asignar *True* a las dos para que se añadan y configuren automáticamente las dos bandas indicadas. De este modo, queda configurado el esqueleto del informe. Solamente queda colocar sobre las bandas los componentes de impresión, de tipo *TQRDBText*, que queremos que aparezcan en el informe.

Por supuesto, es incómodo tener que configurar los componentes de la manera explicada cada vez que necesitamos un informe con varias tablas. Por ese motivo, todas las versiones de QuickReport incluyen una plantilla en la cual ya están incluidas las bandas necesarias para este tipo de informe. Solamente necesitamos añadir sobre las mismas los componentes de impresión correspondientes.

## Informes compuestos

Si necesita imprimir consecutivamente varios informes, *TQRCompositeReport* será su salvación. Este componente ofrece los métodos *Print*, *PrintBackground* y *Preview*, al igual que un informe normal, para imprimir o visualizar el resultado de su ejecución. Pongamos por caso que queremos un listado de una tabla con muchas columnas, de modo que es imposible colocar todas las columnas en una misma línea. Por lo tanto, diseñamos varios informes con subconjuntos de las columnas: en el primer informe se listan las siete primeras columnas, en el segundo las nueve siguientes, etc. A la hora de imprimir estos informes queremos que el usuario de la aplicación utilice un solo comando. Una solución es agrupar los informes individuales en un único informe, y



controlar la impresión desde éste. Podemos traer entonces un componente *QRCompositeReport1* e interceptar su evento *OnAddReports*:

```
procedure TForm1.QRCompositeReport1AddReports(Sender: TObject);
begin
    QRCompositeReport1.Reports.Add(Form2.QuickReport1);
    QRCompositeReport1.Reports.Add(Form3.QuickReport1);
    QRCompositeReport1.Reports.Add(Form4.QuickReport1);
end;
```

Normalmente, los informes se imprimen uno a continuación del otro. La documentación indica que basta con asignar *True* a la propiedad *ForceNewPage* de la banda de título de un informe para que este comience su impresión en una nueva página. Sin embargo, esto no funciona. Lo que sí puede hacerse es crear un manejador para el evento *BeforePrint* del informe:

```
procedure TInforme3.TitleBand1BeforePrint(
    Sender: TQRCustomBand; var PrintBand: Boolean);
begin
    QuickRep1.NewPage;
end;
```

Al parecer, QuickReport ignora la propiedad *ForceNewPage* cuando se debe aplicar en la primera página de un informe.

Cuando se utilizan informes compuestos, todos los informes individuales deben tener el mismo tamaño de papel y la misma orientación.

## Previsualización a la medida

Aunque hasta el momento sólo he mencionado la existencia de *Preview*, en realidad son tres los métodos de vista preliminar de QuickReport:

Método	Resultados
<i>Preview</i>	Vista modal; impresión en el hilo principal del programa.
<i>PreviewModal</i>	Vista modal; impresión en hilo paralelo.
<i>PreviewModeless</i>	Vista no modal; impresión en hilo paralelo.

Me estoy ajustando a la escasa documentación de QuickReport para describir las diferencias entre los tres métodos anteriores. Algunos programadores han encontrado pérdidas de memoria, en algunas versiones, cuando utilizan métodos de vista previa distintos de *PreviewModal*.

El programador puede también crear un diálogo o ventana de previsualización a la medida. La base de esta técnica es el componente *TQRPreview*. El diseño de una ventana de previsualización a la medida comienza por crear un formulario, al cual llamaremos *Visualizador*, e incluir en su interior un componente *TQRPreview*. A este formulario básico pueden añadirse entonces componentes para controlar el grado de acercamiento, el número de página, la impresión, etc. Digamos, para precisar, que

el formulario que contiene el informe se llama *Informe*. Para garantizar la liberación de los recursos asignados al formulario de previsualización, debemos interceptar su evento *OnClose*:

```
procedure TVisualizador.FormClose(Sender: TObject;
    var Action: TCloseAction);
begin
    QRPreview1.QRPrinter := nil;
    Action := caFree;
end;
```

Después hay que regresar al formulario del informe para interceptar el evento *OnPreview* del componente *QuickRep1* de la siguiente manera:

```
procedure TInforme.QuickRep1Preview(Sender: TObject);
begin
    with TVisualizador.Create(nil) do
        try
            QRPreview1.QRPrinter := TQRPrinter(Sender);
            ShowModal;
            // O también: Show, sin destruir el objeto,
            // si queremos una previsualización no modal.
        finally
            Free;
        end;
    end;
end;
```

El parámetro *Sender* del evento apunta al objeto de impresora que se va a utilizar con el informe. Recuerde que QuickReport permite la impresión en paralelo, por lo cual cada informe define un objeto de tipo *TQRPrinter*, diferente del objeto *QRPrinter* global.

Si queremos navegar por las páginas de la vista preliminar, podemos traer botones o comandos de menú al formulario y crear un manejador de eventos como el siguiente:

```
procedure TVisualizador.Navigate(Sender: TObject);
begin
    // Utilizaré Tag para distinguir entre acciones
    case TComponent(Sender).Tag of
        0: QRPreview1.PageNumber := 1;
        1: QRPreview1.PageNumber := QRPreview1.PageNumber - 1;
        2: QRPreview1.PageNumber := QRPreview1.PageNumber + 1;
        3: QRPreview1.PageNumber := QRPreview1.QRPrinter.PageCount;
    end;
end;
```

Para imprimir hace falta el siguiente método:

```
procedure TVisualizador.Print(Sender: TObject);
begin
    QRPreview1.QRPrinter.Print;
end;
```

Existen incontables variaciones sobre este tema. Si usted va a imprimir un documento con un procesador de texto, puede optar por ver una presentación preliminar

y luego imprimir. Pero también puede entrar a saco en el diálogo de impresión, elegir un rango de páginas e imprimir directamente. Para obtener este comportamiento de QuickReport, necesitamos algo parecido a esto:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    Form2.QuickRep1.PrinterSetup;
    if Form2.QuickRep1.Tag = 0 then
        Form2.QuickRep1.Print;
end;
```

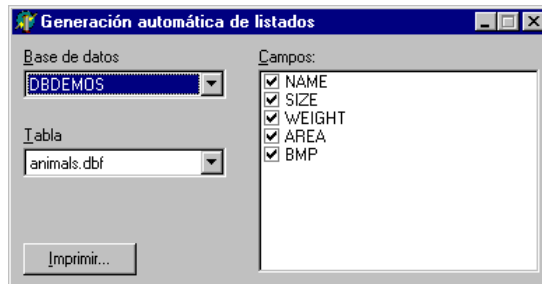
Observe que el método *PrinterSetup* del informe es un procedimiento, no una función. Para saber si ha terminado correctamente, o si el usuario ha cancelado el diálogo, hay que revisar el contenido de la propiedad *Tag* del propio informe; si ésta vale cero, todo ha ido bien. Sinceramente, cuando veo cochinadas como éstas en los productos de mi fabricante de software favorito, que además no están correctamente documentadas, siento vergüenza ajena.

## Listados al vuelo

En la unidad *QRExtra* de QuickReport se ofrecen clases y funciones que permiten generar informes en tiempo de ejecución. Aquí solamente presentaré la técnica más sencilla, basada en el uso de la función *QRCreateList*:

```
procedure QRCreateList(var AReport: TCustomQuickRep;
    AOwner: TComponent; ADataSet: TDataSet; ATitle: string;
    AFieldList: TStrings);
```

En el primer parámetro debemos pasar una variable del tipo *TCustomQuickRep*, el ancestro de *TQuickRep*. Si inicializamos la variable con el puntero *NULL*, la función crea el objeto. Pero si pasamos un objeto creado, éste se aprovecha. *AOwner* corresponde al propietario del informe, *aDataSet* es el conjunto de datos en que se basará el listado, y *aTitle* será el título del mismo. Por último, si pasamos una lista de nombres de campos en *AFieldList*, podremos indicar qué campos deseamos que aparezcan en el listado. En caso contrario, se utilizarán todos. Este será el aspecto de la aplicación que crearemos:



Estos manejadores de eventos son los que obtienen la lista de alias del BDE, las tablas que estos contienen y finalmente sus campos. Debo advertir que he utilizado un objeto persistente *TTable* como ayuda:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    Session.GetAliasNames(ComboBox1.Items);
    ComboBox1.ItemIndex := 0;
    ComboBox1Change(nil);
end;

procedure TForm1.ComboBox1Change(Sender: TObject);
begin
    Session.GetTableNames(ComboBox1.Text, '', True, False,
        ComboBox2.Items);
    ComboBox2.ItemIndex := 0;
    ComboBox2Change(nil);
end;

procedure TForm1.ComboBox2Change(Sender: TObject);
var
    I: Integer;
begin
    CheckListBox1.Clear;
    Table1.DatabaseName := ComboBox1.Text;
    Table1.TableName := ComboBox2.Text;
    Table1.FieldDefs.Update;
    for I := 0 to Table1.FieldDefs.Count - 1 do
    begin
        CheckListBox1.Items.Add(Table1.FieldDefs[I].Name);
        CheckListBox1.Checked[I] := True;
    end;
end;
```

La parte principal de la aplicación es la respuesta al botón de impresión, que mostramos a continuación:

```
procedure TForm1.Button1Click(Sender: TObject);
var
    MyRep: TCustomQuickRep;
    Fields: TStrings;
    I: Integer;
begin
    MyRep := nil;
    Table1.Open;
    try
        Fields := TStringList.Create;
    try
        for I := 0 to CheckListBox1.Items.Count - 1 do
            if CheckListBox1.Checked[I] then
                Fields.Add(CheckListBox1.Items[I]);
        if Fields.Count > 0 then
            begin
                QRCreateList(MyRep, Self, Table1, '', Fields);
            try
                MyRep.Preview;
```

```

        finally
            MyRep.Free;
        end;
    end;
    finally
        Fields.Free;
    end;
    finally
        Table1.Close;
    end;
end;

```

Entre las posibles mejoras que admite el algoritmo están permitir la configuración del tipo de letra, el poder especificar un título (aunque solamente lo utilizará la vista preliminar), y descartar automáticamente los campos gráficos, que esta función no maneja correctamente.

## Filtros de exportación

Casi al final de la página de QuickReport en la Paleta de Componentes, se encuentran tres componentes agrupados como *filtros*. Sirven para guardar un informe generado en diversos formatos, y sus nombres son:

- 1 *TQRTextFilter*: Exportación a texto, columnas de tamaño fijo.
- 2 *TQRCSVFilter*: Formato de campos separados por comas.
- 3 *TQRHTMLFilter*: Exportación a HTML.

¿Cómo se utilizan? Como comprobará, ninguno de ellos tiene eventos o propiedades dignas de mención, excepto el filtro de exportación CSV, que nos permite definir el separador de campos, aunque no el de líneas. La forma más sencilla para utilizar un filtro es dejar caer uno de estos componentes en un formulario de QuickReport. Cuando muestre la vista preliminar del informe, ejecute el comando de guardar el contenido del informe. Comprobará que, en el cuadro de diálogo que selecciona el nombre del fichero que se va a crear, aparece ahora el tipo de fichero asociado al filtro.

