La cara oculta de DELPHI 4

...Mientras intentaba inútilmente atrapar un rayo de luz
en los confines de una oscura galaxia,
Ella escuchaba a los tres guardianes
del secreto de la Pirámide,
y con el Resplandor de sus manos
me indicaba el camino de regreso a casa...

A Naroa, porque sigo creyendo en ella.

INDICE

PRÓLOGO DEL AUTOR **19**

CONVENIOS SINTÁCTICOS 21
ACERCA DE LA TERMINOLOGÍA EMPLEADA 21
AGRADECIMIENTOS 21

PRÓLOGO A LA SEGUNDA EDICIÓN **23**

PARA MÁS EJEMPLOS Y ACTUALIZACIONES... 23

ENTORNO Y HERRAMIENTAS **27**

1. ¿ES DELPHI SU LENGUAJE? **29**

BREVE HISTORIA DE UN LENGUAJE LLAMADO DELPHI 29
¿ES DELPHI UN LENGUAJE "SENCILLO"? 31
PROGRAMACIÓN ORIENTADA A OBJETOS VERDADERA 32
ARQUITECTURA INTEGRADA DE COMPONENTES 33
DELPHI GENERA CONTROLES ACTIVEX 34
TRATAMIENTO DE EXCEPCIONES 34
VELOCIDAD DE EJECUCIÓN 35
VELOCIDAD DE COMPILACIÓN Y ENLACE 35
RETROALIMENTACIÓN INMEDIATA 36
EL DEPÓSITO DE OBJETOS Y LA HERENCIA VISUAL 36
APLICACIONES DISTRIBUIDAS 37
COMPONENTES DE DELPHI 38

2. HERRAMIENTAS Y UTILIDADES **41**

DELPHI Y EL TECLADO 41
CODE INSIGHT: AYUDA DURANTE LA EDICIÓN 44
CLASS COMPLETION 48
HERRAMIENTAS DE DEPURACIÓN 49
OBJECT BROWSER 51
BDE ADMINISTRATOR 51
DATABASE EXPLORER 52
DATABASE DESKTOP 53
DATA MIGRATION WIZARD 54
SQL MONITOR 55
IMAGE EDITOR 55

3. UNIDADES, PROYECTOS Y PAQUETES 57

LA ESTRUCTURA DE UN PROYECTO DE APLICACIÓN 58
¿QUÉ SE PUEDE DECLARAR? 60
SINTAXIS DE LAS UNIDADES 62
LA CLÁUSULA *USES* Y LAS REFERENCIAS CIRCULARES 64
LA INCLUSIÓN AUTOMÁTICA DE UNIDADES 65
LOS FICHEROS *DFM* 66
BIBLIOTECAS DE ENLACE DINÁMICO 68
PAQUETES 69
EL ABC DEL USO DE PAQUETES 72
PAQUETES DE TIEMPO DE DISEÑO Y DE EJECUCIÓN 73
LOS GRUPOS DE PROYECTOS 74

4. SISTEMAS DE BASES DE DATOS 77

ACERCA DEL ACCESO TRANSPARENTE A BASES DE DATOS 77
BASES DE DATOS RELACIONALES 78
INFORMACIÓN SEMÁNTICA = RESTRICCIONES 80
RESTRICCIONES DE UNICIDAD Y CLAVES PRIMARIAS 81
INTEGRIDAD REFERENCIAL 82
¿QUÉ TIENE DE MALO EL MODELO RELACIONAL? 83
BASES DE DATOS LOCALES Y SERVIDORES SQL 85
CARACTERÍSTICAS GENERALES DE LOS SISTEMAS SQL 87
EL FORMATO PARADOX 88
EL FORMATO DBF7 91
CRITERIOS PARA EVALUAR UN SERVIDOR SQL 92
INTERBASE 95
MICROSOFT SQL SERVER 97
ORACLE 99
OTROS SISTEMAS DE USO FRECUENTE 100

5. EL MOTOR DE DATOS DE BORLAND 103

QUÉ ES, Y CÓMO FUNCIONA 104
CONTROLADORES LOCALES Y SQL LINKS 105
ACCESO A FUENTES DE DATOS ODBC 106
¿DÓNDE SE INSTALA EL BDE? 106
EL ADMINISTRADOR DEL MOTOR DE DATOS 108
CONFIGURACIÓN DEL REGISTRO E INFORMACIÓN DE VERSIÓN 108
EL CONCEPTO DE ALIAS 110
PARÁMETROS DEL SISTEMA 110
PARÁMETROS DE LOS CONTROLADORES PARA BD LOCALES 112
BLOQUEOS OPORTUNISTAS 114
PARÁMETROS COMUNES A LOS CONTROLADORES SQL 114
CONFIGURACIÓN DE INTERBASE 117

Configuración de MS SQL Server 118
Configuración de Oracle 120
Configuración de otros sistemas 121
Creación de alias para bases de datos locales y SQL 121
Alternativas al Motor de Datos 122

FUNDAMENTOS DEL LENGUAJE 125

6. Programación orientada a objetos: encapsulamiento 127

El problema de la división en módulos 127
Los módulos en la programación estructurada 128
Reusabilidad y el principio "abierto/cerrado" 129
Llegan los objetos 130
Operaciones sobre un tipo de datos: funciones 132
Operaciones sobre un tipo de datos: métodos 134
La primera ventaja: nombres más cortos 135
La implementación de un método 136
El estado interno de un objeto 138
Parámetros por omisión 140
Público y privado 140
Las variables de objetos son punteros 142
Construcción y destrucción: el ciclo de la vida 143
Definiendo constructores y destructores 144

7. Herencia y polimorfismo 147

Herencia = Extensión + Especialización 147
Un gato con botas sigue siendo un gato 149
La clase *TObject* 152
Jerarquías de herencia 153
Herencia múltiple, ¿sí o no? 154
Redefinición de métodos 155
La Tabla de Métodos Virtuales 158
Utilizando el comportamiento heredado 160
Sobrecarga: hay muy pocos nombre buenos 161
Público, protegido, privado... 163
Métodos abstractos 164
Referencias de clase 165
Constructores virtuales 167
Información de tipos en tiempo de ejecución 168
Métodos de clase 170

8. ELEMENTOS DE PROGRAMACIÓN CON WINDOWS **173**

SI ME NECESITAS, ¡LLÁMAME! 173
¿QUÉ ES UNA VENTANA? 174
CLASES DE VENTANA 175
EVENTOS O, MÁS BIEN, MENSAJES 176
EL CICLO DE MENSAJES Y LA COLA DE MENSAJES 177
EJECUCIÓN MODAL Y NO MODAL 180
EL PROCEDIMIENTO DE VENTANA 180
INTERCEPTANDO MENSAJES EN DELPHI 181
TAMBIÉN PODEMOS ENVIAR MENSAJES 184
APLICACIONES EN LA BANDEJA DE ICONOS 184
CICLO DE MENSAJES Y CONCURRENCIA 187
CUANDO NO HAY NADA MEJOR QUE HACER… 188
EJECUTAR Y ESPERAR 189

9. PROPIEDADES **191**

LOS PROBLEMAS DE LA POO CLÁSICA 191
PROPIEDADES 194
IMPLEMENTACIÓN DE PROPIEDADES MEDIANTE ATRIBUTOS 195
SI NO CREEMOS EN LA ALQUIMIA… 197
LA SEMÁNTICA DE LA ASIGNACIÓN A PROPIEDADES 198
PROPIEDADES VECTORIALES 200
PROPIEDADES VECTORIALES POR OMISIÓN 202
ESPECIFICACIONES DE ALMACENAMIENTO 202
ACCESO A PROPIEDADES POR SU NOMBRE 205

10. EVENTOS **207**

PUNTEROS A FUNCIONES 207
PUNTEROS A MÉTODOS 208
OBJETOS ACTIVOS: ¡ABAJO LA VIGILANCIA! 210
EVENTOS VS REDEFINICIÓN DE MÉTODOS VIRTUALES 212
EL EMISOR Y EL RECEPTOR DEL EVENTO 214
RECEPTORES COMPARTIDOS 215
TIPOS DE EVENTOS 216
NOS PIDEN NUESTRA OPINIÓN... 218
LOS EVENTOS EXPRESAN CONTRATOS SIN OBLIGACIONES 218
ACTIVACIÓN RECURSIVA 219

11. EXCEPCIONES **223**

SISTEMAS DE CONTROL DE ERRORES 223
CONTRATOS INCUMPLIDOS 224
CÓMO SE INDICA UN ERROR 225

LA EJECUCIÓN DEL PROGRAMA FLUYE EN DOS DIMENSIONES 226
EL ESTADO DE PÁNICO 227
PAGAMOS NUESTRAS DEUDAS 227
EL CONCEPTO DE "RECURSO DE PROGRAMACIÓN" 228
CÓMO TRANQUILIZAR A UN PROGRAMA ASUSTADO 229
EJEMPLOS DE CAPTURA DE EXCEPCIONES 230
CAPTURANDO EL OBJETO DE EXCEPCIÓN 231
DISTINGUIR EL TIPO DE EXCEPCIÓN 231
LAS TRES REGLAS DE MARTEENS 233
CICLO DE MENSAJES Y MANEJO DE EXCEPCIONES 234
EXCEPCIONES A LA TERCERA REGLA DE MARTEENS 235
EL EVENTO *ONEXCEPTION* 236
LA EXCEPCIÓN SILENCIOSA 239
CONSTRUCTORES Y EXCEPCIONES 240
ASERCIONES 243

12. TIPOS DE DATOS DE DELPHI 247

TIPOS NUMÉRICOS 247
FECHAS Y HORAS 249
CADENAS DE CARACTERES CORTAS 251
PUNTEROS A CARACTERES, AL ESTILO C 253
CADENAS DE CARACTERES LARGAS 254
VECTORES ABIERTOS 256
VECTORES DINÁMICOS 258
VECTORES ABIERTOS VARIANTES 259
VARIANTES 260
CLASES PARA LA REPRESENTACIÓN DE LISTAS 261
STREAMS: FICHEROS CON INTERFAZ DE OBJETOS 264