También es posible exportar un informe desde el código:

```

class procedure TwndInforme.ExportarHTML(const FileName: string);
var
    Filtro: TQRHTMLDocumentFilter;
begin
    with Create(nil) do
        try
            Filtro := TQRHTMLDocumentFilter.Create(FileName);
            try
                QuickRep1.ExportToFilter(Filtro);
            finally
                FreeAndNil(Filtro);
            end;
        end;
    end;
end;

```

```

    finally
        Free;
    end;
end;

```

Pero, como acaba de ver, hay que utilizar otras clases que están bien escondida dentro de QuickReport. En este caso, hemos recurrido a *TQRHTMLDocumentFilter*, hay también un *TQRCommaSeparatedFilter* y un *TQRAsciiExportFilter*. Tome nota de la falta de uniformidad en los nombres de clase.

## Enviando códigos binarios a una impresora

A veces nuestras principales ventajas se convierten en nuestras limitaciones esenciales. En el caso de QuickReport, se trata del hecho de que toda la impresión se realice a través del controlador para Windows de la impresora que tengamos conectada. Un controlador de impresora en Windows nos permite, básicamente, simular que dibujamos sobre la página activa del dispositivo. Por ejemplo, si tenemos que imprimir un círculo utilizamos la misma rutina de Windows que dibuja un círculo en la pantalla del monitor, y nos evitamos tener que enviar códigos binarios para mover el cabezal de la impresora y manchar cuidadosamente cada píxel de los que conforman la imagen. Y esto es una bendición.

Ahora bien, hay casos en los que nos interesa manejar directamente la impresión. Tal es la situación con determinados informes que requieren impresoras matriciales, o con dispositivos de impresión muy especiales como ciertas impresoras de etiquetas. Aún en el caso en que el fabricante proporcione un controlador para Windows, el rendimiento del mismo será generalmente pobre, al tratar de compaginar el modo gráfico que utiliza Windows con las limitaciones físicas del aparato.

Para estos casos especiales, he decidido incluir en este capítulo una función que permite enviar un *buffer* con códigos binarios directamente a la impresora, haciendo caso omiso de la interfaz de alto nivel del controlador. La función realiza llamadas al API de Windows, a funciones declaradas en la unidad *WinSpool*. Primero definimos la siguiente función auxiliar para transformar valores de retorno de error en nuestras queridas excepciones:

```

procedure PrinterError;
begin
    raise Exception.Create('Error de impresión');
end;

```

No puedo entrar en explicaciones sobre cada una de las funciones utilizadas aquí, pues esto sobrepasa los objetivos del presente libro. He aquí el código:

```

procedure Imprimir(const PrinterName: string;
    Data: Pointer; Count: Cardinal);
type
    TDocInfo1 = record
        DocName, OutputFile, DataType: PChar;

```

```

    end;
var
    hPrinter: THandle;
    DocInfo: TDocInfo;
    Bytes: Cardinal;
begin
    // Llenar estructura con la información sobre el documento
    DocInfo.DocName := 'Documento';
    DocInfo.OutputFile := nil;
    DocInfo.DataType := 'RAW';
    // Obtener un handle de impresora
    if not OpenPrinter(PChar(PrinterName), hPrinter, nil) then
        PrinterError;
    try
        // Informar al spooler del comienzo de impresión
        if StartDocPrinter(hPrinter, 1, @DocInfo) = 0 then
            PrinterError;
        try
            // Iniciar una página
            if not StartPagePrinter(hPrinter) then PrinterError;
            try
                // Enviar los datos directamente
                if not WritePrinter(hPrinter, @Data, Count, Bytes) or
                    (Bytes <> Count) then PrinterError;
            finally
                // Terminar la página
                if not EndPagePrinter(hPrinter) then PrinterError;
            end;
        finally
            // Informar al spooler del fin de impresión
            if not EndDocPrinter(hPrinter) then PrinterError;
        end;
    finally
        // Devolver el handle de impresora
        ClosePrinter(hPrinter);
    end;
end;
end;

```

Esta otra versión de *Imprimir* ha sido adaptada para que imprima una secuencia de caracteres y códigos contenidos en una cadena en la impresora:

```

procedure Imprimir(const PrinterName, Data: string);
begin
    Imprimir(PrinterName, PChar(Data), Length(Data));
end;

```

Por ejemplo:

```

Imprimir('Priscilla', 'Hola, mundo...'#13#10 +
    '...adiós, mundo cruel'#12);

```

Puede también generar un documento a imprimir enviando primero los códigos necesarios a un objeto *TMemoryStream*, y pasando posteriormente su propiedad *Memory* a la función *Imprimir*.





## Gráficos

ENTRE LOS COMPONENTES INTRODUCIDOS por la versión 3 de la VCL, destacan los que se encuentran la página *Decision Cube*, que nos permiten calcular y mostrar gráficos y rejillas de decisión. Con los componentes de esta página podemos visualizar en pantalla informes al estilo de *tablas cruzadas (crosstabs)*. En este tipo de informes se muestran estadísticas acerca de ciertos datos: totales, cantidades y promedios, con la particularidad de que el criterio de agregación puede ser multidimensional. Por ejemplo, nos interesa saber los totales de ventas por delegaciones, pero a la vez queremos subdividir estos totales por meses o trimestres. Además, queremos ocultar o expandir dinámicamente las dimensiones de análisis, que los resultados se muestren en rejillas, o en gráficos de barras. Todo esto es tarea de *Decision Cube*. Y ya que estamos hablando de gráficos, explicaremos también como aprovechar los componentes *TChart* y *TDBChart*, que permiten mostrar series de datos generadas mediante código o provenientes de tablas y consultas.

Para ilustrar el empleo de *Decision Cube* y del componente *TDBChart* utilizaremos una base de datos que he incluido en el CD-ROM, extraída de una vieja aplicación que gestionaba libretas de ahorro. Para mayor comodidad, he incluido versiones en formato *Paradox* y en *InterBase*.

### Gráficos y biorritmos

La civilización occidental tiende a infravalorar la importancia de los ritmos en nuestra vida. Existen, sin embargo, teorías cognitivas que asocian la génesis de la conciencia con asociaciones rítmicas efectuadas por nuestros antepasados. Muchos mitos que perviven entre nosotros reconocen de un modo u otro este vínculo. Y uno de los mitos modernos relacionados con los ritmos es la “teoría” de los biorritmos: la suposición de que nuestro estado físico, emocional e intelectual es afectado por ciclos de 23, 28 y 33 días respectivamente. Se conjetura que estos ciclos arrancan a partir de la fecha de nacimiento de la persona, de modo que para saber el estado de los mismos para un día determinado basta con calcular el total de días transcurridos desde entonces y realizar la división entera con resto. Al resultado, después de una transformación lineal sencilla, se le aplica la función seno y, ¡ya hay pronóstico meteorológico!

Mi interés es mostrar solamente cómo puede utilizarse el componente *TChart* para este propósito. Este componente se encuentra en la página *Additional* de la Paleta de Componentes. Pero si exploramos un poco la Paleta, encontraremos también los componentes *TDBChart* y *TQRDBChart*. En principio, *TChart* y *TDBChart* tienen casi la misma funcionalidad, pero el segundo puede ser llenado a partir de un conjunto de datos, especificando determinados campos del mismo. El conjunto de datos puede ser indistintamente una tabla, una consulta o el componente derivado de *TDataSet* que se le ocurra. Por su parte, *TQRDBChart* puede incluirse dentro de un informe generado con QuickReport.

Un gráfico debe mostrar valores de una colección de datos simples o puntuales. La colección de datos de un gráfico de tarta, por ejemplo, debe contener pares del tipo:

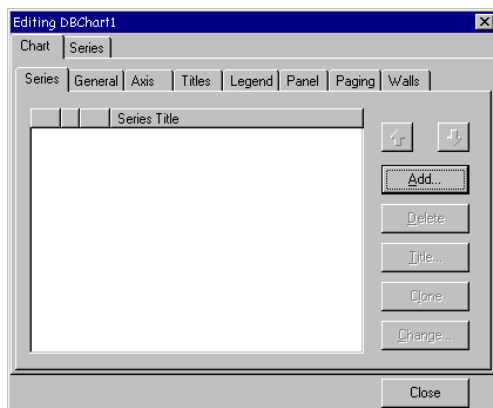
(etiqueta, proporción)

La proporción determina el ángulo que ocupa cada trozo de la tarta, y la etiqueta sirve ... pues para eso ... para etiquetar el trozo. En cambio, un gráfico lineal puede contener tripletas en vez de pares:

(etiqueta, valor X, valor Y)

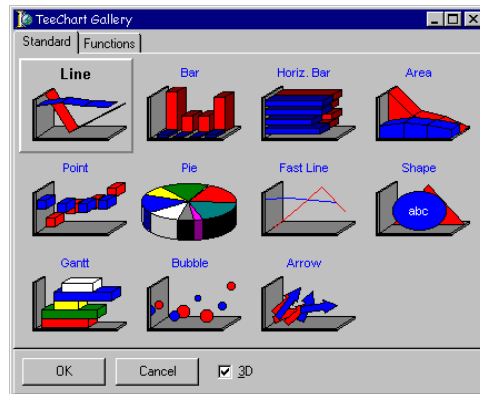
Ahora se ha incluido un valor X para calcular distancias en el eje horizontal. Si un gráfico lineal contiene tres puntos, uno para enero, uno para febrero y otro para diciembre, se espera que los puntos de enero y febrero estén más alejados del punto correspondiente a diciembre.

A estas colecciones de puntos se les denomina *series*. Un componente *TChart* contiene una o más series, que son objetos derivados de la clase *TSeries*. Esto quiere decir que puede mostrar simultáneamente más de un gráfico, en el sentido usual de la palabra, lo cual puede valer para realizar comparaciones. He dicho antes que los tipos de series concretas se derivan de la clase abstracta *TSeries*, y es que la estructura de una serie correspondiente a un gráfico lineal es diferente a la de una serie que contenga los datos de un gráfico de tarta.



Las series deben ser creadas por el programador, casi siempre en tiempo de diseño. Para ilustrar la creación, traiga a un formulario vacío un componente *TChart*, y pulse dos veces sobre el mismo, para invocar a su editor predefinido. Debe aparecer el cuadro de diálogo anterior.

A continuación pulse el botón *Add*, para que aparezca la galería de estilos disponible. Podemos optar por series simples, o por funciones, que se basan en datos de otras series para crear series calculadas. Escogeremos un gráfico lineal, por simplicidad y conveniencia.



Cuando creamos una serie para un gráfico, estamos creando explícitamente un componente con nombre y una variable asociada dentro del formulario. Por omisión, la serie creada se llamará *Series1*. Repita dos veces más la operación anterior para tener también las series *Series2* y *Series3*. Las tres variables apuntan a objetos de la clase *TLineSeries*, que contiene el siguiente método:

```
function Add(const AValue: Double; const ALabel: string = '';
  AColor: TColor = clTeeColor): Integer;
```

Utilizaremos este método, en vez de *AddXY*, porque nuestros puntos irán espaciados uniformemente. En el tercer parámetro podemos utilizar la constante especial de color *clTeeColor*, para que el componente decida qué color utilizar.