13. TÉCNICAS DE GESTIÓN DE VENTANAS 267

¿QUÉ HACE DELPHI CUANDO LO DEJAMOS SOLO? 267
CREACIÓN INMEDIATA DE VENTANAS MODALES 269
CREACIÓN INMEDIATA DE VENTANAS NO MODALES 271
MÉTODOS DE CLASE PARA LA CREACIÓN DINÁMICA 273
CÓMO TRANSFORMAR INTROS EN TABS 276
CÓMO LIMITAR EL TAMAÑO DE UNA VENTANA 277
CÓMO DISTRIBUIR EL ÁREA INTERIOR ENTRE DOS REJILLAS 279
VENTANAS DE PRESENTACIÓN 281

14. LOS MÓDULOS DE DATOS Y EL DEPÓSITO DE OBJETOS 283

LA GALERÍA DE PLANTILLAS DE DELPHI 1 283
EL DEPÓSITO DE OBJETOS 284
LAS PROPIEDADES DEL DEPÓSITO 286

La ubicación del Depósito de Objetos 287
¿Usar, copiar o heredar? 289
Herencia visual dentro del proyecto 292
Herencia de eventos 292
Las plantillas de componentes 293
Los módulos de datos 294

COMPONENTES PARA BASES DE DATOS 297

15. Conjuntos de datos: tablas 299

La jerarquía de los conjuntos de datos 299
La arquitectura de objetos del Motor de Datos 301
Tablas 303
Conexión con componentes visuales 306
Navegando por las filas 308
Marcas de posición 310
Encapsulamiento de la iteración 311
La relación *master/detail* 313
Navegación y relaciones *master/detail* 316
El estado de un conjunto de datos 321

16. Acceso a campos 323

Creación de componentes de campos 323
Clases de campos 325
Nombre del campo y etiqueta de visualización 327
Acceso a los campos por medio de la tabla 328
Extrayendo información de los campos 328
Las máscaras de formato y edición 330
Los eventos de formato de campos 331
Validación a nivel de campos 333
Propiedades de validación 334
Campos calculados 335
Campos de búsqueda 336
La caché de búsqueda 338
El orden de evaluación de los campos 339
El Diccionario de Datos 340
Conjuntos de atributos 341
Importando bases de datos 342
Evaluando restricciones en el cliente 343
Información sobre campos 346
Creación de tablas 347

17. CONTROLES DE DATOS Y FUENTES DE DATOS **351**

CONTROLES *DATA-AWARE* 351
LOS ENLACES DE DATOS 353
CREACIÓN DE CONTROLES DE DATOS 354
LOS CUADROS DE EDICIÓN 355
EDITORES DE TEXTO 356
TEXTOS NO EDITABLES 357
COMBOS Y LISTAS CON CONTENIDO FIJO 357
COMBOS Y LISTAS DE BÚSQUEDA 360
CASILLAS DE VERIFICACIÓN Y GRUPOS DE BOTONES 362
IMÁGENES EXTRAÍDAS DE BASES DE DATOS 363
LA TÉCNICA DEL COMPONENTE DEL POBRE 363
PERMITIENDO LAS MODIFICACIONES 365
BLOB, BLOB, BLOB… 367
LA CLASE *TBLOBSTREAM* 368

18. REJILLAS Y BARRAS DE NAVEGACIÓN **371**

EL FUNCIONAMIENTO BÁSICO DE UNA REJILLA DE DATOS 371
OPCIONES DE REJILLAS 372
COLUMNAS A LA MEDIDA 373
GUARDAR Y RESTAURAR LOS ANCHOS DE COLUMNAS 376
LISTAS DESPLEGABLES Y BOTONES DE EDICIÓN 377
NÚMEROS VERDES Y NÚMEROS ROJOS 379
MÁS EVENTOS DE REJILLAS 381
LA BARRA DE DESPLAZAMIENTO DE LA REJILLA 382
REJILLAS DE SELECCIÓN MÚLTIPLE 383
BARRAS DE NAVEGACIÓN 384
HABÍA UNA VEZ UN USUARIO TORPE, MUY TORPE… 385
AYUDAS PARA NAVEGAR 385
EL COMPORTAMIENTO DE LA BARRA DE NAVEGACIÓN 386
REJILLAS DE CONTROLES 388

19. INDICES **391**

INDICES EN PARADOX 391
INDICES EN DBASE 392
INDICES EN INTERBASE 394
INDICES EN MS SQL SERVER 394
INDICES EN ORACLE 395
CON QUÉ ÍNDICES PODEMOS CONTAR 396
ESPECIFICANDO EL ÍNDICE ACTIVO 398
ESPECIFICANDO UN ORDEN EN TABLAS SQL 399
BÚSQUEDA BASADA EN ÍNDICES 401
IMPLEMENTACIÓN DE REFERENCIAS MEDIANTE *FINDKEY* 403

Búsquedas utilizando *SetKey* 403
Experimentando con *SetKey* 404
¿Por qué existe *SetKey*? 405
Rangos: desde el Alfa a la Omega 406
El ejemplo de rangos de casi todos los libros 408
Más problemas con los índices de dBase 410
Cómo crear un índice temporal 411

20. Métodos de búsqueda 413

Filtros 413
Esto no lo dice la documentación… 414
Un ejemplo con filtros rápidos 415
El evento *OnFilterRecord* 418
Localización y búsqueda 419
Un diálogo genérico de localización 422
Filtros latentes 424
Filter By Example 426
Búsqueda en una tabla de detalles 430

PROGRAMACIÓN CON SQL 433

21. Breve introducción a SQL 435

La estructura de SQL 435
Para seguir los ejemplos de este libro… 436
La creación y conexión a la base de datos 438
Tipos de datos en SQL 439
Creación de tablas 440
Columnas calculadas 441
Valores por omisión 442
Restricciones de integridad 442
Claves primarias y alternativas 444
Integridad referencial 445
Acciones referenciales 446
Nombres para las restricciones 447
Definición y uso de dominios 448
Creación de índices 449
Modificación de tablas e índices 450
Creación de vistas 451
Creación de usuarios 451
Asignación de privilegios 453
Roles 454
Un ejemplo completo de *script* SQL 455

22. CONSULTAS Y MODIFICACIONES EN SQL **457**

LA INSTRUCCIÓN SELECT: EL LENGUAJE DE CONSULTAS 457
LA CONDICIÓN DE SELECCIÓN 459
OPERADORES DE CADENAS 459
EL VALOR NULO: ENFRENTÁNDONOS A LO DESCONOCIDO 460
ELIMINACIÓN DE DUPLICADOS 461
PRODUCTOS CARTESIANOS Y ENCUENTROS 462
ORDENANDO LOS RESULTADOS 464
EL USO DE GRUPOS 465
FUNCIONES DE CONJUNTOS 466
LA CLÁUSULA HAVING 467
EL USO DE SINÓNIMOS PARA TABLAS 467
SUBCONSULTAS: SELECCIÓN ÚNICA 468
SUBCONSULTAS: LOS OPERADORES *IN* Y *EXISTS* 469
SUBCONSULTAS CORRELACIONADAS 471
EQUIVALENCIAS DE SUBCONSULTAS 472
ENCUENTROS EXTERNOS 473
LAS INSTRUCCIONES DE ACTUALIZACIÓN 475
VISTAS 476

23. PROCEDIMIENTOS ALMACENADOS Y TRIGGERS **479**

¿PARA QUÉ USAR PROCEDIMIENTOS ALMACENADOS? 479
CÓMO SE UTILIZA UN PROCEDIMIENTO ALMACENADO 481
EL CARÁCTER DE TERMINACIÓN 482
PROCEDIMIENTOS ALMACENADOS EN INTERBASE 483
PROCEDIMIENTOS QUE DEVUELVEN UN CONJUNTO DE DATOS 486
RECORRIENDO UN CONJUNTO DE DATOS 488
TRIGGERS, O DISPARADORES 489
LAS VARIABLES *NEW* Y *OLD* 491
MÁS EJEMPLOS DE *TRIGGERS* 491
GENERADORES 493
SIMULANDO LA INTEGRIDAD REFERENCIAL 495
EXCEPCIONES 497
ALERTADORES DE EVENTOS 498

24. MICROSOFT SQL SERVER **501**

HERRAMIENTAS DE DESARROLLO EN EL CLIENTE 501
CREACIÓN DE BASES DE DATOS CON MS SQL SERVER 502
TIPOS DE DATOS PREDEFINIDOS 504
TIPOS DE DATOS DEFINIDOS POR EL PROGRAMADOR 505
CREACIÓN DE TABLAS Y ATRIBUTOS DE COLUMNAS 506
INTEGRIDAD REFERENCIAL 507
INDICES 508

SEGURIDAD EN MS SQL SERVER 508
PROCEDIMIENTOS ALMACENADOS 509
CURSORES 510
TRIGGERS EN TRANSACT-SQL 512
INTEGRIDAD REFERENCIAL MEDIANTE *TRIGGERS* 514

25. ORACLE 517

SOBREVIVIENDO A SQL*PLUS 517
INSTANCIAS, BASES DE DATOS, USUARIOS 519
TIPOS DE DATOS 520
CREACIÓN DE TABLAS 521
PROCEDIMIENTOS ALMACENADOS EN PL/SQL 522
CURSORES 524
TRIGGERS EN PL/SQL 525
SECUENCIAS 526
TIPOS DE OBJETOS 528
EXTENSIONES DE DELPHI PARA LOS TIPOS DE OBJETOS 532

26. USANDO SQL CON DELPHI 537

EL COMPONENTE *TQUERY* COMO CONJUNTO DE DATOS 537
¿QUIÉN EJECUTA LAS INSTRUCCIONES? 538
CONSULTAS ACTUALIZABLES 539
SIEMPRE HACIA ADELANTE 540
CONSULTAS PARAMÉTRICAS 543
CONSULTAS DEPENDIENTES 545
LA PREPARACIÓN DE LA CONSULTA 546
ACTUALIZACIÓN DE DATOS CON SQL 548
ALMACENAR EL RESULTADO DE UNA CONSULTA 549
¿EJECUTAR O ACTIVAR? 550
UTILIZANDO PROCEDIMIENTOS ALMACENADOS 553
VISUAL QUERY BUILDER 555