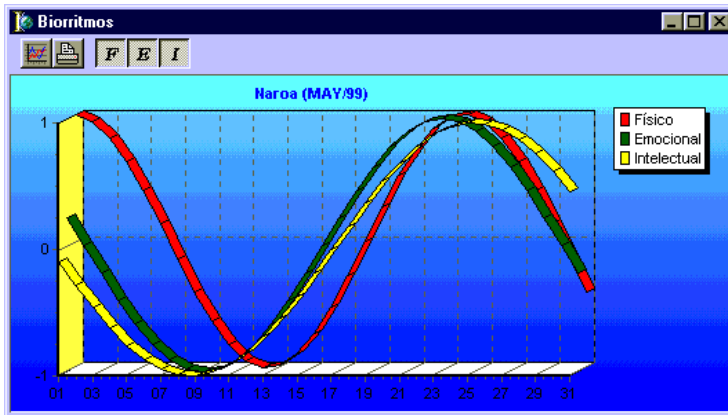
Ahora hay que añadir al programa algún mecanismo para que el usuario pueda teclear una fecha de nacimiento, decir a partir de qué fecha desea el gráfico de los biorritmos, y el número de meses que debe abarcar. Todo esto lo dejo en sus manos expertas. Me limitaré a mostrar la función que crea el gráfico a partir de estos datos:

```
procedure TwndMain.FillChart(const Nombre: string;
  Nacimiento, PrimerDia: TDateTime; Meses: Integer);
var
  Pi2: Double;
  UltimoDia: TDateTime;
  I, Dias: Integer;
  MS1, MS2, Etiqueta: string;
```

```

begin
    // Calcular el último día
    UltimoDia := IncMonths(PrimerDia, Meses) - 1;
    // El título del gráfico
    MS1 := AnsiUpperCase(FormatDateTime('mmm/yy', PrimerDia));
    MS2 := AnsiUpperCase(FormatDateTime('mmm/yy', UltimoDia));
    if MS1 <> MS2 then
        MS1 := MS1 + '-' + MS2;
    if Nombre = '' then
        Nombre := 'BIORRITMOS';
    Chart1.Title.Text.Text := Nombre + ' (' + MS1 + ')';
    // Los puntos del gráfico
    Series1.Clear;
    Series2.Clear;
    Series3.Clear;
    Pi2 := 2 * PI;
    for I := 1 to Round(UltimoDia - PrimerDia) + 1 do
    begin
        Dias := Round(PrimerDia - Nacimiento);
        Etiqueta := FormatDateTime('dd', PrimerDia);
        Series1.Add(Sin((Dias mod 23) * Pi2 / 23),
            Etiqueta, clTeeColor);
        Series2.Add(Sin((Dias mod 28) * Pi2 / 28),
            Etiqueta, clTeeColor);
        Series3.Add(Sin((Dias mod 33) * Pi2 / 33),
            Etiqueta, clTeeColor);
        PrimerDia := PrimerDia + 1;
    end;
end;
end;

```



Como el lector puede observar, he incluido tres botones para que el usuario pueda “apagar” selectivamente la visualización de alguna de las series, mediante la propiedad *Active* de las mismas. La propiedad *Tag* de los tres botones han sido inicializadas a 0, 1 y 2 respectivamente. De este modo, la respuesta al evento *OnClick* puede ser compartida, aprovechando la presencia de la propiedad vectorial *Series* en el gráfico:

```

procedure TwndMain.SwitchSeries(Sender: TObject);
begin
    with Sender as TToolButton do
        Chart1.Series[Tag].Active := Down;
    end;
end;

```

Vuelvo a advertirle: nunca utilice los biorritmos para intentar predecir su futuro, pues es una verdadera superstición. No es lo mismo, por ejemplo, que mirar en una bola de cristal o echar el tarot. Si el lector lo desea, puede llamarme al 906-999-999, y por una módica suma le diré lo que le tiene reservado el destino.

## El componente *TDBChart*

Abra la aplicación de las libretas de banco, y añadimos una nueva ventana a la misma. Mediante el comando de menú *File | Use unit* hacemos que utilice la unidad del módulo de datos. Después colocamos sobre la misma un componente *TPageControl* con tres páginas:

- 1 Evolución del saldo.
- 2 Rejilla de análisis.
- 3 Gráfico de análisis.

En la primera página situaremos un gráfico de línea para mostrar la curva del saldo respecto al tiempo, con el propósito de deprimir al usuario de la aplicación. En la segunda y tercera página desglosaremos los ingresos y extracciones con respecto al concepto de la operación y el mes en que se realizó; en una de ellas utilizaremos una rejilla, mientras que en la segunda mostraremos un gráfico de barras basado en dos series.

Comenzaremos con la curva del saldo. Necesitamos saber qué saldo teníamos en cada fecha; la serie de pares fecha/saldo debe estar ordenada en orden ascendente de las fechas. Aunque podemos usar una tabla ordenada mediante un índice, para mayor generalidad utilizaremos una consulta. Creamos entonces un módulo de datos para la aplicación, y le añadimos un componente *TQuery*. Dentro de su propiedad SQL tecleamos la siguiente instrucción:

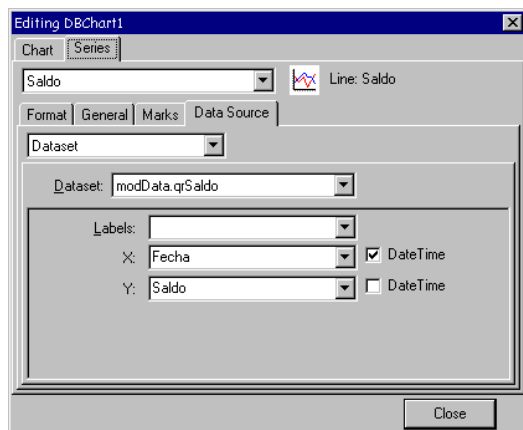
```
select Fecha, Saldo
from Apuntes
where Libreta = :Codigo
order by Fecha
```

La cláusula **where** especifica que solamente nos interesan los apuntes correspondientes a la libreta activa en la aplicación, por lo cual debemos asignar la fuente de datos asociada a la tabla de libretas, *dsLib*, en la propiedad *DataSource* de la consulta. Recuerde que en este caso no debe asignarse un tipo al parámetro *Codigo*, pues corresponde a la columna homónima de la tabla de libretas.

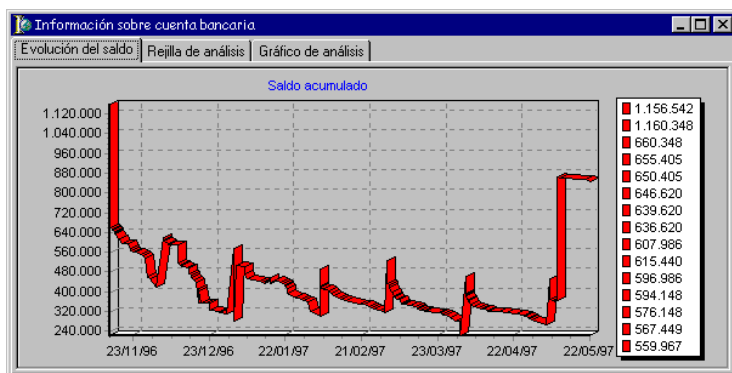
Ahora vamos a la primera página del formulario principal, seleccionamos la página *Data Controls* de la Paleta de Componentes, y traemos un *TDBChart*, cambiándole su alineación *Align* a *alClient*. ¿Por qué *TDBChart*? Porque en este caso, los datos pueden obtenerse directamente de la base de datos.

El próximo paso es crear una serie lineal. Recuerde que basta con realizar un doble clic sobre el componente *TDBChart*. Pulsando después el botón *Add* podemos seleccionar un tipo de serie para añadir: elija nuevamente una serie lineal. Para colocar más gráficos en el mismo formulario y evitar un conflicto que el autor de *TChart* podía haber previsto, cambiaremos el nombre del componente a *Saldos*. Podemos hacerlo seleccionando el componente *Series1* en el Inspector de Objetos, y modificando la propiedad *Name*.

La fuente de datos de la serie se configura esta vez en la página *Series | Data Source*. Tenemos que indicar que alimentaremos a la serie desde un conjunto de datos (en el combo superior), el nombre del conjunto de datos (la consulta que hemos definido antes), y qué columnas elegiremos para los ejes X e Y. En este caso, a diferencia de lo que sucedió con los biorritmos, nos interesa espaciar proporcionalmente los valores del eje horizontal, de acuerdo a la fecha de la operación:



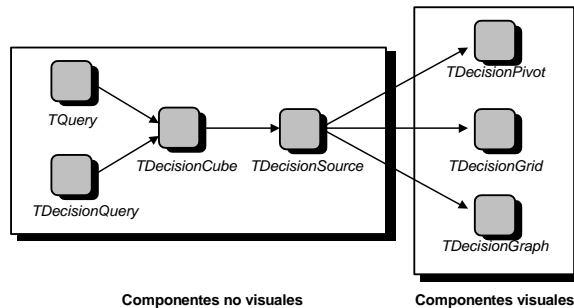
A continuación, podemos configurar detalles del gráfico, como el color del fondo, el título, etc. Puede también intentar añadir una serie que represente el saldo promedio, utilizando la página *Functions* de la galería de series. El gráfico final puede apreciarse en la siguiente figura:



## Componentes no visuales de *Decision Cube*

Para preparar los gráficos y rejillas que utilizaremos con *Decision Cube*, necesitamos unos cuantos componentes no visuales, que colocaremos en el módulo de datos. El origen de los datos a visualizar puede ser indistintamente un componente *TQuery* o un *TDecisionQuery*; la diferencia entre ambos es la presencia de un editor de componente en *TDecisionQuery* para ayudar en la generación de la consulta. La consulta se conecta a un componente *TDecisionCube*, que es el que agrupa la información en forma matricial para su uso posterior por otros componentes. El cubo de decisión se conecta a su vez a *TDecisionSource*, que sirve como fuente de datos a los componentes visuales finales: *TDecisionGrid*, para mostrar los datos en formato de rejilla, *TDecisionGraph*, un derivado de los gráficos TeeChart, y *TDecisionPivot*, para manejar dinámicamente las dimensiones del gráfico y la rejilla.

El siguiente diagrama muestra cómo se conectan los distintos componentes de esta página:



Utilizaremos, para simplificar la exposición, una consulta común y corriente como el conjunto de datos que alimenta toda la conexión anterior; la situaremos, como es lógico, en el módulo de datos de la aplicación. Este es el contenido de la instrucción SQL que debemos teclear dentro de un componente *TQuery*:

```

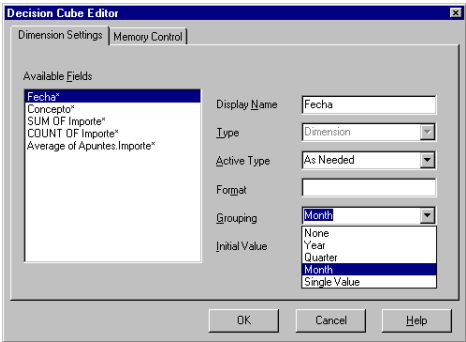
select A.Fecha, C.Concepto, sum(A.Importe), count(A.Importe)
from Apuntes A, Conceptos C
where A.Concepto = C.Codigo and
        A.Libreta = :Codigo
group by A.Fecha, C.Concepto
  
```

Nuevamente, hemos restringido el resultado a los apuntes correspondientes a la libreta actual. Recuerde asignar *dsLib* a la propiedad *DataSource* de esta consulta.