27. COMUNICACIÓN CLIENTE/SERVIDOR 559

NUESTRA ARMA LETAL: SQL MONITOR 559
¿TABLA O CONSULTA? 560
LA CACHÉ DE ESQUEMAS 562
OPERACIONES DE NAVEGACIÓN SIMPLE 563
BÚSQUEDAS EXACTAS CON *LOCATE* 564
BÚSQUEDAS PARCIALES 564
UNA SOLUCIÓN PARA BÚSQUEDAS PARCIALES RÁPIDAS 565
BÚSQUEDAS CON FILTROS LATENTES 567

ACTUALIZACIONES Y CONCURRENCIA 569

28. ACTUALIZACIONES 571

LOS ESTADOS DE EDICIÓN Y LOS MÉTODOS DE TRANSICIÓN 571
ASIGNACIONES A CAMPOS 572
CONFIRMANDO LAS ACTUALIZACIONES 574
DIFERENCIAS ENTRE *INSERT* Y *APPEND* 575
COMO POR AZAR… 576
MÉTODOS ABREVIADOS DE INSERCIÓN 577
ACTUALIZACIÓN DIRECTA VS VARIABLES EN MEMORIA 577
AUTOMATIZANDO LA ENTRADA DE DATOS 579
ENTRADA DE DATOS CONTINUA 581
ELIMINANDO REGISTROS 582
COMPRESIÓN DE TABLAS LOCALES 583

29. EVENTOS DE TRANSICIÓN DE ESTADOS 585

CUANDO EL ESTADO CAMBIA… 585
REGLAS DE EMPRESA: ¿EN EL SERVIDOR O EN EL CLIENTE? 586
INICIALIZACIÓN DE REGISTROS: EL EVENTO *ONNEWRECORD* 587
VALIDACIONES A NIVEL DE REGISTROS 588
ANTES Y DESPUÉS DE UNA MODIFICACIÓN 589
PROPAGACIÓN DE CAMBIOS EN CASCADA 591
ACTUALIZACIONES COORDINADAS MASTER/DETAIL 592
ANTES Y DESPUÉS DE LA APERTURA DE UNA TABLA 593
VACIANDO LOS *BUFFERS* 594
LOS EVENTOS DE DETECCIÓN DE ERRORES 595
LA ESTRUCTURA DE LA EXCEPCIÓN *EDBENGINEERROR* 597
APLICACIONES DE LOS EVENTOS DE ERRORES 600
UNA VEZ MÁS, LA ORIENTACIÓN A OBJETOS… 602

30. BASES DE DATOS Y SESIONES 603

EL COMPONENTE *TDATABASE* 603
OBJETOS DE BASES DE DATOS PERSISTENTES 604
CAMBIANDO UN ALIAS DINÁMICAMENTE 605
BASES DE DATOS Y CONJUNTOS DE DATOS 607
PARÁMETROS DE CONEXIÓN 609
LA PETICIÓN DE CONTRASEÑAS 609
EL DIRECTORIO TEMPORAL DE WINDOWS 611
PROBLEMAS CON LA HERENCIA VISUAL 611
SESIONES 613
CADA SESIÓN ES UN USUARIO 613
SESIONES E HILOS PARALELOS 614

Información sobre esquemas 617
El MiniExplorador de Bases de Datos 618
Gestión de alias a través de *TSession* 620
Directorios privados, de red y contraseñas 622

31. Transacciones y control de concurrencia 625

El Gran Experimento 625
El Gran Experimento: tablas locales 626
El Gran Experimento: tablas SQL 628
Pesimistas y optimistas 628
El modo de actualización 630
La relectura del registro actual 632
Las propiedades "ácidas" de una transacción 633
Transacciones SQL y en bases de datos locales 635
Transacciones implícitas y explícitas 636
Entrada de datos y transacciones 638
Aislamiento de transacciones 639
Aislamiento de transacciones mediante bloqueos 641
El jardín de los senderos que se bifurcan 643
De nuevo los optimistas 644

32. Actualizaciones en caché 649

¿Caché para qué? 649
Activación de las actualizaciones en caché 650
Confirmación de las actualizaciones 651
Marcha atrás 653
El estado de actualización 653
El filtro de tipos de registros 655
Un ejemplo integral 656
El Gran Final: edición y entrada de datos 658
Combinando la caché con grabaciones directas 660
Prototipos y métodos virtuales 663
Cómo actualizar consultas "no" actualizables 665
El evento *OnUpdateRecord* 668
Detección de errores durante la grabación 669

33. Libretas de bancos 673

Descripción del modelo de datos 673
Creación de vistas 675
Manteniendo actualizados los saldos 675
Libretas de banco y MS SQL Server 677
Ahora, en Oracle 679
El módulo de datos 681

GESTIÓN DE LIBRETAS 684
CÓDIGOS DE OPERACIONES 686
LA VENTANA PRINCIPAL 688
ENTRADA DE APUNTES 690
CORRIGIENDO EL IMPORTE DE UN APUNTE 692

LOS DETALLES FINALES 695

34. IMPRESIÓN DE INFORMES CON QUICKREPORT 697

LA HISTORIA DEL PRODUCTO 697
LA FILOSOFÍA DEL PRODUCTO 699
PLANTILLAS Y EXPERTOS PARA QUICKREPORT 700
EL CORAZÓN DE UN INFORME 701
LAS BANDAS 703
EL EVENTO *BEFOREPRINT* 704
COMPONENTES DE IMPRESIÓN 705
EL EVALUADOR DE EXPRESIONES 706
DEFINIENDO NUEVAS FUNCIONES 708
UTILIZANDO GRUPOS 709
ELIMINANDO DUPLICADOS 711
INFORMES *MASTER/DETAIL* 712
INFORMES COMPUESTOS 714
PREVISUALIZACIÓN A LA MEDIDA 715
LISTADOS AL VUELO 716
ENVIANDO CÓDIGOS BINARIOS A UNA IMPRESORA 718

35. GRÁFICOS DE DECISIÓN 721

GRÁFICOS Y BIORRITMOS 721
EL COMPONENTE *TDBCHART* 725
COMPONENTES NO VISUALES DE *DECISION CUBE* 727
REJILLAS Y GRÁFICOS DE DECISIÓN 729
MODIFICANDO EL MAPA DE DIMENSIONES 731

36. BIBLIOTECAS DE ENLACE DINÁMICO 733

ARQUITECTURA BÁSICA 733
PROYECTOS DLL 735
EXPORTACIÓN DE FUNCIONES 736
IMPORTACIÓN DE FUNCIONES Y UNIDADES DE IMPORTACIÓN 739
TIPOS DE DATOS EN FUNCIONES EXPORTADAS 740
CÓDIGO DE INICIALIZACIÓN Y FINALIZACIÓN 742
EXCEPCIONES EN BIBLIOTECAS DINÁMICAS 744
CARGA DINÁMICA DE BIBLIOTECAS 745

DEPURACIÓN DE BIBLIOTECAS DINÁMICAS 746
FUNCIONES DE USUARIO EN INTERBASE 746
BIBLIOTECAS DE RECURSOS E INTERNACIONALIZACIÓN 749
FICHEROS ASIGNADOS EN MEMORIA 752

37. SERVIDORES DE INTERNET 757

EL MODELO DE INTERACCIÓN EN LA WEB 757
APRENDA HTML EN 14 MINUTOS 758
EXTENSIONES DEL SERVIDOR Y PÁGINAS DINÁMICAS 761
¿QUÉ NECESITO PARA ESTE SEGUIR LOS EJEMPLOS? 762
MÓDULOS WEB 763
ACCIONES 765
RECUPERACIÓN DE PARÁMETROS 767
GENERADORES DE CONTENIDO 768
GENERADORES DE TABLAS 770
MANTENIMIENTO DE LA INFORMACIÓN DE ESTADO 771
UN EJEMPLO: BÚSQUEDA DE PRODUCTOS 773
EL MOTOR DE BÚSQUEDAS 775
CREANDO LA EXTENSIÓN WEB 778
GENERANDO LA TABLA DE RESULTADOS 780
DOCUMENTOS HTML Y SUSTITUCIÓN DE ETIQUETAS 781
RESPONDIENDO A LAS ACCIONES 783

38. CONJUNTOS DE DATOS CLIENTES 785

CREACIÓN DE CONJUNTOS DE DATOS 785
CÓMO EL *TCLIENTDATASET* CONSIGUIÓ SUS DATOS... 787
NAVEGACIÓN, BÚSQUEDA Y SELECCIÓN 788
FILTROS 789
EDICIÓN DE DATOS 790
CONJUNTOS DE DATOS ANIDADOS 791
CAMPOS CALCULADOS INTERNOS 794
INDICES, GRUPOS Y VALORES AGREGADOS 795

39. EL MODELO DE OBJETOS COMPONENTES: LA TEORÍA 799

COM, DCOM, OLE... 800
INTERFACES 801
LA INTERFAZ *IUNKNOWN* 803
IMPLEMENTACIÓN DE INTERFACES EN DELPHI 805
CÓMO OBTENER UN OBJETO COM 807
DENTRO DEL PROCESO, EN LA MISMA MÁQUINA, REMOTO... 808
EL HUEVO, LA GALLINA Y LAS FÁBRICAS DE CLASES 809
OLE Y EL REGISTRO DE WINDOWS 810
¿CÓMO SE REGISTRA UN SERVIDOR? 812

AUTOMATIZACIÓN OLE 812
CONTROLADORES DE AUTOMATIZACIÓN EN DELPHI 814
INTERFACES DUALES 815
EVENTOS Y CONTROLES ACTIVEX 816
BIBLIOTECAS DE TIPOS 817
ACTIVEFORMS: FORMULARIOS EN LA WEB 819

40. EL MODELO DE OBJETOS COMPONENTES: EJEMPLOS 823

CREACIÓN DE OBJETOS Y MANEJO DE INTERFACES 823
INTERCEPTANDO OPERACIONES EN DIRECTORIOS 824
INFORMES AUTOMATIZADOS 828
CONTROLADORES DE AUTOMATIZACIÓN 833
DECLARANDO UNA INTERFAZ COMÚN 835
UN SERVIDOR DE BLOQUEOS 836
LA IMPLEMENTACIÓN DE LA LISTA DE BLOQUEOS 838
CONTROL DE CONCURRENCIA 840
PONIENDO A PRUEBA EL SERVIDOR 843

41. MIDAS 845

¿QUÉ ES MIDAS? 845
CUÁNDO UTILIZAR Y CUÁNDO NO UTILIZAR MIDAS 847
MIDAS Y LAS BASES DE DATOS DE ESCRITORIO 849
MÓDULOS DE DATOS REMOTOS 850
PROVEEDORES 853
SERVIDORES REMOTOS Y CONJUNTOS DE DATOS CLIENTES 854
GRABACIÓN DE DATOS 856
RESOLUCIÓN 859
CONTROL DE ERRORES DURANTE LA RESOLUCIÓN 862
RECONCILIACIÓN 864
RELACIONES *MASTER/DETAIL* Y TABLAS ANIDADAS 866
ENVÍO DE PARÁMETROS 866
EXTENDIENDO LA INTERFAZ DEL SERVIDOR 867
LA METÁFORA DEL MALETÍN 870
TIPOS DE CONEXIÓN 871
BALANCE DE CARGA SIMPLE 874
INTERFACES DUALES EN MIDAS 875

42. CREACIÓN DE INSTALACIONES 877

LOS PROYECTOS DE INSTALLSHIELD EXPRESS 877
LA PRESENTACIÓN DE LA INSTALACIÓN 879
LAS MACROS DE DIRECTORIOS 881
GRUPOS Y COMPONENTES 881
INSTALANDO EL BDE Y LOS SQL LINKS 884

Configuración adicional del BDE 885
Instalación de paquetes 887
Interacción con el usuario 888
Las claves del registro de Windows 890
Cómo se registran los componentes ActiveX 891
Iconos y carpetas 892
Generando y probando la instalación 894
La versión completa de InstallShield Express 894
Las extensiones de InstallShield Express 895

Prólogo del Autor

And if the band you're in start playing different tunes
I'll see you on the dark side of the moon
Pink Floyd

…those dark and hideous mysteries which lie in the outer regions of the moon, regions which,
owing to the almost miraculous accordance of the satellite's rotation on its own axis
with its sidereal revolution about the earth, have never yet turned,
and, by God's mercy, never shall be turned, to the scrutiny of the telescopes of man.
Edgar Allan Poe, The Adventure Of One Hans Pfaall

EN LA PRIMAVERA DEL 95 APARECIÓ DELPHI. En aquel momento, el autor trabajaba para una empresa de cartografía, programando en C/C++ para el entorno Windows, y estaba metido en un proyecto bastante complicado. La aplicación en cuestión debía trabajar con grandes cantidades de datos, con relaciones complejas entre los mismos, y tenía que realizar cálculos algebraicos a diestra y siniestra; todo esto aprovechando la interfaz gráfica de Windows 3.11. De entrada descarté a Visual Basic, pues aunque simplificaba la programación de la interfaz de usuario, era demasiado lento para las operaciones de cálculo y, al no ser un lenguaje orientado a objetos, no me permitía modelar elegantemente las clases que manipulaba mi proyecto. En cuanto al acceso a datos, ya había trabajado con CodeBase, una biblioteca de funciones para manipular tablas en formato *dbf*. En un principio, podía utilizar algo así, aunque la versión 5.1, que era la que estaba utilizando, fallaba cuando los ficheros de datos pasaban de cierto volumen.

Necesitaba una herramienta de programación que cumpliera con estos requerimientos, y escribí a Borland, pensando en comprar un C++ con las Database Tools. En cambio, Borland me envió información sobre un nuevo producto llamado Delphi, cuyo lenguaje de programación parecía ser Pascal, ofrecía un entorno de desarrollo visual parecido al de Visual Basic, y que permitía el acceso a bases de datos en diversos formatos. No lo pensé mucho y compré el producto; tengo que reconocer que el precio, relativamente barato, también influyó en mi decisión. La historia tuvo, por supuesto, un final feliz como los de los anuncios de detergentes: el proyecto se terminó a tiempo, la aplicación se ejecutaba de maravillas y todos quedaron contentos, excepto yo, que tardé más de seis meses en cobrar lo que me correspondía.

Mi primera experiencia con Delphi fue similar a la de miles de programadores, y la conclusión que extraje de este primer encuentro con el lenguaje debe haber sido semejante a la de ellos: Delphi es un sistema muy fácil de aprender y utilizar, y ofrece una potencia de programación similar a la de los lenguajes tradicionales como C/C++.

No obstante, tras esta aparente sencillez se oculta una trampa. La programación con Delphi es, efectivamente, muy sencilla si nos limitamos a las tareas básicas. Pero dominar el lenguaje requiere tiempo y dedicación. Los tres problemas básicos a los que se enfrenta el programador en Delphi son:

- La Programación Orientada a Objetos.
- La arquitectura basada en mensajes o eventos.
- El control de errores mediante excepciones.

Además, el programador que quiere desarrollar aplicaciones serias de bases de datos tiene que afrontar las dificultades de la programación concurrente: mecanismos de control de acceso, transacciones, integridad y consistencia, el dominio del lenguaje SQL... Si el entorno del cual procede es la programación para bases de datos locales, todos estos conceptos son nuevos o se abordan desde enfoques diferentes en su sistema de programación original.

Por todas estas razones, decidí escribir un libro dedicado especialmente a la programación para bases de datos con Delphi, que entrara en profundidad en los conceptos específicos de este área. Un libro de carácter "tutorial", que además considerase todos los temas anteriores con la profundidad necesaria, hubiera tenido el tamaño de una enciclopedia, y su redacción necesitaría mucho más tiempo. En vez de llevar una línea de desarrollo, conducida quizás por una aplicación única de ejemplo, he preferido utilizar ejemplos pequeños para cada tema. De este modo, cada capítulo puede leerse de forma independiente a los demás, y el libro puede utilizarse como referencia.

El mayor problema de un libro escrito en ese estilo es el de las *referencias circulares*: necesito explicarle qué es un campo antes de poder hacer ejemplos sencillos con las tablas, pero también necesito presentar las tablas antes de poder explicar los campos. Y la única solución que se me ocurre es ofrecer *adelantos*: escribir un poco sobre el recurso necesario, prometiendo más información sobre éste más adelante.

"La Cara Oculta de Delphi" está destinada a personas que *ya* han estado sentadas frente a Delphi. No es necesario que hayan *programado* en Delphi, pero no es mi intención enseñar cómo se inicia Delphi, qué comandos ofrece, qué es un menú... Hay bastante literatura por ahí que se ocupa de estos temas. En particular, me permito recomendar el "Manual Fundamental de Borland Delphi 2.0", de un tal Ian Marteens; no será el mejor, pero conozco al autor y no es mala persona.

Convenios sintácticos

A lo largo de este libro tendré que explicar la sintaxis de algunas construcciones de Delphi, especialmente en el área de objetos y clases, y de las instrucciones del lenguaje SQL. Utilizaré en estos casos la notación habitual, que explico a continuación para los lectores no familiarizados:

Cursivas	Nombres o constantes que el programador debe suministrar.
[Corchetes]	Lo que se pone entre corchetes es opcional.
Puntos suspensivos...	Los puntos suspensivos indican la repetición de la cláusula anterior.
Verdadero|Falso	La barra vertical se utiliza para indicar una alternativa.

Acerca de la terminología empleada

La mayor parte de este libro ha sido redactada directamente en mi (pobre) castellano. He tratado, en lo posible, de mantener los nombres originales de los objetos y conceptos utilizados. En los casos en que he traducido el término, incluyo también su nombre en inglés. No tiene sentido traducir términos nuevos o que se utilizan con poca frecuencia; el próximo autor posiblemente utilizará una traducción diferente, aunque sea por ansias de originalidad.

Durante la corrección del libro, he insistido en dos traducciones que pueden resultar extrañas para el lector español. Una de ellas es traducir *library* como *biblioteca*, cuando el término más extendido es *librería*. La otra "singularidad" es utilizar el género femenino para la palabra *interfaz*. El castellano moderno tiende a masculinizar las palabras importadas de otro idioma, así que casi todos dicen en España: "el interfaz". Pero esta palabra no es realmente una importación; se deriva de *faz*, que es de género femenino.

Agradecimientos

En primer lugar, quiero agradecer la acogida que tuvo el "Manual Fundamental". Sin el apoyo del público, no me hubiera decidido a escribir un segundo libro acerca de Delphi. Por otra parte, gracias a los que compraron el libro, Christine (mi ordenador personal) pasó de ser un 486 a 100MHz con 12MB, a convertirse en un Pentium potente de mente fría y corazón caliente; vamos, que le habéis financiado el *lifting*. Quiero decir, no obstante, que antes de la actualización Delphi 3 podía ejecutarse en mi máquina; no muy felizmente, pero podía.

Durante la redacción de "La Cara Oculta…" tuve el placer de impartir unos cuantos cursos de programación con Delphi. En cada curso aprendí algo, por lo que es de justicia mencionar a todos los profesionales y estudiantes que asistieron a los mismos.

Dos personas han resultado ser imprescindibles en la revisión del libro. Nuevamente, Octavio Hernández ha tenido que soportar largas charlas cada vez que se me ocurría "una idea genial". Muchos de los ejemplos que utilizo son fruto de su imaginación; además, le debo el título del libro. Octavio, desgraciadamente, no comparte mi aversión a C++: nadie es perfecto. Y Marta Fernández ha tenido el mérito de hacerme comprender que el sentido del humor no es mi punto fuerte. Si este libro resulta al final "legible" e "inteligible" es gracias a ella. A propósito, Marta tampoco es perfecta: le he dicho que si acertaba con la lotería le daría la mitad, y no se lo ha creído.

Finalmente, de no ser por Danysoft International, este libro no hubiera visto la luz, literalmente. Estuvieron las copias de evaluación, las horas de prácticas en red con los más diversos sistemas de bases de datos ... y el apoyo personal de Martha y José Luis Castaño al autor, todas las veces que éste pensó que hacer libros de Informática en España era como escribir frases en la arena de una playa.

A todos, gracias.