Queremos mostrar la suma de los importes de las operaciones, el promedio y la cantidad de las mismas, desglosadas según la fecha y el concepto; del concepto queremos la descripción literal, no su código. Para cada dimensión, se incluye directamente en la consulta la columna correspondiente, y se menciona dentro de una cláusula **group by**. Por su parte, los datos a mostrar se asocian a expresiones que hacen uso

de funciones de conjuntos. En el ejemplo hemos incluido la suma y la cantidad del importe; no es necesario incluir el promedio mediante la función **avg**, pues Decision Cube puede deducirlo a partir de los otros dos valores.

Una vez que está configurada la consulta, traemos un componente *TDecisionCube*, lo conectamos a la consulta anterior mediante su propiedad *DataSet*, y realizamos un doble clic sobre el componente para su configuración. Necesitamos cambiar el título de las dimensiones y de las estadísticas, cambiando títulos como “SUM OF Importe” a “Importe total”. Es muy importante configurar las dimensiones de tipo fecha, indicando el criterio de agrupamiento. En este ejemplo hemos cambiando la propiedad *Grouping* de las fechas de los apuntes a *Month*, para agrupar por meses; también pueden agruparse por días, años y trimestres. Para terminar, colocamos un componente *TDecisionSource* dentro del módulo de datos, y asignamos *DecisionCube1* a su propiedad *DecisionCube*.



Rejillas y gráficos de decisión

Es el momento de configurar los componentes visuales. Vamos a la segunda página del formulario principal, y añadimos un componente *TDecisionGrid* en la misma, cambiando su alineación a *alClient*, para que ocupe toda el área interior de la página. Este es el aspecto de la rejilla:

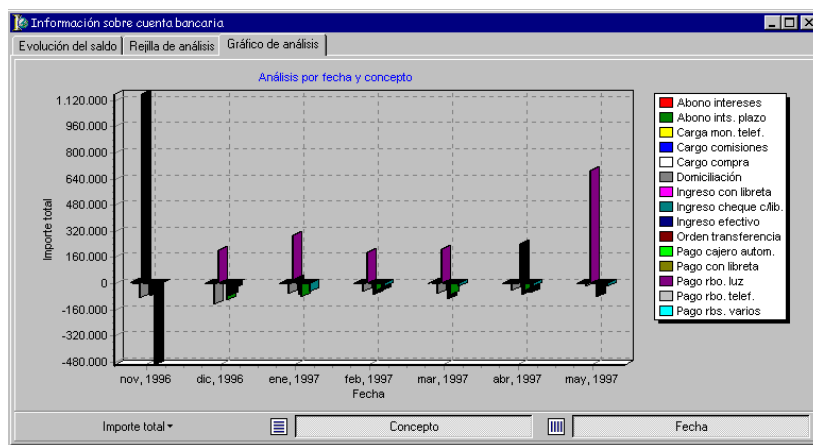
Información sobre cuenta bancaria								
Evolución del saldo    Rejilla de análisis    Gráfico de análisis								
Fecha	nov. 1996	dic. 1996	ene. 1997	feb. 1997	mar. 1997	abr. 1997	may. 1997	
Concepto								
Abono intereses	3.806,00 Pts	2.532,00 Pts						
Abono int. plazo					3.750,00 Pts		1.625,00 Pts	
Carga mon. telef.					-5.000,00 Pts	-2.000,00 Pts		
Cargo comisiones			-50,00 Pts					
Cargo compra	-32.689,00 Pts	-130.943,00 Pts	-63.969,00 Pts	-53.476,00 Pts	-64.453,00 Pts	-43.095,00 Pts	-17.030,00 Pts	
Domiciliación	1.156.542,00 Pts							
Ingreso con libreta		202.400,00 Pts	295.000,00 Pts	190.000,00 Pts	210.000,00 Pts		690.000,00 Pts	
Ingreso cheque c/ib			25.000,00 Pts					
Ingreso efectivo						240.000,00 Pts		
Orden transferencia	-75.700,00 Pts		-68.000,00 Pts	-68.000,00 Pts	-95.000,00 Pts	-68.000,00 Pts	-86.000,00 Pts	
Pago cajero autom.	-39.000,00 Pts	-104.000,00 Pts	-87.000,00 Pts	-43.000,00 Pts	-74.000,00 Pts	-47.000,00 Pts	-8.000,00 Pts	
Pago con libreta	-500.000,00 Pts	-75.000,00 Pts						
Pago rbo. luz		-5.067,00 Pts						
Pago rbo. telef.		-32.986,00 Pts		-43.571,00 Pts		-58.425,00 Pts		
Pago rbs. varios			-49.496,00 Pts	-18.363,00 Pts	-18.363,00 Pts	-18.363,00 Pts	-18.363,00 Pts	
Sum	452.959,00 Pts	-143.064,00 Pts	51.475,00 Pts	-36.410,00 Pts	-43.066,00 Pts	3.117,00 Pts	562.232,00 Pts	



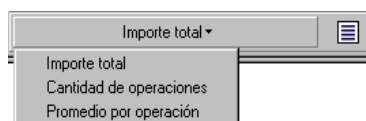
Como se observa en la imagen, en la dimensión horizontal aparece la fecha, mientras que la vertical corresponde a los distintos tipos de operaciones. Utilizando directamente la rejilla podemos intercambiar las dimensiones, mezclarlas, ocultarlas y restaurarlas. Mostramos a continuación, como ejemplo, el resultado de contraer la dimensión de la fecha; aparecen los gastos totales agrupados por conceptos:

Información sobre cuenta bancaria							
Evolución del saldo		Rejilla de análisis		Gráfico de análisis			
Concepto							
Abono intereses	Abono ints. plazo	Carga mon. telef.	Cargo comisiones	Cargo compra	Domiciliación	Ingreso con libreta	Ingreso cheque
6.338,00 Pts	5.375,00 Pts	-7.000,00 Pts	-50,00 Pts	-465.675,00 Pts	1.156.542,00 Pts	1.587.400,00 Pts	25.000,00 Pts

Sin embargo, es más sencillo utilizar el componente *TDecisionPivot* para manejar las dimensiones, y es lo que haremos en la siguiente página, añadiendo un *TDecisionGraph* y un pivote; el primero se alinearán con *alClient* y el segundo con *alBottom*. Realmente, es necesario incluir pivotes para visualizar los gráficos de decisión, pues de otro modo no pueden modificarse dinámicamente las dimensiones de análisis. La siguiente figura muestra la tercera página de la aplicación en funcionamiento:

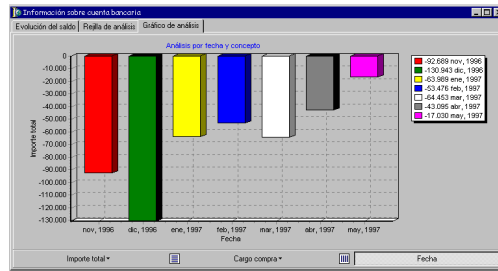


Mediante el pivote se puede cambiar el tipo de estadística que se muestra en el gráfico; para esto se utiliza el botón de la izquierda, que al ser pulsado muestra las funciones previstas durante el diseño de la consulta.



Otra operación importante consiste en ocultar dimensiones o en realizar la operación conocida como *drill in* (aproximadamente, una perforación). Hemos visto antes el resultado de ocultar la dimensión de la fecha: quedan entonces los totales, con independencia de esta dimensión. Ahora bien, al efectuar un *drill in* podemos mostrar

exclusivamente los datos correspondientes a un valor determinado de la dimensión afectada. El *drill in* se activa pulsando el botón derecho del ratón sobre el botón de la dimensión que queremos modificar. La figura siguiente muestra exclusivamente el importe dedicado a *Cargo de compras*. Las barras están invertidas, pues los valores mostrados son negativos por corresponder a extracciones.



Por último, es conveniente saber que los gráficos de decisión cuentan con una amplia variedad de métodos de impresión. De esta forma, podemos dejar que el usuario de nuestra aplicación elija la forma de visualizar los datos que le interesan y, seleccionando un comando de menú, pueda disponer en papel de la misma información que se le presenta en la pantalla del ordenador.

Por ejemplo, el siguiente método sirve para imprimir, de forma independiente, un gráfico de decisión, con la orientación apaisada:

```
procedure TForm1.miImprimirClick(Sender: TObject);
begin
    DecisionGraph1.PrintOrientation(poLandscape);
end;
```

La constante *poLandscape* está definida en la unidad *Printers*, que debe ser incluida explícitamente en la unidad.

## Uso y abuso de Decision Cube

El lector se habrá dado cuenta de que llevamos unas cuantas secciones sin escribir sino una sola línea de código. En realidad, la configuración de un cubo de decisión es una tarea muy sencilla. Pero esta misma sencillez lleva al programador a utilizar equivocadamente estos componentes. No voy a decirle cuáles son los usos “correctos” de Decision Cube, pues no me gustaría poner límites a su imaginación, pero sí voy a ilustrar algunos ejemplos realmente desacertados.

Comencemos por la interfaz visual. Hemos visto que *TDecisionGraph* admite la visualización de al menos dos dimensiones de forma simultánea. Si se quiere mostrar sólo una dimensión, obtendremos una serie (en el sentido que esta palabra tiene para *TChart* y *TDBChart*); si activamos dos dimensiones a la vez, se dibujará una sucesión de series. Y aquí tenemos el primer problema: a no ser que la segunda dimensión

contenga un conjunto muy pequeño de valores posibles, el gráfico resultante será un galimatías, pues el usuario verá muchas series apiladas una sobre otras. Reto a cualquiera a que saque algo en claro de este tipo de gráficos.

Es posible configurar una dimensión para que contenga el valor *binSet* en su forma de agrupamiento, aunque no es posible cambiar este valor en tiempo de diseño. En la próxima sección mostraremos cómo hacerlo en tiempo de ejecución. Cuando este valor está activo, la dimensión siempre estará en el estado *drilled in*, y la cantidad de memoria asignada al cubo será mucho menor. Desgraciadamente, los componentes visuales, como *TDecisionPivot*, tienen problemas al enfrentarse a este tipo de dimensiones, y la misma funcionalidad puede obtenerse mediante consultas paramétricas.

Más problemas con la cardinalidad de las dimensiones: he visto un programa que intentaba usar como dimensión nada menos que un nombre de cliente<sup>41</sup>. ¿Qué tiene de malo? En primer lugar, que se generan demasiadas celdas para que el Cubo pueda manejarlas. Aunque Borland no ha hecho público el código fuente de Decision Cube con los detalles de implementación, sabemos que este componente utiliza una estructura de datos en memoria cuyo tamaño es proporcional a la multiplicación de el número de valores por cada dimensión. Lo más probable es que el Decision Cube “corte” la dimensión en algún sitio y solamente muestre un pequeño número de clientes. Precisamente esto era lo que pasaba con el ejemplo que mencioné al comenzar el párrafo.

En segundo lugar, ¿qué información útil, qué tendencia podemos descubrir en un cubo que una de sus dimensiones sea el nombre o código de cliente? Supongamos incluso que utilizamos esa única dimensión. ¿Cuál sería el resultado que mostraría un *TDecisionGrid*? Uno similar al de un *TDBGGrid*, y está claro que la rejilla de datos de toda la vida lo haría mejor. ¿Y sobre un *TDecisionGraph*? Tendríamos una serie que, a simple vista, semejaría una función discontinua y caótica; la información gráfica no tendría valor alguno. Y en cualquier caso, un sencillo *TDBChart* sería mucho más útil y eficiente.

## Modificando el mapa de dimensiones

Las posibilidades de configuración dinámica de un cubo de decisión van incluso más allá de las que ofrece directamente el componente de pivote. Por ejemplo, puede interesarnos el cambiar el agrupamiento de una dimensión de tipo fecha. En vez de agrupar los valores por meses, queremos que el usuario decida si las tendencias que analiza se revelan mejor agrupando por trimestres o por años.

El siguiente procedimiento muestra como cambiar el tipo de agrupamiento para determinado elemento de un cubo de decisión, dada la posición del elemento:

```
procedure ModificarCubo(ACube: TDecisionCube; ItemNo: Integer;
    Grouping: TBinType);
```

---

<sup>41</sup> En esta discusión asumiré que se trata de una tabla de clientes grande, que es lo típico.

```

var
  DM: TCubeDims;
begin
  DM := TCubeDims.Create(nil, TCubeDim);
  try
    DM.Assign(ACube.DimensionMap);
    DM.Items[ItemNo].BinType := Grouping;
    ACube.Refresh(DM, False);
  finally
    DM.Free;
  end;
end;

```

El parámetro *ItemNo* indica la posición de la dimensión dentro del cubo, mientras que *Grouping* es el tipo de agrupamiento que vamos a activar: *binMonth*, *binQuarter* ó *binYear*. Estas constantes están definidas en la unidad *MxCommon*. También puede utilizar el valor *binSet*, para que la propiedad aparezca en estado *drilled in*, siempre que también asignemos algún valor a la propiedad *StartValue* o *StartDate* de la dimensión.

El programador puede incluir algún control para que el usuario cambie el agrupamiento, por ejemplo, un combo, e interceptar el evento *OnChange* del mismo para aprovechar y ejecutar el procedimiento antes mostrado.

Esta técnica a veces provoca una excepción. El fallo puede reproducirse en tiempo de diseño fácilmente: basta con crear un formulario basado en Decision Cube e intentar cerrar el conjunto de datos de origen. Después de mucho experimentar con estos controles, he notado que la probabilidad de fallo disminuye si no existe un *TDecisionGraph* a la vista en el momento del cambio de dimensiones.

## Transmitiendo un gráfico por Internet

Para terminar, quiero mostrar cómo se puede generar un gráfico con TeeChart dentro de una aplicación CGI/ISAPI, para enviarlo a través de Internet. He llamado *Modigliani* al programa, porque al igual que el conocido pintor, dibuja óvalos y pretende que son gráficos de tarta (*pie charts*). En realidad, el pintor pretendía que eran rostros femeninos, pero se trata de un detalle sin importancia.

El núcleo de la técnica está en la función global *GetBitmap*. La función debe recibir un control, y copiar su imagen en un mapa de bits creado al vuelo:

```

function GetBitmap(WinCtrl: TWinControl;
  AWidth, AHeight: Integer): TBitmap;
begin
  Result := TBitmap.Create;
  try
    Result.Width := AWidth;
    Result.Height := AHeight;
    Result.Canvas.Lock;
    try
      WinCtrl.PaintTo(Result.Canvas.Handle, 0, 0);
    finally
      Result.Canvas.Unlock;
    end;
  end;
end;

```

```

    except
        Result.Free;
        raise;
    end;
end;

```

Note que estamos pasando el ancho y altura del mapa de bits que debe crear como resultado. Es cierto que el propio control podría indicar utilizarse para extraer las dimensiones del gráfico final, pero he preferido dejar abierta la posibilidad de mostrar solamente una parte del área del control.

Tenemos que crear el control que vamos a utilizar. En este ejemplo, queremos que sea un *TDBChart*, lo que implica que necesitaremos también un conjunto de datos; preferiblemente una consulta. Supondremos que la consulta y la conexión se han añadido en tiempo de diseño al módulo de datos Web, y que el nombre del componente de consulta es *TheQuery*. Estas son las declaraciones que hay que añadir a la clase del módulo:

```

type
    TmodChart = class(TWebModule)
        // ...
    private
        FForm: TForm;
        FChart: TDBChart;
        FPie: TPieSeries;
        procedure CreateGraph(const Labels, Values: string;
                               AWidth, AHeight: Integer; ShowLegend: Boolean);
        procedure DestroyGraph;
        // ...
    end;

```

Esta es la implementación de *CreateGraph*:

```

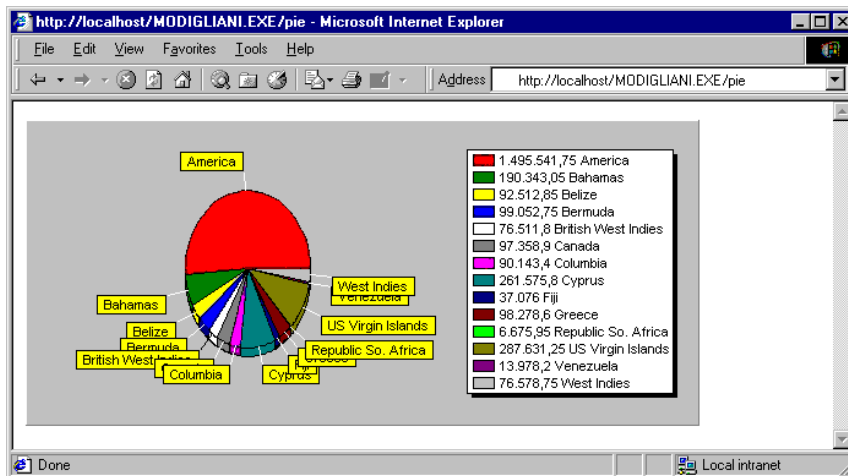
procedure TmodChart.CreateGraph(const Labels, Values: string;
    AWidth, AHeight: Integer; ShowLegend: Boolean);
begin
    FForm := TForm.Create(nil);
    try
        FChart := TDBChart.Create(FForm);
        FChart.Parent := FForm;
        FChart.Height := AHeight;
        FChart.Width := AWidth;
        FChart.Legend.Visible := ALegend;
        FPie := TPieSeries.Create(FForm);
        FPie.ParentChart := FChart;
        FPie.DataSource := TheQuery;
        FPie.XLabelsSource := Labels;
        FPie.PieValues.ValueSource := Values;
        FPie.Active := True;
    except
        DestroyGraph;
        raise;
    end;
end;

```

¡He ahí el truco real! No podemos poner un control sobre un módulo de datos en tiempo de diseño. Por lo tanto, tenemos que crear un formulario “al vuelo” para luego crear el gráfico en su interior. Todos los objetos que se acaban de crear deben ser destruidos por el método *DestroyGraph*. Nos bastará con destruir el formulario, porque es el propietario del gráfico y de la serie:

```
procedure TmodChart.DestroyGraph;
begin
    FreeAndNil (FForm);
    FChart := nil;
    FPie := nil;
end;
```

No voy a incluir aquí todos los detalles: la aplicación del CD recibe una petición que contiene parámetros para crear una consulta SQL, y a partir de ella generar el mapa de bits del gráfico. Ese mapa se comprime en formato JPEG y se envía como respuesta HTML, de vuelta al navegador. Este es el aspecto de una página creada por *Modigliani*:



Puede ensayar la técnica también con los componentes de Decision Cube.

# ANTICLIMAX

*“Cuando el sabio señala a la Luna...”*

EL ÚLTIMO LIBRO QUE ESCRIBÍ TERMINABA con una cita erudita del Maestro Lao Tse. Está muy de moda, o estuvo en cierto tiempo, echar mano de refranes chinos para justificar cualquier cosa. Por lo general, mientras más confusos y ambiguos, mejor que mejor, porque así eran más manejables. Ya he confesado haber cometido este pecado; aprovecharé esta nueva oportunidad para redimirme.

... cuando el sabio señala a la Luna, el idiota se empeña en ver solamente el objetivo: la Luna. Pero lo más probable es que el señor sabio esté intentando que su discípulo aprenda a señalar. En la mayoría de los casos que conozco, son más importantes los medios que el fin. En buen español, lo importante no es lo que pienso sobre Delphi y la programación (para eso están los manuales), sino los métodos que utilizo para llegar a esas conclusiones, sean certeras o erradas. Da lo mismo si sostengo que el InterBase es bueno y el SQL Server es malo, o todo lo contrario; sin embargo, hay personas que me han recriminado por alguna opinión.

O dicho de otra forma: me gustaría que todos desconfiaran como yo de los argumentos basados en la autoridad de alguien. Quien te promete un futuro, te está escamoteando el presente. Y aunque alguien se empeñó en señalar durante años a la Luna y esconder el dedo de las miradas indiscretas, mientras los tontos aplaudían como las ratas del cuento del flautista, nunca pudo ese señor subir más allá de las hermosas montañas del Tibet. Y todavía puede caer rodando de ellas.





# INDICE ALFABETICO

---

## A

Abrazo mortal · 238, 265  
AbsolutePage · 664  
abstract · 33  
Acciones  
    predefinidas · 461, 730  
    referenciales · 190  
    TActionManager · 358  
    WebSnap · 914  
ACID · 255  
Actions  
    TAdapter · 912  
ActionValue  
    TAdapterField · 916, 919  
ActivateWebModule · 846  
ActiveForms · 891  
ActiveX  
    libraries · 94, 682  
Actualizaciones en caché · 591, 632  
Actualizaciones en lote · 667  
Adaptadores · 903  
ADO  
    biblioteca de tipos · 646  
    cadenas de conexión · 643  
    clases · 640  
    conexiones · 641  
    conjuntos de registros · 648  
    control de errores · 657  
    propiedades dinámicas · 653  
    transacciones · 656  
ADO Express · 735  
Advise · 146  
AfterApplyUpdates · 734  
AfterClose · 711  
AfterDispatch · 850  
AfterGetRecords · 693, 699  
AfterInsert · 484  
AfterOpen · 711  
AfterPost · 481  
AfterPrint · 976  
AfterRowRequest · 716  
AfterUpdateRecord · 736  
AggregatesActive · 447  
Alias · 562  
    local · 562  
    persistente · 562  
Alignment · 420

AlignToBand · 978  
AllowGrayed · 409  
AllowStreamedConnect · 621  
Almacén de Objetos · 496, 956  
alter table · 191, 194  
Apartamentos · 119, 754, 768  
Append · 483  
AppendRecord · 469  
Application · 844  
ApplyUpdates · 717, 720  
    TDatabase · 592  
    TDBDataset · 592  
    TDBDataSet · 591  
AppServer · 771  
AsFieldValue · 915  
AsFloat · 383  
AsInteger · 383  
ASP · 881  
    objetos globales · 883  
Assert · 368  
AsString · 383  
AutoCalcFields · 391  
AutoClone · 525  
AutoDisplay · 410  
Autoedición · 468  
AutoEdit · 353, 400, 413, 419, 468  
Automatización OLE · 97  
AutoSessionName · 577  
AutoSize · 404  
AutoStopAction · 623  
AutoStretch · 978

---

## B

Balance de carga · 780  
Bandas · 971, 975  
BeforeAction · 433  
BeforeApplyUpdates · 775  
BeforeCheckAccessRights · 949  
BeforeDelete · 419  
BeforeDispatch · 848, 850, 877  
BeforeExecute · 775  
BeforeGetRecords · 693, 775  
BeforeInsert · 484  
BeforePost · 245, 481, 632  
BeforePrint · 976, 977  
BeforeRowRequest · 716, 775