Ian Marteens
Pamplona, Agosto de 1997

Prólogo a la Segunda Edición

Uno escribe un libro cuando siente que tiene algo interesante que contar ... y claro, cuando hay alguien dispuesto a leerlo, y tienes el tiempo disponible, y alguien te ofrece dinero por hacerlo ... Uno escribe una segunda edición cuando se da cuenta de todo lo que faltaba en la primera. Así que quiero dar las gracias en primer lugar a todos los lectores de la primera edición, por la cálida acogida que le dieron a este libro, y por hacerme llegar todos sus consejos, sugerencias y comentarios.

Un libro de este tamaño no puede ser obra exclusiva de la experiencia de una sola persona. Aquí está reflejado el trabajo de muchos programadores que tuvieron a bien compartir sus conocimientos conmigo. Me gustaría mencionarlos a todos, pero la lista es larga y temo dejarme a alguien en el tintero. Gracias de todos modos.

Agradecimientos especiales a Octavio Hernández, que ha sufrido la revisión de estas páginas sin saltarse un capítulo. Después de ver todos los errores que ha encontrado, he decidido matricularme en clases nocturnas de Expresión Escrita. Como único podré pagárselo es revisando sus tres próximos libros.

Para más ejemplos y actualizaciones...

Es inevitable que se quede algo en el tintero al terminar un libro. Por este motivo, he decidido mantener una página Web en la dirección **www.marteens.com**. Consulte periódicamente esta página para nuevos ejemplos y ampliaciones del material cubierto en este libro.

Ian Marteens
Madrid, Agosto de 1998

Who is the third who walks always beside you?
When I count, there are only you and I together
But when I look ahead up the white road
There is always another one walking beside you
Gliding wrapt in a brown mantle, hooded
I do not know whether a man or a woman,
— But who is that on the other side of you?

T.S. Eliot, The Waste Land

1

Entorno y Herramientas

- ¿Es Delphi su lenguaje?
- Herramientas y utilidades
- Unidades, proyectos y paquetes
- Sistemas de bases de datos
- El Motor de Datos de Borland

Parte

Capítulo

1

¿Es Delphi su lenguaje?

ESTIMADO LECTOR: si está leyendo estas líneas, es muy probable que haya comprado el libro y tenga claro que la mejor forma de abordar su próximo proyecto de programación es utilizar Delphi. Puede también que hayan impuesto Delphi en su departamento y no le quede otro remedio que aprenderlo lo mejor posible. Pero cabe la posibilidad de que esté hojeando este libro en la tienda, y que todavía tenga dudas acerca de cuál es el lenguaje de programación que más le conviene para sus aplicaciones de bases de datos.

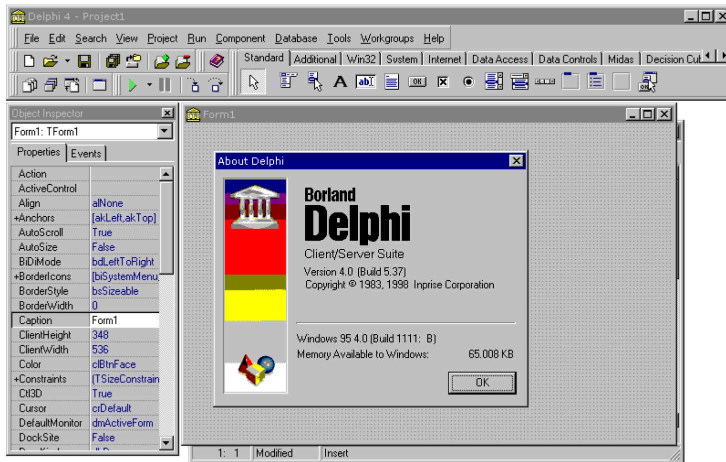
Si está usted convencido de que Delphi es el lenguaje que necesita, adelante: sáltese esta introducción y entre en materia en el siguiente capítulo. Para los que decidan seguir leyendo, describiré en este capítulo las principales características de Delphi como herramienta de desarrollo, comparándolo con otros lenguajes y entornos cuando sea necesario. Espero entonces que este breve resumen le sea de utilidad.

Breve historia de un lenguaje llamado Delphi

Delphi 1 salió al mercado en la primavera del 95. Estaba diseñado para ejecutarse en Windows 3.1, aunque era compatible con las versiones beta de Windows 95, que aún no se había comercializado. Fue el primer entorno de desarrollo RAD que ofrecía un compilador de código nativo. El lenguaje en que se basaba, como todos sabemos, era Pascal; de hecho, internamente el compilador definía directivas de compilación como *VER80*, para indicar que se consideraba la versión 8 de Turbo Pascal. Se comercializaron dos versiones: la Desktop y la Client/Server. Ambas incluían un producto de Borland que ya se distribuía con ciertas versiones del Borland C++: el Motor de Bases de Datos de Borland, o *Borland Database Engine* (BDE). Este era un conjunto de DLLs que permitían el acceso funcional a dBase, Paradox, fuentes de datos ODBC y diversos sistemas SQL: Oracle, InterBase, Informix y Sybase. En la versión Desktop, sin embargo, no se incluían los controladores necesarios para acceder a bases de datos en un servidor, conformándose con las bases de datos locales y ODBC.

Aunque en el momento de la aparición de Delphi 1 Visual Basic llevaba unos cuantos años en el mercado, la nueva herramienta tuvo una clamorosa acogida en la comunidad de programadores. Por primera vez se alcanzaban velocidades de ejecución profesionales en programas desarrollados con el paradigma RAD. Por primera vez, un lenguaje RAD era totalmente extensible, permitiendo sin ningún tipo de maniobras de distracción el acceso al temido API de Windows. Por primera vez, un lenguaje basado en componentes permitía crear también esos componentes, sin necesidad de recurrir a otro entorno de programación

La siguiente versión de Delphi apareció también en la primavera (¡qué romántico!) del 96. El principal avance fue la conversión del compilador para generar aplicaciones de 32 bits, para Windows 95 (que todos pensábamos que iba a ser Windows 96) y para Windows NT. Además, se incluyó una fase de optimización de código, todo un lujo para un lenguaje compilado, mientras Visual Basic 4 seguía generando código interpretado. La metodología de programación experimentó mejoras importantes: la inclusión de módulos de datos, el Depósito de Objetos, el Diccionario de Datos, la herencia visual, etc. Y, gracias a la retroalimentación por parte de los programadores, los componentes existentes se hicieron más potentes y manejables. Se incluyó también la posibilidad de crear controladores y servidores de automatización, para aprovechar las nuevas características de OLE de Windows 95 y NT. El Motor de Datos se aumentó con un controlador para DB2. Posteriormente, en ese mismo año, se realizó una actualización de Delphi con el número de versión 2.01 que, además de corregir errores, incluyó el Internet Solutions Pack, un conjunto de controles ActiveX para aplicaciones de Internet.



En mayo del 97 (¡también primavera!), después de una tensa espera, apareció la versión 3. Las novedades se centraron en tres campos: la tecnología de paquetes (*packages*), el aprovechamiento de la tecnología OLE (ActiveX, automatización, bases de datos en múltiples capas) y la programación para servidores Web. El objetivo global

de todas estas incorporaciones fue convertir a Delphi en un verdadero entorno de desarrollo cliente/servidor, permitiendo la programación de aplicaciones que se ejecutan en el servidor. El Motor de Datos siguió añadiendo formatos: en aquel momento le tocó a Access y a FoxPro. Por otra parte, la arquitectura de acceso a datos fue modificada para facilitar el desarrollo de componentes de acceso a datos que dejaran a un lado el BDE. En esta línea, Borland lanzó Delphi/400, para acceder a bases de datos de AS/400, que utilizaba la técnica de saltarse el BDE.

Al poco tiempo, Borland lanzó una versión de actualización de Delphi, con el número de serie 3.01. Además de la corrección de *bugs*, incorporaba un nuevo componente, *TMIDASConnection*, para permitir el desarrollo de aplicaciones en múltiples capas en redes TCP/IP. Con este componente, se podía evitar la instalación y configuración de DCOM en redes mayoritariamente formadas por terminales Windows 95. También hubo un parche, el 3.02, que corrigió algunos problemas relacionados con la tecnología Midas y con los dichosos controles del Internet Explorer, que iban cambiando de versión mes tras mes. Una versión intermedia del BDE, la 4.51 que apareció con Visual dBase 7, añadió el soporte nativo para Access 97, pues la versión incluida en Delphi solamente soportaba Access 95.

Finalmente, en julio del 98, Delphi 4 vio la luz. ¡En verano, no en primavera, por primera vez! Delphi 4 ha mejorado ostensiblemente el entorno de programación y desarrollo, al añadir el Explorador de Código, las facilidades de completamiento de clases, etc. Hay mejoras en el lenguaje, pero usted no las notará en su programación cotidiana. Hay novedades en los controles visuales, como el soporte de *dockable windows* (ponga aquí su traducción preferida) y las listas de acciones. En el apartado de bases de datos, se ha añadido soporte para las extensiones de objetos de Oracle 8 (¡la única herramienta RAD que ofrece este soporte, por el momento!) y se ha oficializado el soporte de Access 97. La tecnología Midas se ha beneficiado de adelantos impresionantes, tanto en estabilidad como en posibilidades de conexión: ahora es posible utilizar CORBA como mecanismo de enlace en aplicaciones multicapas y hay soporte para Microsoft Transaction Server. También se ha mejorado el sistema de base de datos en el cliente de Midas, con la posibilidad de utilizar tablas anidadas, campos agregados, etc. No estamos, sin embargo, ante cambios revolucionarios: es muy difícil mejorar algo que ya era muy bueno.

¿Es Delphi un lenguaje "sencillo"?

La respuesta al título de la sección depende de lo que usted quiera hacer con él. ¿Quiere montar un mantenimiento simple, con objetos que se pueden representar en registros independientes? En ese caso, sí, es muy fácil. ¿Quiere trabajar sobre un esquema relacional complejo, con referencias complicadas entre tablas, estructuras jerárquicas y todo eso? Bueno, con un poco de disciplina, la aplicación se desarrolla en poco tiempo; es fácil plantearse una metodología de trabajo, y el resultado será un

programa "robusto", que es algo de agradecer. ¿Quiere que el programa anterior sea independiente del formato de bases de datos subyacente, que se acople lo mismo a sistemas locales, como Paradox y dBase, que a Oracle, InterBase o MS SQL Server? Entonces, amigo mío, necesita leer este libro, para que sepa en qué terreno se ha metido.