BeforeUpdateRecord · 729, 737  
 BeforeValidateUser · 948  
 BeginTrans · 656  
 Bibliotecas  
   ActiveX · 94, 682  
   de tipos · 98, 759  
 Bibliotecas de Enlace Dinámico  
   UDFs para InterBase · 250  
 BLOCK SIZE · 565  
 BlockReadSize · 456, 663  
 Bloqueos  
   en dos fases · 263  
   en transacciones · 261  
   oportunistas · 565  
 Bookmark · 369  
 BufferChunks · 626  
 Búsquedas · 439  
 ButtonStyle · 424

---

## C

Caché de módulos · 846  
 CachedUpdates  
   TDBDataSet · 591  
   TIBCustomDataSet · 626  
 CacheSize · 663  
 Campos  
   agregados · 446  
   calculados · 387, 390  
   calculados internos · 437  
   de búsqueda · 424  
   de referencia · 391  
   TBCDField · 533  
   TFMTBCDField · 533  
   TSQLTimeStampField · 533  
   validaciones · 477  
 Campos blob · 183, 414, 434  
   LoadFromFile · 414  
   LoadFromStream · 415  
   SaveToFile · 414  
   SaveToStream · 415  
 Cancel · 467, 494  
 CancelBatch · 668  
 CancelUpdates  
   TClientDataSet · 474  
   TDBDataSet · 593  
 CanModify · 580  
 CGI · 841

---

## Ch

ChangeCount · 360  
   TClientDataSet · 474  
 check · 187  
 CheckBrowseMode · 476, 494  
 ChildDefs · 460  
 ChildName · 779

---

## C

Clases  
   fábricas de · Véase Fábricas de clases  
 CloneCursor · 375, 693  
 CoCreateInstance · 81  
 CoCreateInstanceEx · 85  
 CoGetClassFactory · 106  
 CoGetClassObject · 85, 91  
 Columns · 421, 425  
 Comandos HTTP · 791  
 CommandText  
   TADOCommand · 655  
   TADODataSet · 659  
   TClientDataSet · 685  
   TSQLDataSet · 531  
 CommandTimeout · 655  
 CommandType · 531, 655, 659  
 Commit  
   TDatabase · 590  
   TIBTransaction · 623  
   TSQLConnection · 524  
 CommitRetaining · 623  
 CommitTrans · 656  
 CommitUpdates  
   TDBDataSet · 592  
 Componentes de impresión · 977  
 ComponentState · 364  
 Conciliación · 718, 747  
 Conexiones compartidas · 776  
 Conjuntos de datos  
   anidados · 459  
   marcas de posición · 369  
 Connected  
   TIBDatabase · 621  
 Connection · 776  
 ConnectionString · 651  
 ConstraintErrorMessage · 478, 676  
 Constraints · 480  
 Consultas  
   no actualizables · 597  
   preparación · 541  
 Consultas recursivas · 306

- DB2 · 328
- ContentFields · 858
- ContentStream · 861
- ContentType · 861
- Contraseñas
  - codificación · 516
  - DB Express · 515
- Controles de datos · 354
- Controles dibujados por el propietario · 405
- ControlStyle · 434
- Cookies · 869
- CopyToClipboard · 410
- CreateCOMObject · 81
- CreateDataSet · 460, 745
- CreateForm · 846
- CreateFromFactory · 107
- CreateOleObject · 641
- Cursores · 576
  - deallocate · 290
  - especiales del BDE · 612
  - for update · 317
  - Microsoft SQL Server · 660
  - Oracle · 308
  - propiedades · 615
  - retención de · 623
  - SQL Server · 289
- CursorLocation · 660
- CursorType · 660, 662
- CustomConstraint · 478, 676
- CutToClipboard · 410

## D

- Data
  - TAdapter · 912
  - TClientDataSet · 471
  - TCustomProvider · 697
- Database Desktop · 209
- DatabaseError · 480
- DatabaseName
  - TDatabase · 578
  - TIBDatabase · 621
  - TQuery · 580
  - TTable · 579
- DataSetField
  - TNestedTable · 588
- DB Express
  - conexiones · 525
  - configuración · 503
  - conjuntos de datos · 529
  - contraseñas · 515
  - controladores · 502

- perfiles · 200
- transacciones · 521, 733
- DB2
  - Control Center · 322
  - fetch first · 211
  - índices · 302
  - tipos de datos · 323
  - triggers · 326
- DbiBatchMove · 618
- DbiCreateTable · 606
- DbiDoRestructure · 608
- DbiExit · 603
- DbiInit · 602
- DbiPackTable · 610
- DbiRegenIndexes · 611
- DCOM · 81
- DeactivateWebModule · 846
- Decision Cube · 993
- DEFAULT DRIVER · 574
- DefaultAction · 622
- DefaultDatabase · 622
- DefaultDrawColumnCell · 425
- DefaultExpression · 483, 676
- DefaultTimeout · 939
- DefaultTransaction · 622
- Delegación · 72, 401
- delete · 223
- Delete · 470, 591
- DeleteSQL · 598
- Delta · 471
- deref · 589
- Diálogo
  - conciliación de errores · 749
- Diccionario de Datos
  - dominios · 192
- DisableControls · 370, 455
- DispatchPageName · 948
- dispinterface · 133, 771
- DisplayFormat · 385
- DisplayLabel · 381, 402, 420
- DisplayText · 388
- DisplayType · 940
- DisplayValues · 386
- distinct · 207, 215, 219
- Distributed COM · Véase DCOM
- DllCanUnloadNow · 95
- DllGetClassObject · 90, 95, 106
- DllRegisterServer · 95
- DllUnregisterServer · 95
- Dominios · 192
- DRIVER FLAGS · 256
- DropDownAlign · 409
- DropDownRows · 409, 424
- DropDownWidth · 409

DroppedDown · 407

---

## **E**

EDatabaseError · 480  
EDBEngineError · 603  
Edit · 413, 467  
EditFormat · 386  
EditMask · 386, 402, 477  
Editor de Campos · 379, 390  
Editor de Enlaces · 452  
Ejecución asíncrona · 655  
ENABLE INTEGERS · 302  
ENABLE SCHEMA CACHE · 568, 582  
EnableControls · 370, 455  
EnableSocketTransport · 762  
EnableWebTransport · 766  
Encuentro natural · 207  
Encuentros externos · 221  
Enumerator · 910  
Errors  
    TADOConnection · 657  
Etiquetas transparentes · 852  
Execute  
    TADOCommand · 655  
    TADOConnection · 654  
ExecuteAccess · 946  
ExecuteDirect · 519, 728, 742  
ExecuteOptions · 666  
    TADOCommand · 656  
exists · 218  
explain plan · 307  
Exported · 756  
Expresiones regulares · 837  
Expression · 446  
ExtractContentFields · 858  
ExtractQueryFields · 858

---

## **F**

Fábricas de clases · 89, 764  
FetchAll · 591  
FetchOnDemand · 709  
FetchParams · 681  
Ficheros asignados en memoria · 563  
FieldByName · 383  
FieldDefs · 396, 460  
FieldFlags · 928, 935, 940  
FieldName · 381  
Fields · 383  
Fields editor · Véase Editor de Campos

FieldValues · 384  
FILL FACTOR · 565  
Filter · 442, 935  
Filtered · 442, 445  
FilterOptions · 442  
Filtros · 441, 585  
    expresiones en MyBase · 442  
    latentes · 444, 585  
    rápidos · 443  
FindField · 383  
FindFirst · 445  
FindLast · 445  
FindNext · 445  
FindPrior · 445  
First · 366  
Flat · 431  
FocusControl · 402, 478  
ForceNewPage · 985  
foreign key · 189  
FormatDateTime · 385, 854  
FormatFloat · 385  
Formularios activos · 891  
Found · 445  
free\_it · 252  
Funciones de conjuntos · 213  
Funciones de respuesta  
    del BDE · 616  
Funciones definidas por el usuario · 206, 250

---

## **G**

GeneratorField · 631  
GetGroupState · 450  
GetIDsOfNames · 133  
GetInterface · 52  
GetRecords · 697  
GetServer · 771  
grant · 197  
group by · 213, 214  
GUID · 47

---

## **H**

HandleReconcileError · 749  
having · 215  
HideOptions · 946  
Hint · 431  
holdlock · 289, 316  
HResult · 102  
HTML · 811  
    campos ocultos · 864

- etiquetas · 812
- formularios · 795, 798
- HTTP
  - cabeceras · 800
  - cookies · 869
  - GET · 791
  - imágenes · 859
  - peticiones · 855
  - POST · 794
  - protocolo · 766, 787
  - redirección · 861
  - respuestas · 800
  - SOAP · 953
- HttpExtensionProc · 846

## I

- IAppServer · 698, 755, 775, 780, 967
- IClassFactory · 90
- ICConnectionPoint · 145
  - Advise · 146
  - Unadvise · 146
- ICConnectionPointContainer · 145
- IDataBlock · 763
- IDataBroker · 757
- IDataIntercept · 763
- Identificadores Globales Unicos · Véase GUID
- identity
  - identidades · 735
- identity, atributo · 284
- IDL · 99
- IdleTimer
  - TIBDatabase · 622
  - TIBTransaction · 622
- IEnumConnectionPoints · 145
- IGetProducerComponent · 904
- IGetWebAppServices · 904
- IInterface · 38
  - QueryInterface · 38, 51
- IInvokable · 957
- Image Editor · 430
- Imágenes · 859
- ImportedConstraint · 479
- IndexFieldNames · 428, 439, 452
  - TADOTable · 665
  - TTable · 583
- IndexName · 452, 583
- Indices
  - creación · 192
  - en DB2 · 325
  - en Oracle · 302

- reconstrucción y optimización · 194
- tablas organizadas por · 303
- TClientDataSet · 435
- Initialize · 106
- inner join · 223
- insert · 223
- Insert · 483
- InsertRecord · 469
- InsertSQL · 598
- Integridad referencial · 164, 189
  - Acciones referenciales · 190
  - relaciones maestro/detalles · 458
  - simulación · 293
- InterBase
  - alertadores de eventos · 248
  - check · 187
  - cláusula rows · 212
  - computed by · 185
  - configuración en el BDE · 569
  - excepciones · 246
  - external file · 185
  - generadores · 244
  - perfiles · 199, 570
  - procedimientos almacenados · 232
  - recogida de basura · 269
  - set statistics · 194
  - tipos de datos · 182
  - UDF · 250
  - usuarios · 195
- InterBase Express · 619
- InterBase Server Manager · 195
- InterceptGUID · 763
- InterceptName · 763
- Interceptores · 763
- Interfaces
  - de envío · 133, 771
  - delegación · 72
  - duales · 134
  - eventos · 61
  - implementación · 34
  - introspección · 49
  - polimorfismo · 39
  - resolución de métodos · 35
  - salientes · 83, 144
  - tiempo de vida · 40
- Internet Express · 894
- InTransaction
  - TADOConnection · 656
- Invoke · 133, 772
- InvRegistry · 960
- IPageResult · 904
- IPersistFile · 82
- IProducerContent · 906
- IProviderSupport · 715, 718, 723, 732