El problema no reside en Delphi, pues las mismas dificultades se le presentarán con C++ Builder, Visual Basic ó PowerBuilder. La causa de las complicaciones es, a mi entender, el modelo de programación y acceso concurrente que es necesario dominar para este tipo de aplicaciones. El programador de bases de datos locales (léase Clipper, Paradox, los distintos COBOL para PCs, etcétera) aborda las cuestiones de concurrencia imponiendo bloqueos en cada registro antes de editarlos. En el 95 por ciento de los casos, no se plantea la garantía de atomicidad de las actualizaciones compuestas, ni la recuperación en transacciones, ni el elevar al máximo la concurrencia de la aplicación (conozco aplicaciones de facturación escritas en Clipper "para red" que sólo dejan facturar desde un puesto a la vez). Por supuesto, al programador que solamente ha trabajado en este tipo de entorno le cuesta más acostumbrarse a distinguir entre bloqueos optimistas, pesimistas y arquitecturas multigeneracionales. Este libro tiene el propósito de ayudar en el aprendizaje de estos conceptos.

En las secciones que vienen a continuación analizaremos en detalle cada una de las principales características de Delphi.

Programación orientada a objetos verdadera

A 28 años de la definición del primer lenguaje de programación orientado a objetos, y a casi 10 años de la popularización de sus conceptos con la aparición de C++, nadie duda ya de las grandes ventajas aportadas por este estilo de programación y pensamiento. A pesar de esto, muchos de los lenguajes populares RAD no soportan verdaderamente la programación orientada a objetos. Para que un lenguaje sea orientado a objetos debe dar soporte a estas tres características:

1. Encapsulación
2. Herencia
3. Polimorfismo

En particular, Visual Basic no permite la herencia, en ninguna de sus versiones. VB está basado completamente en el modelo COM, propuesto por Microsoft. En este modelo, el concepto fundamental es el de interfaz: un conjunto de funciones que deben ser implementadas por cada objeto perteneciente a una clase determinada. Otros modelos de objetos, como CORBA y JavaBeans, admiten que una clase herede las características de otras, lo cual no sucede en COM.

Delphi incorpora un modelo completo de programación orientada a objetos, incluyendo encapsulación, herencia simple y polimorfismo. En este sentido Delphi se puede comparar con lenguajes que son paradigmas de ese estilo de programación, como C++. Por ejemplo, aunque Delphi no implementa la herencia múltiple al estilo C++, la versión 3 incluye en compensación el concepto de *interfaces*, que se encuentra en lenguajes modernos como Java, y que puede utilizarse como mecanismo alternativo en la gran mayoría de las aplicaciones prácticas de la herencia múltiple. El soporte de interfaces hace posible la programación de objetos COM, controles ActiveX, automatización OLE, etc.

Por ser un lenguaje desarrollado cuando los conceptos de Programación Orientada a Objetos han alcanzado su madurez, Delphi ofrece técnicas de programación que no se encuentran en lenguajes más antiguos como C++. Por ejemplo, Delphi implementa como parte fundamental de su arquitectura los constructores virtuales, métodos dinámicos, manejadores de mensajes y un estilo mucho más "humano" y fácil de trabajar con la Información de Tipos en Tiempo de Ejecución (*RTTI*). Estas extensiones se han incorporado también al nuevo C++ Builder, de Inprise Corporation, la compañía antes conocida como Borland International[1].

Arquitectura integrada de componentes

Una de las principales características de Delphi es la arquitectura de sus componentes, que permite una completa integración de estos dentro del lenguaje. Los componentes de Delphi están basados en clases desarrolladas en el propio lenguaje; no hay que utilizar otros entornos de programación para crear o extender componentes. Otra consecuencia es que el código de estas clases se ejecuta dentro del mismo espacio de direcciones de la aplicación, con la consiguiente ventaja en necesidades de memoria y tiempo de ejecución. Al conjunto de componentes básicos de Delphi se le denomina la *Biblioteca de Componentes Visuales*; en inglés, *Visual Component Library*, ó VCL.

Visual Basic utiliza como componentes los llamados controles ActiveX, que Delphi también admite. Desde el punto de vista de su implementación, los componentes ActiveX son objetos COM implementados en bibliotecas de enlace dinámico, o DLLs. Un objeto COM (*Component Object Model*), a su vez, es un objeto con un formato binario estándar definido por Microsoft, que permite que diferentes lenguajes de programación hagan uso de las propiedades, métodos y eventos del mismo. Esta es una buena idea, que ahora llega a su madurez al implementarse el llamado DCOM, o *Distributed COM*, que permite la activación y uso de objetos remotos en red.

---

[1] ¿Os imagináis a Mr. Yocam cantando *Purple Rain* en la próxima conferencia de Borland ... ejem, perdón ... de Inprise?

Sin embargo, este modelo tiene sus luces y sus sombras. Anteriormente hemos mencionado uno de sus defectos: no hay soporte directo de herencia, lo cual limita la extensibilidad de las clases. Hay otro inconveniente importante de orden práctico: los controles ActiveX se implementan como DLLs de Windows. Una DLL es, desde el punto de vista del enlazador (*linker*), un sistema autoconsistente; esto quiere decir que si implementamos un control ActiveX en C, el control debe cargar con el *runtime* de C; si lo implementamos en Delphi o VB5/6, la librería básica de estos lenguajes deberá estar presente en tiempo de ejecución. Si nuestra aplicación está basada completamente en este tipo de recursos y tenemos que utilizar (como es lo típico) unas cuantas decenas de ellos procedentes de las más diversas fuentes, piense en el gasto de recursos en que incurriremos.

Por el contrario, los componentes VCL se integran directamente dentro de los ejecutables producidos por Delphi y por C++ Builder. No tenemos que cargar con copias duplicadas de una biblioteca de tiempo de ejecución. Las llamadas a métodos, propiedades y eventos transcurren de modo más eficiente, pues no tienen que pasar por la capa OLE. Además, son completamente extensibles por medio de la herencia.

Delphi genera controles ActiveX

A pesar de todo lo comentado en la sección anterior, Delphi ofrece, a partir de la versión 3, lo mejor de ambos mundos: los controles VCL pueden ser convertidos, mediante un sencillo paso automático, en controles ActiveX. Hasta el momento, el mercado de componentes ha estado orientado a la producción de controles ActiveX; si tanto Delphi como VB podían utilizar estos controles, y VB no tenía capacidad para trabajar con controles VCL, era evidente que los fabricantes de componentes producirían controles compatibles con estos dos lenguajes. Normalmente, estos controles se programaban en C/C++, y todo el mecanismo de generación de interfaces y tablas de métodos virtuales era manual y tedioso.

Al introducir el concepto de interfaces, Delphi da soporte directo a la programación de objetos COM. El proceso de implementar una interfaz consiste en declarar una clase que incluya en la lista de ancestros la interfaz que se va a implementar, y en suministrar cuerpos para los métodos de la interfaz. Delphi tiene clases predefinidas que proporcionan la mayoría de los métodos requeridos por los controles ActiveX y otras interfaces COM comunes, además de expertos que facilitan el desarrollo.

Tratamiento de excepciones

El tratamiento de errores mediante excepciones es la alternativa moderna al tratamiento tradicional mediante códigos de retorno de errores. Gracias a esa técnica, se evita mezclar la lógica de control del algoritmo con las instrucciones de salto de la

detección de errores por medio de códigos de estado. Tanto Delphi como VB implementan excepciones, pero solamente Delphi ofrece un conjunto de instrucciones estructuradas correctamente desde el punto de vista del diseño de lenguajes. El diseño de las excepciones de VB, por el contrario, está basado en el sistema tradicional de Basic (¡que no es un lenguaje estructurado!) y sufre la influencia del carácter interpretado del código generado, en las versiones anteriores a la actual.

Velocidad de ejecución

Delphi, con mucho, sigue siendo el más rápido de los lenguajes RAD. En pruebas realizadas por una prestigiosa revista de Informática, se constató que Delphi es de 3 a 6 veces más rápido que Visual Basic 5 a pesar de que, gracias al compilador incorporado en la esa versión, los programas de VB5 pueden ejecutarse hasta 20 veces más rápido que antes (según la propia Microsoft). La versión 4 de Delphi es la tercera versión de este producto que genera código de 32 bits, por lo cual estamos ante un compilador más estable y confiable. Según pruebas realizadas con Delphi 4 y Visual Basic 6, se sigue manteniendo la proporción 5:1 en las velocidades de ejecución de los programas desarrollados con estos sistemas.

Delphi comparte la misma etapa final de optimización y generación de código nativo que la línea de compiladores de C/C++ de Borland. Las optimizaciones implementadas por estos compiladores son muy sofisticadas. Se incluye la detección de subexpresiones comunes a nivel local y global, la optimización de registros, la detección de invariantes de bucles, etc.

En el área de la Automatización OLE, es de destacar que a partir de la versión 3 se puede trabajar con las interfaces de objetos COM utilizando la *v-table*: la tabla de punteros a funciones de la interfaz. Esta mejora a OLE permite alcanzar mayor velocidad en las llamadas a funciones en objetos con interfaces duales, en contraste con la técnica anterior, que pasaba por invocar una función del API de Windows.

Velocidad de compilación y enlace

Uno de los puntos fuertes de Delphi es la velocidad con que se compilan y enlazan las aplicaciones. En este aspecto no tiene parangón con los sistemas competidores. Y es que la velocidad de compilación es muy importante, tratándose de sistemas de diseño y programación interactivos. El formato de unidades de compilación nativo de Delphi, el formato *dcu*, permite alcanzar mayores velocidades de compilación y enlace al estar basado en el formato *obj*, especificado por Intel hace ya muchos años, con la mente puesta en técnicas obsoletas de enlace. No obstante, Delphi acepta y genera ficheros *obj* (a partir de la versión 2), incluso si han sido generados por otros lenguajes. Delphi ofrece un enlazador "inteligente", que es capaz de eliminar todo el

código que no va a ser utilizado por la aplicación. ¿Imagina las consecuencias? Aplicaciones más compactas, que se cargan más rápido y que ocupan menos espacio en memoria RAM y en el disco.

A partir de la versión 3 de Delphi, los tiempos de enlace disminuyen aún más gracias a los *packages*. Estos *packages*, o paquetes, se pueden utilizar opcionalmente, y son DLLs que contienen el código de componentes. De esta forma, dos aplicaciones escritas en Delphi que se ejecutan en paralelo en la misma máquina utilizan una sola copia en memoria del código de las librerías de tiempo de ejecución del lenguaje. Como consecuencia, el tamaño físico de los ejecutables disminuye dramáticamente, junto con el tiempo invertido por el enlazador en generar el fichero.

De forma adicional, el Entorno de Desarrollo es mucho más rápido y eficiente. Este programa está escrito en Delphi (¡hay que predicar con el ejemplo!) y utiliza la VCL. Antes, la librería de componentes, la *complib.dcl* (16 bits) ó *complib32.dcl* (32 bits), y el Entorno de Desarrollo utilizaban copias diferentes de la VCL. Ahora, gracias a los paquetes, este código se comparte, de modo que disminuyen las necesidades de memoria. A esto añádale que cuando se depura una aplicación nos ahorramos otra copia de la VCL en memoria.

Retroalimentación inmediata

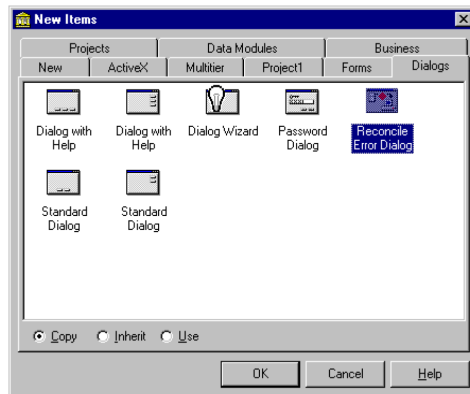
Se supone que uno de los objetivos fundamentales de un sistema RAD es permitir que el programador pueda hacerse una idea, mientras desarrolla su aplicación, del aspecto que va a tener la misma, sin necesidad de efectuar el tedioso ciclo de compilación/enlace/puesta a punto. ¿Ha visto alguna vez la rejilla de datos de Visual Basic durante el tiempo de diseño? Este control no es capaz de mostrar la disposición visual que tendrá en tiempo de ejecución, por lo que el programador debe trabajar a ciegas. Esto, hay que aclarar, no es una consecuencia del lenguaje en sí, sino de un mal diseño de la librería. En cambio, todos los controles de bases de datos de Delphi ofrecen una retroalimentación inmediata, en tiempo de diseño, de su aspecto final. Incluso al establecer relaciones *master/detail* entre varias tablas, es posible explorar todas ellas de forma sincronizada, utilizando el Editor de Campos. Del mismo modo, cualquier modificación en el formato de campos se refleja inmediatamente en los controles visuales afectados, e incluso los campos de búsqueda (*lookup fields*), que permiten representar referencias entre tablas, se pueden visualizar durante el diseño de la aplicación.

El Depósito de Objetos y la herencia visual

La herencia visual es una potente técnica disponible cuando utilizamos plantillas del Depósito de Objetos de Delphi. Es una técnica dirigida tanto a desarrolladores aisla-

dos como a grupos de trabajo. El comportamiento típico de otros entornos de desarrollo en relación con el uso de plantillas y expertos, consiste en que la aplicación que solicita una plantilla obtenga una copia local del código necesario. Si posteriormente se realizan modificaciones en la plantilla, estos datos no se propagan a la copia local, por lo cual es necesario repetir los cambios en esta copia, o comenzar nuevamente desde cero.

Delphi implementa un Depósito de Objetos, en el cual pueden colocarse plantillas desarrolladas por nosotros mismo e incluso asistentes desarrollados en el propio lenguaje. Delphi permite, al traer plantillas desde el Depósito, hacer una copia, utilizar el original o crear un objeto por herencia a partir del objeto plantilla. Incluso permite llamadas a eventos heredados con una sintaxis simple y sin demoras en tiempo de ejecución.



Por supuesto, Visual Basic no implementa ningún tipo de herencia, mucho menos la herencia visual.

Aplicaciones distribuidas

Quizás la novedad más importante que introdujo Delphi 3, y que sigue rindiendo frutos en la versión 4, es la posibilidad de crear aplicaciones de bases de datos multicapas, en las que todas las peticiones de datos se dirigen a un servidor de aplicaciones remoto. Este servidor se comunica con los clientes utilizando DCOM, OLEnterprise (un producto cuya versión de evaluación se incluye con Delphi), TCP/IP o CORBA. El Motor de Datos de Borland reside, en esta configuración, en el ordenador en que se ejecuta este servidor remoto. Si vamos a utilizar DCOM como protocolo de transporte, por ejemplo, el ordenador que actúa como servidor debe ejecutar Windows NT 4 o Windows 95 con los parches DCOM que ofrece Microsoft en su Web, o el nuevo Windows 98. Los clientes, o estaciones de trabajo, no necesitan el Motor de Datos, ahorrándose el desarrollador la instalación y configuración del

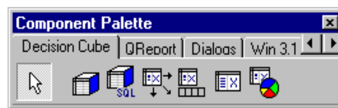
mismo. ¡Piense cuanto tiempo se ahorra si se trata de instalar una aplicación de bases de datos con decenas de usuarios accediendo a la misma! El conjunto de técnicas que hacen posible esta comunicación se conoce como *Midas* (*Multi-tiered Distributed Application Services*), y le dedicaremos todo un capítulo en este libro.

A propósito del Motor de Datos, ahora Delphi puede acceder directamente a bases de datos de Access y FoxPro, si existe una copia del JetEngine de Microsoft en el ordenador. Se siguen soportando todos los formatos anteriores: Paradox, dBase, InterBase, Oracle, Informix, DB2, Sybase y MS SQL Server. El Motor de Datos ha sido mejorado para permitir la adición de nuevos controladores SQL-Links, y la arquitectura de la VCL ha sido modificada para permitir la incorporación incruenta de sustitutos del BDE, tales como Titan, Apollo, etc.

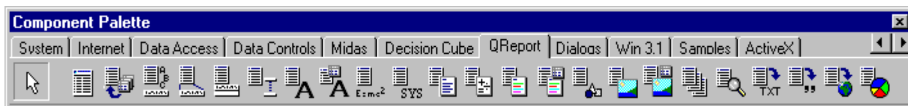
Componentes de Delphi

Por último, se han incorporado en Delphi componentes para dar respuesta a las principales necesidades de las aplicaciones de gestión: gráficos, tablas cruzadas para soporte de decisiones, informes. ¿Gráficos tridimensionales? No hay problema. ¿Los quiere en pantalla?, tenemos *TChart*. ¿Alimentados por una base de datos?, entonces necesita *TDbChart*. ¿En un informe?, para eso está *TQrChart*. Y no tema, que estos controles son componentes VCL, nada de controles ActiveX ni cosas semejantes.

Una nueva página, *Decision Cube*, contiene componentes para sofisticados análisis de datos dinámicos. Estos componentes serán fundamentales en la programación de aplicaciones de negocios, que ofrezcan posibilidades de análisis multi-dimensional, lo que comúnmente se clasifica como técnicas de *data warehousing*.



QuickReport, la herramienta de creación e impresión de informes, también ha sido mejorada y aumentada. Ahora es más fácil el diseño de la página de impresión, con un componente más "visual", y un editor de propiedades más amigable. Se han incluido componentes para imprimir imágenes de forma más rápida y conveniente, texto con formato e incluso gráficos de negocios. También se puede guardar el resultado en formato HTML, para ser distribuido en la Web.



La versión cliente/servidor de Delphi permite, finalmente, la programación de extensiones para servidores Web. De este modo, es muy sencillo exportar datos que pueden ser visualizados en Internet, sin necesidad de comprar herramientas adicionales.

En esta misma línea se incluye un amplio conjunto de componentes para el desarrollo de todo tipo de aplicaciones clientes de Internet: correo electrónico, clientes de FTP, visores para el formato HTML, etc.

Capítulo

2

Herramientas y utilidades

EL PROPÓSITO DE ESTE BREVE CAPÍTULO es presentar las herramientas que acompañan a Delphi, y las novedades del propio Entorno de Desarrollo dignas de mención en la versión 4. No es mi intención entrar en detalles con estas herramientas: mejor que leer una larga parrafada, es preferible familiarizarse con las mismas mediante la práctica.

Delphi y el teclado

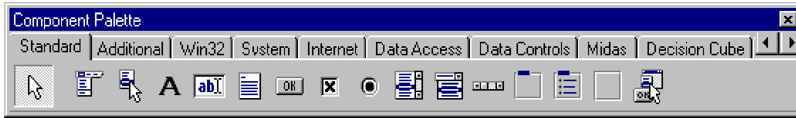
Nadie discute las ventajas de alcanzar una buena velocidad de edición y diseño con cualquier herramienta de programación. Es doloroso, en consecuencia, ver a ciertos programadores de Delphi sufrir calladamente mientras realizan determinadas operaciones que desconocen que se pueden realizar de una forma más sencilla. En esta sección menciono algunos trucos para acelerar el trabajo con el Entorno de Desarrollo.

Comencemos por la adición de componentes a un formulario. Delphi ofrece, que yo conozca, cuatro formas diferentes de traer un componente a un formulario, aunque muchos programadores solamente aprovechen una o dos:

- *Realizar un doble clic sobre el componente en la Paleta*: Se añade un nuevo componente con el tamaño predeterminado y situado *sobre* el centro del componente activo del formulario. Supongamos que tiene un panel que ocupa todo el formulario (*Align=alClient*), y que quiere añadir una barra de mensajes debajo del panel. Si pincha la barra sobre el panel, la barra pertenecerá al panel, no al formulario. La solución es seleccionar el formulario (ver truco con la tecla ESC más adelante), ir al botón *TStatusBar* y hacer el doble clic.
- *Realizar un clic en la Paleta y un clic sobre el futuro padre del componente*: El componente se crea en el sitio indicado, pero con un tamaño predeterminado. Es la mejor forma de colocar botones, cuadros de edición y otros componentes que se benefician con una apariencia estándar.
- *Realizar un clic en la Paleta, y arrastrar sobre el padre para definir el tamaño*: Este método ahorra tiempo cuando se va a utilizar un tamaño diferente del predefinido. Es el

método soportado por Visual Basic, y el único que muchos programadores principiantes utilizan.