PSGetTableName · 723  
ISAPI · 807, 842, 846  
IsBlob · 726  
IShellLink · 82  
IsMasked · 402  
IsNull · 384, 391  
IsolationLevel · 657  
IUnknown · 38, 957  
IWebAppServices · 902

---

## **J**

JScript · 646

---

## **K**

KeyField · 408  
KeyFields · 393  
KeyPreview · 407

---

## **L**

Last · 366  
Lenguaje de Descripción de Interfaces ·  
    *Véase* IDL  
like · 205, 935  
LinkToPage · 915, 918  
LoadBalanced · 782  
Loaded · 364, 652, 676, 711, 777  
LoadMemo · 404  
LoadPicture · 410  
LOCAL SHARE · 563  
Locate · 439, 584  
LockType · 667  
LoginPrompt · 515  
    TCustomRemoteServer · 773  
    TIBDatabase · 621  
Lookup · 439, 949  
Lookup fields · *Véase* Campos de referencia  
LookupDataset · 393  
LookupKeyFields · 393  
LookupResultField · 393

---

## **M**

Marcas de posición · 369  
Marshaling · 698, 759  
MasterFields · 452

MasterSource · 452  
Max · 478  
MAX DBPROCESSES · 573  
MAX ROWS · 616  
MaxConnections · 847, 939  
MaxRecords  
    TADODataSet · 663  
MaxSessions · 939  
MaxStmtsPerConn · 525  
Mensajes  
    CM\_GETDATALINK · 434  
MergeChangeLog · 472  
Métodos de clase · 490  
Microsoft SQL Server · *Véase* SQL Server  
Midas · 544, 781, 893  
    intercepción de paquetes · 763  
    seguridad · 772  
Min · 478  
Mode · 935  
Modelo de Objetos Componentes  
    apartamentos · 754, 768, 892  
    in-process servers · 93  
Modified · 360, 474, 485, 591  
ModifySQL · 598  
Módulos  
    caché · 767  
    remotos · 682, 751  
    SOAP · 967  
Módulos Web · 842  
    acciones · 848  
Monitor  
    BDE · 581  
    DB Express · 526  
Motor de Datos  
    Administrador · 560  
MoveBy · 366  
MoveNext · 661  
Multicasting · 146

---

## **N**

Navegación · 366  
Navigate · 357  
NestedDataSet  
    TDataSetField · 588  
NET DIR · 564  
NewValue · 598, 729  
Next · 366  
NextRecordset · 664  
NSAPI · 842  
Null · 206, 441, 469

## O

OBJECT MODE · 572, 586  
 Object Repository · *Véase* Almacén de Objetos  
 ObjectBroker · 781  
 ODBC · 564  
 OldValue · 482, 598  
 OleCheck · 83  
 OnAccept · 461  
 OnAction · 849, 852, 867  
 OnAddReports · 985  
 OnCalcFields · 390, 440  
 OnCellClick · 427, 433  
 OnChange · 403  
     TField · 479  
 OnClose · 423  
 OnCloseQuery · 493, 731  
 OnCreate · 423  
 OnDataChange · 412  
 OnDrawColumnCell · 425, 433, 473, 594  
 OnDrawItem · 406  
 OnEditButtonClick · 424  
 OnExecute · 920  
     TAction · 372  
     TAdapterAction · 916, 934, 942  
 OnExecuteComplete · 656  
 OnExit · 403  
 OnFetchComplete · 667  
 OnFetchProgress · 667  
 OnFilterRecord · 442  
 OnFindTemplateFile · 908  
 OnFormatCell · 859  
 OnGetParams · 941  
 OnGetTableName · 724  
 OnGetText · 386, 448, 450  
 OnGetUsername · 773  
 OnGetValue  
     TAdapterField · 919, 935  
 OnHTMLTag · 852  
 OnInfoMessage · 658  
 OnLogin · 200, 774  
 OnLogTrace · 527  
 OnMeasureItem · 406  
 OnNewRecord · 316, 484, 632  
 OnPostError · 604  
 OnPrint · 982  
 OnReconcileError · 721  
 OnSetText · 386, 479  
 OnSQLEvent · 628  
 OnStateChange · 411  
 OnTitleClick · 427, 439  
 OnTrace · 527

OnUpdateData · 412, 734  
 OnUpdateRecord · 597, 599  
 OnValidate · 479  
 Open Tools API · 53  
 Optimización de consultas  
     Oracle · 307  
 Oracle · 302  
     clusters · 303  
     configuración en el BDE · 570  
     connect by · 306  
     controlador OLE DB · 644  
     cursores · 308  
     packages · 312  
     Procedimientos almacenados · 304  
     rownum · 211  
     secuencias · 315  
     SQL\*Plus · 297  
     tipos de datos · 300, 589  
     tipos de objetos · 317, 585, 586  
     transacciones · 524  
     triggers · 309  
 order by · 210  
 organization index · 303  
 outer join · *Véase* Encuentros externos  
 Outgoing interfaces · *Véase* Interfaces salientes  
 OutputDebugString · 438

## P

Packages  
     Oracle · 312  
     QuickReport · 972  
 PageCount · 664  
 PageSize · 664, 931  
 Paginación · 931  
 Paradox  
     configuración de la caché · 566  
     empaquetamiento · 610  
     integridad referencial · 610  
 Parámetros · 536  
 Params  
     TDatabase · 579  
     TIBDatabase · 621  
     TIBTransaction · 624  
 ParentConnection · 779  
 Password · 767, 774  
 PasteFromClipboard · 410  
 Perro  
     de Codd · 162, 166  
 PickList · 424  
 Plantillas

WebSnap · 907  
 poAllowCommandText · 685  
 Polimorfismo · 39  
 poPropagateChanges · 729  
 Post · 467, 494, 591  
 Prepare · 542  
 primary key · 188  
 Prior · 366  
 Privilegios · 197  
 Procedimientos almacenados · 169, 229  
     de selección · 625  
     SQL Server · 288  
 ProgIDFromCLSID · 88  
 PromptDataSource · 654  
 Proveedores · 673  
 ProviderFlags · 714, 725  
 ProviderName · 697, 706

---

## Q

QueryFields · 858  
 QueryInterface · 38, 51, 82, 771, 957  
 QuickReport  
     Componentes de impresión · 977  
     expresiones · 978  
     Informes compuestos · 984  
     Informes con grupos · 980  
     Informes master/detail · 983  
     plantillas y asistentes · 972

---

## R

raise · 480  
 Randomize · 782  
 RDB\$DB\_KEY · 629  
 ReadOnly · 400  
 RecNo · 366  
 Redirección · 861  
 RedirectOptions  
     TAdapterAction · 919  
 Referrer · 861  
 Refresh · 713  
 RefreshRecord · 714  
 RegisterInvokableClass · 962  
 RegisterPackageWizard · 56  
 RegisterPooled · 767  
 RegisterProvider · 757  
 RegisterXSClass · 966  
 RegisterXSInfo · 966  
 Registros de transacciones · 261  
 Rejillas de selección múltiple · 428

Relación master/detail · 191, 451, 593, 983  
 Replicación · 266  
 RequestLive · 580  
 Required · 477  
 ResetAfterPrint · 979  
 Resolutor · 717  
 ResolveToDataSet · 716  
 resourcestring · 431  
 Restricciones de integridad · 162, 186  
     clave artificial · 188  
     clave externa · 164, 189  
     clave primaria · 163, 188  
     claves alternativas · 188  
 RevertRecord  
     TClientDataSet · 474  
     TDBDataSet · 593  
 revoke · 197  
 ROLE NAME · 200  
 Rollback  
     TDatabase · 590  
     TIBTransaction · 623  
     TSQLConnection · 524  
 RollbackRetaining · 623  
 RollbackTrans · 656  
 ROT · 112, 114, 139  
 rownum · 211  
 rows · 212, 225, 873  
 ROWSET SIZE · 571  
 RPC · 47  
 Running Object Table · Véase ROT

---

## S

safecall · 102  
 SavePoint · 475  
 select · 204  
     selección única · 217  
     subconsultas correlacionadas · 218  
 SelectedField · 422  
 SelectedIndex · 422  
 SelectedRows · 428  
 Serializabilidad · 259  
 SERVER NAME · 569  
 Servidores de automatización · 131  
 Sesiones  
     BDE · 575, 576, 601  
     WebSnap · 937  
 Session · 577  
 SessionID · 938  
 SessionName  
     TDatabase · 578  
     TSession · 577



- TTable · 579
- SetFields · 469
- ShowException · 721
- SOAP · 953
  - WSDL · 954
- Sort · 665
- SortFieldNames · 531
- SQL
  - Lenguaje de Manipulación de Datos · 223
- SQL Links · 560
- SQL Monitor · 581
- SQL Net · 570
- SQL Server · 271
  - @@identity · 736
  - cláusula top · 211
  - cursores · 289
  - grupos de ficheros · 279
  - identidades · 284, 735
  - Índices · 288
  - integridad referencial · 286
  - SQL Enterprise Manager · 271
  - tipos de datos · 282
- STA · 892
- starting with · 540, 684
- StartTransaction · 590
- State · 438, 456, 466
- StatusFilter · 473
- Stored procedures · Véase Procedimientos almacenados
- StreamedConnected · 507
- StringReplace · 629
- Subcomponentes · 461
- Suecas
  - en *topless* · 775
- SupportCallbacks · 762
- Supports · 52, 66, 74
- suspend · 236
- SyncObjs · 628

---

## T

- Tablas
  - mutantes · 310
  - temporales · 285
- TAbstractWebSession · 938
- TAction
  - OnExecute · 372
- TActionList · 359
- TActionManager · 358, 372
- TAdapter
  - Actions · 912
  - Data · 912

- TAdapterAction · 934
  - ExecuteAccess · 946
  - OnExecute · 916, 920, 934, 942
  - OnGetParams · 941
  - RedirectOptions · 919
- TAdapterActionButton · 924
  - HideOptions · 946
- TAdapterField · 934
  - ActionValue · 916, 919
  - FieldFlags · 928, 935, 940
  - OnGetValue · 919, 935
- TAdapterGrid · 930
- TAdapterPageProducer · 914, 920
- TADOCCommand · 650
- TADOCConnection · 650
- TADODDataSet · 659
  - NextRecordset · 664
  - Sort · 665
- TADOQuery · 659
- TADOStoredProc · 659
- TADOTable · 659
- TADTField · 586
- TAggregateField · 381, 446
- TApplication
  - ShowException · 721
- TArrayField · 586
- TBCDField · 533
- TBinaryField · 381
- TBlobStream · 415
- TBookmark · 369
- TBookmarkList · 428
- TBookmarkStr · 369
- TBooleanField · 383, 386, 409, 676
- TCalendar · 411
- TCGIApplication · 844, 845
- TChart · 994
- TCheckConstraint · 480
- TClientDataSet
  - AfterRowRequest · 716
  - ApplyUpdates · 717, 720
  - BeforeGetRecords · 693
  - BeforeRowRequest · 716
  - CancelUpdates · 474
  - ChangeCount · 360, 474
  - CloneCursor · 375, 693
  - CommandText · 685
  - CreateDataSet · 745
  - Data · 471
  - Delta · 471
  - FetchOnDemand · 709
  - Filter · 935
  - GetGroupState · 450
  - Locate · 439
  - Lookup · 439

- MasterFields · 452
- MergeChangeLog · 472
- OnReconcileError · 721
- ProviderName · 706
- RecNo · 366
- Refresh · 713
- RefreshRecord · 714
- RevertRecord · 474
- SavePoint · 475
- StatusFilter · 473
- UndoLastChange · 474
- TColumn · 421
- TComObject · 107
  - CreateFromFactory · 107
  - Initialize · 106
- TComponentFactory · 754
- TConnectionBroker · 776
- TControlClass · 402
- TCorbaConnection · 758
- TCriticalSection · 628
- TCustomADODataset · 659
- TCustomConnection
  - RegisterClient · 509
  - UnRegisterClient · 509
- TCustomMaskEdit · 402
- TCustomObjectBroker · 782
- TCustomProvider
  - Data · 697
  - Exported · 756
  - GetRecords · 697
- TCustomRemoteServer
  - LoginPrompt · 773
  - OnGetUsername · 773
  - OnLogin · 774
- TDatabase · 200, 562, 577, 590, 592
  - DatabaseName · 578
  - InTransaction · 590
  - IsSqlBased · 578
- TDataLink · 401
- TDataSet
  - ActiveBuffer · 365
  - AfterClose · 711
  - AfterOpen · 711
  - AfterPost · 481
  - BeforePost · 481
  - Cancel · 467
  - Constraints · 480
  - Delete · 470
  - DisableControls · 370
  - Edit · 467
  - EnableControls · 370
  - Filter · 442
  - Filtered · 442
  - Found · 445
  - Locate · 439
  - Lookup · 439, 949
  - Modified · 360, 474, 485
  - MoveBy · 366
  - navegación · 366
  - OnFilterRecord · 442
  - OnNewRecord · 484
  - Post · 467
  - State · 438, 466
  - UpdateStatus · 472
- TDataSetAdapter · 927
  - PageSize · 931
- TDataSetAdaptor
  - Mode · 935
- TDataSetField · 381, 425, 586, 706
- TDataSetPageProducer · 865
- TDataSetProvider · 673
  - AfterApplyUpdates · 734
  - AfterGetRecords · 693
  - AfterRowRequest · 716
  - AfterUpdateRecord · 736
  - BeforeRowRequest · 716
  - BeforeUpdateRecord · 729, 737
  - OnGetTableName · 724
  - OnUpdateData · 734
  - Options · 729
  - poAllowCommandText · 685
  - resolutor · 717
  - ResolveToDataSet · 716
  - UpdateMode · 724
- TDataSetTableProducer · 858
  - OnFormatCell · 859
- TDataSource · 353, 399, 411, 430, 454
  - AutoEdit · 468
- TDBChart · 993, 997
  - series · 998
- TDBCheckBox · 402, 409
- TDBCColumn · 450
- TDBComboBox · 405
- TDBCtrlGrid · 366, 433
- TDBDataSet · 593
- TDBEdit · 366, 402
  - OnChange · 403
- TDBGrid · 354, 366, 419
  - Expanded · 587
  - OnDrawColumnCell · 473
  - OnTitleClick · 439
  - ShowPopupEditor · 425
- TDBGridColumn · 421
- TDBImage · 402, 410, 434
- TDBListBox · 405
- TDBLookupComboBox · 394, 402, 408, 424
- TDBLookupListBox · 408
- TDBMemo · 402, 403, 434

- TDBNavigator · 354, 430, 713
- TDBRadioGroup · 405, 410, 434
- TDBRichEdit · 403, 434
- TDBText · 404
- TDCOMConnection · 759
- TDecisionCube · 999
- TDecisionGraph · 999
- TDecisionGrid · 999, 1000
- TDecisionPivot · 999, 1001
- TDecisionQuery · 999
- TDecisionSource · 999
- TDispatchConnection · 781
- TDrawGrid · 419
- TEndUserSessionAdapter · 944
- Text
  - TField · 388, 479
- TField · 381
  - CustomConstraint · 478
  - DefaultExpression · 483
  - DisplayText · 388
  - FocusControl · 478
  - IsBlob · 726
  - IsNull · 384, 391
  - KeyFields · 393
  - Lookup · 393
  - LookupDataset · 393
  - LookupKeyFields · 393
  - LookupResultField · 393
  - Name · 381
  - NewValue · 729
  - OldValue · 482
  - OnChange · 479
  - OnSetText · 479
  - OnValidate · 479
  - ProviderFlags · 714, 725
  - Required · 477
  - Text · 388, 479
  - ValidChars · 389
  - Value · 383
- TFieldDataLink · 401
- TFieldDef · 460
- TFMTBCDField · 533
- TForm
  - OnCloseQuery · 731
- TGetRecordOptions · 698
- TGraphicControl · 404
- TGUID · 48
- Thread affinity · 150
- threadvar · 964
- THTTTPRIO · 957, 965
- THTTPSoapDispatcher · 960
- THTTPSoapPascalInvoker · 960
- TIBDatabase · 620
  - AllowStreamedConnect · 621
  - Connected · 621
  - DatabaseName · 621
  - IdleTimer · 622
  - LoginPrompt · 621
- TIBDataSet · 625, 630
- TIBEvents · 250
- TIBQuery · 625, 630
  - GeneratorField · 631
  - UpdateSQL · 630
- TIBSQL · 625
- TIBSQLMonitor · 627
- TIBStoredProc · 625
- TIBTable · 625, 630
- TIBTransaction · 620, 622
  - AutoStopAction · 623
  - IdleTimer · 622
- TIBUpdateSQL · 630
- TInetXPageProducer · 894, 922
- TInterfacedObject · 39, 52
- TISAPIApplication · 844
- TLabel · 402
- TLineSeries · 995
- TListView · 533
  - StringWidth · 535
- TLocalConnection · 779
- TLocateFileService · 907
- TLoginFormAdapter · 944
- TMemoryStream · 415, 991
- TNestedTable · 586, 588
- TNumericField · 381
- TObjectField · 381
- TPacketInterceptFactory · 764
- TPageControl · 997
- TPageProducer · 851, 921
- TPopupMenu · 443
- TQRCompositeReport · 984
- TQRDBImage · 977
- TQRDetailLink · 983
- TQRExpr · 977, 978
- TQRExprMemo · 977, 980
- TQRGroup · 980
- TQRMemo · 977
- TQuery · 204, 580, 981, 997
  - CanModify · 580
  - DatabaseName · 580
  - RequestLive · 580
- TQueryTableProducer · 858
- TQuickRep · 981
  - Preview · 973
  - Print · 973
  - PrintBackground · 973
- TQuickReport · 973
- Transacciones · 253, 254
  - ADO Express · 656

- aislamiento · 258, 523, 579, 624, 657
- arquitectura multigeneracional · 265
- commit work · 256
- DB Express · 521
- distribuidas · 622
- InterBase Express · 622
- lecturas repetibles · 259
- niveles de aislamiento · 732
- resolución · 732
- rollback work · 256
- serializabilidad · 259
- start transaction · 256
- Transact SQL
  - procedimientos almacenados · 288
  - triggers · 291
- Transaction logs · Véase Registros de transacciones
- Transact-SQL · 229
- TransIsolation · 260
- TRDSCONNECTION · 651
- TReferenceField · 381, 586
- TRemotable · 966
- TRemoteDataModule · 752
  - RegisterProvider · 757
  - UnRegisterProvider · 757
- Triggers · 169, 239, 730
  - anidados · 294, 326
  - new/old · 240, 599
  - Oracle · 309
  - recursivos · 295
  - SQL Server · 291
- try/finally · 369
- TSession · 601
- TSessionsService · 937
  - DefaultTimeout · 939
  - MaxSessions · 939
- TSharedConnection · 758, 779
- TSimpleObjectBroker · 782
- TSoapConnection · 758, 967
- TSocketConnection · 761, 894
  - InterceptGUID · 763
  - InterceptName · 763
- TSQLClientDataSet · 703
- TSQLConnection · 509, 776
  - AutoClone · 525
  - Commit · 524
  - ExecuteDirect · 519, 728, 742
  - LoginPrompt · 515
  - MaxStmtsPerConn · 525
  - Rollback · 524
  - StreamedConnected · 507
- TSQLDataSet
  - CommandText · 531
  - CommandType · 531
  - SortFieldNames · 531
- TSQLMonitor · 526
- TSQLQuery
  - Prepare · 542
- TSQLTimeStampField · 533
- TStoredProc · 231
- TStringField · 383
- TStringGrid · 419
- TStrings · 405
- TTable · 580
  - DatabaseName · 579
  - TableName · 579
- TTransactionDesc · 522
- TTypedComObject · 106
- TTypedComObjectFactory · 106
- TUpdateSQL · 598
- TWebActionItem · 848
- TWebAppComponents · 905
- TWebApplication · 844
- TWebAppPageModule · 903
- TWebBrowser · 357
  - Navigate · 357
- TWebConnection · 765, 894
  - Password · 767, 774
  - UserName · 767, 774
- TWebDataModule · 903
- TWebDispatcher · 844, 848
- TWebModule · 843, 959
  - BeforeDispatch · 848, 850
- TWebPageInfo · 947
- TWebPageModule · 903
- TWebRequest
  - ContentFields · 858
  - Cookie · 870
  - CookieFields · 870
  - QueryFields · 858
  - Referrer · 861
- TWebResponse · 850
  - ContentStream · 861
  - ContentType · 861
  - SetCookieField · 869
- TWebSession
  - SessionID · 938
- TWebUserList · 944
  - BeforeCheckAccessRights · 949
  - BeforeValidateUser · 948
- TWSDLHTMLPublish · 960
- TXMLBroker · 894
- TXSRemotable · 966
- Type Library · Véase Bibliotecas de tipos

---

## U

Unadvise · 146  
 Unassigned · 41  
 UndoLastChange · 474  
 Unidirectional · 580  
     TIBCustomDataSet · 626  
 unique · 188  
 UnRegisterProvider · 757  
 update · 223  
     problemas en InterBase · 224  
 Update · 668  
 UpdateMode · 724  
 UpdateObject · 598  
 UpdateRecord · 414  
 UpdateRecordTypes · 594  
 UpdateRegistry · 753, 762, 764, 767  
 UpdatesPending · 591  
 UpdateSQL  
     TIBQuery · 630  
 UpdateStatus · 472, 593  
 URL · 789  
     JavaScript · 839  
 UserAgent · 857  
 UserName · 767, 774

---

## V

Validaciones · 476  
 ValidChars · 389, 476

Valores nulos · 206  
 Valores por omisión  
     default · 186, 245  
     TField · 482  
 Value · 383  
 ValueChecked · 410  
 ValueUnchecked · 410  
 VarArrayOf · 440  
 Variant · 384, 440  
 VarIsNull · 441  
 VarToStr · 440  
 Visible · 420  
 VisibleButtons · 430  
 Vistas · 195, 225  
     actualizables mediante triggers · 314

---

## W

Web Debugger · 809  
 WebContext · 905, 938  
 WebSnap · 899  
     búsquedas · 933  
     paginación · 931  
     sesiones · 908, 937  
     ubicación de plantillas · 907  
 where · 205  
 Windows ISQL · 231  
 with admin option · 200  
 with check option · 227  
 WSDL · 954

*La Cara Oculta de Delphi 6*  
*Copyright © 2002-2010, Ian Marteens*  
*Prohibida la reproducción total o parcial de esta obra,*  
*por cualquier medio, sin autorización escrita del autor.*  
*Madrid, España, 2010*