- *Pulsar la mayúscula y realizar un clic en la Paleta*: Con este método, el botón de la Paleta queda seleccionado, aún después de añadir el componente. Así se pueden traer varias instancias del mismo componente con menos clics del ratón. Para volver al modo de selección, hay que pulsar el botón con la flecha de cursor que aparece a la izquierda de la Paleta de Componentes.



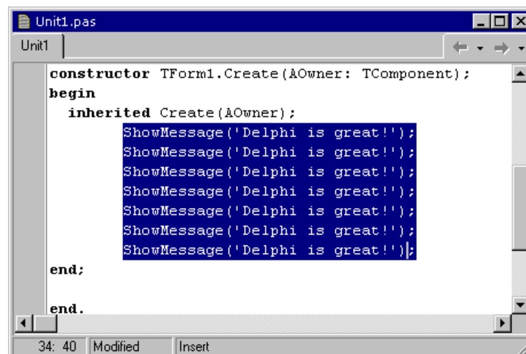
Las siguientes combinaciones de teclas y ratón sirven durante el diseño de formularios, y se aplican cuando hay algún componente seleccionado, y el formulario de diseño está activo, con la barra de títulos en color azul:

CTRL+FLECHA	Mueve el componente seleccionado, píxel a píxel
MAY+FLECHA	Cambia el tamaño del componente seleccionado, píxel a píxel
ESC	Selecciona el "padre" del componente activo
MAY+RATÓN	Selecciona el formulario, que puede estar oculto bajo varias capas de componentes

También podemos activar el movimiento o cambio de tamaño de un componente píxel a píxel si pulsamos la tecla ALT mientras lo manipulamos con el ratón.

Para el Editor de Código también existen trucos. Cuando hay un bloque seleccionado en el editor, pueden aplicarse las siguientes operaciones:

CTRL+K P	Imprime el bloque
CTRL+K I	Aumenta el margen del bloque
CTRL+K U	Disminuye el margen del bloque



Y hablando de bloques, ¿sabía que se pueden definir bloques "rectangulares" en el Editor de Código? El modo de definición de los bloques se controla mediante los siguiente comandos:

CTRL+O+C	Bloques por columnas
CTRL+O+I	Bloques "inclusivos" (¡pruébelos!)
CTRL+O+K	Bloques no inclusivos (los de toda la vida)
CTRL+O+L	Bloques por líneas

¿Tiene una unidad mencionada en la cláusula **uses** de un formulario, y quiere abrir el fichero con el código fuente? Si el fichero se encuentra en el directorio activo, pulse la combinación CTRL+INTRO, estando situado sobre el nombre de la unidad, y Delphi se encargará del asunto. Pulsando CTRL+K y un número del cero al nueve, creamos una marca de posición a la cual podemos regresar en cualquier momento pulsando CTRL+Q, más el número en cuestión.

Pero quizás la mayor sorpresa es descubrir que Delphi ofrece un mecanismo rudimentario para grabar y ejecutar una macro de teclado:

CTRL+MAY+R	Inicia y termina la grabación de una macro
CTRL+MAY+P	Reproduce la última macro grabada

Por último, las siguientes combinaciones de teclas pueden resultar útiles para no perdernos dentro de la maraña de ventanas y diálogos de una sesión típica de desarrollo con Delphi:

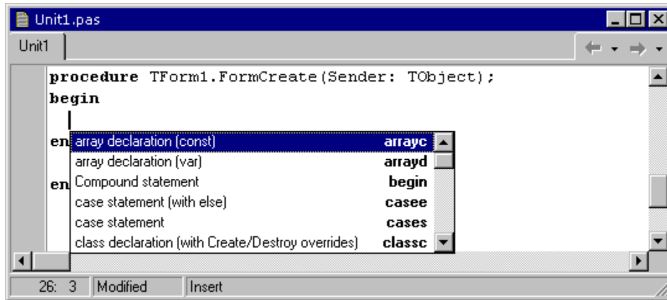
F11	Trae al Inspector de Objetos al primer plano
F12	Intercambia representación visual de un formulario con su código
MAY+F12	Muestra la lista de formularios de un proyecto
CTRL+F12	Muestra la lista de unidades de código de un proyecto
ALT+0	Muestra *todas* las ventanas abiertas del Entorno de Desarrollo

La tecla F12 funciona no sólo con el Editor de Código y los formularios en diseño, sino también con el editor de *paquetes* (ver el capítulo 3) y con el editor de Bibliotecas de Tipos. Finalmente, el nuevo comando *Search|Find in files* nos permite buscar textos dentro de ficheros, que pueden ser los del proyecto activo o los de un directorio determinado.

> He dejado para el final el mejor truco: ¿sabía usted que en el fichero de ayuda aparecen estos comandos, y muchos más, perfectamente documentados? Quizás merezca la pena que pierda unos minutos aprendiendo trucos de teclado, pues le harán ser más productivo.

Code Insight: ayuda durante la edición

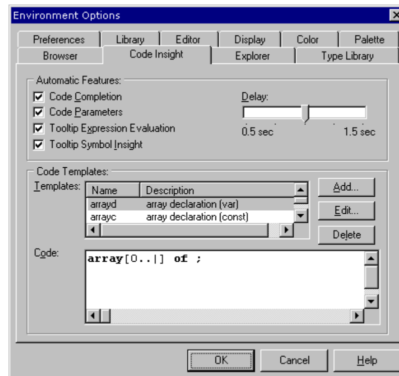
El Editor de Código ha experimentado mejoras sustanciales a partir de Delphi 3. Una de ellas es el uso de plantillas de código, o *code templates*. Sitúese en el editor y pulse la combinación de teclas CTRL+J. En respuesta, aparece una lista de nombres de plantillas. Cuando seleccionamos una, se copia el código asociado, y el cursor se desplaza a un punto que depende de la definición de la plantilla.



Otra forma de lograr el mismo efecto es tecleando primero el nombre de la plantilla y pulsando entonces CTRL+J; pruebe, por ejemplo, con **whileb**, que crea el esqueleto de un bloque **while** con una instrucción compuesta **begin…end**:

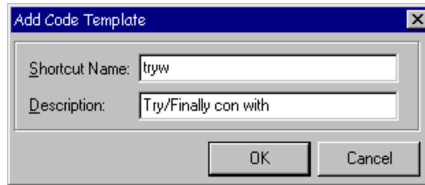
```
while { Aquí se queda el cursor } do
begin
end;
```

Aunque existe una gran variedad de plantillas predefinidas, también podemos añadir las nuestras. Las plantillas de código se configuran en el diálogo asociado al comando de menú *Tools|Environment options*, en la página *Code Insight*:



El texto de las plantillas se guardan en el subdirectorio *bin* de Delphi, en el fichero *delphi32.dci*. Para definir una plantilla nueva, el programador debe pulsar el botón

Add. En respuesta aparece un cuadro de diálogo que pide el nombre y la descripción de la plantilla:

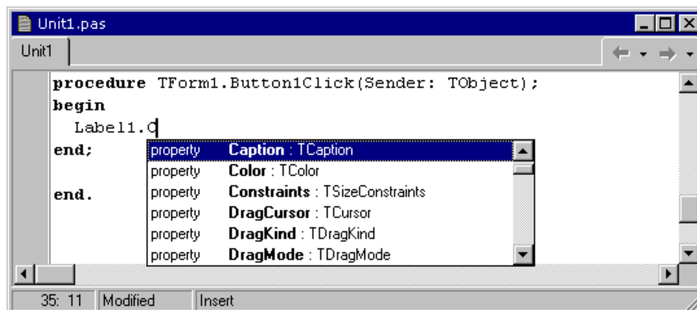


Luego, debe utilizarse el editor para suministrar el texto de la plantilla. La barra vertical | sirve para indicar dónde debe quedar el cursor al ejecutarse la plantilla. Por ejemplo:

```
with T|.Create(Self) do
try
finally
   Free;
end;
```

> Las definiciones de macros de *Code Insight* se almacenan en el fichero *delphi32.dci*, en el directorio *bin* de Delphi.

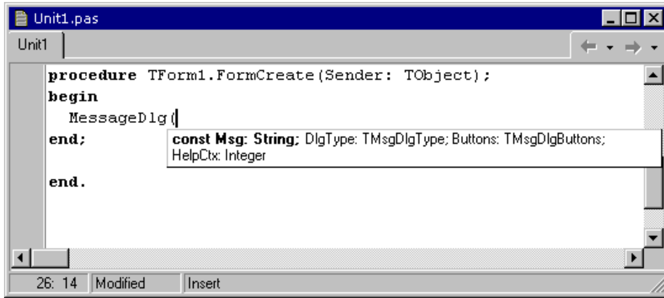
Pero seguramente el programador utilizará con mayor frecuencia las ayudas interactivas de Delphi acerca de los métodos aplicables a un objeto, y de los parámetros de los procedimientos y funciones. Cuando tecleamos el nombre de una variable de objeto, o de un tipo de clase, y añadimos un punto, aparece una ventana emergente con la lista de métodos y propiedades aplicables. Da lo mismo que el tipo del objeto sea predefinido o haya sido creado por nosotros, con tal de que esté compilado.



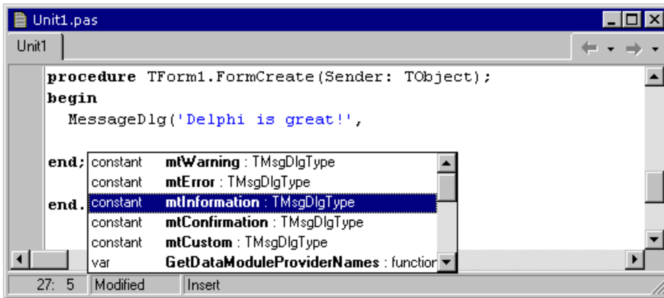
Si quiere que aparezca esta ventana en cualquier otro momento, debe pulsar la combinación de teclas CTRL+ESPACIO.

Para los procedimientos, funciones y métodos, la ayuda se presenta cuando tecleamos el paréntesis de apertura a continuación del nombre de una rutina. La ventana

de ayuda es una ventana de indicaciones (esos recuadros amarillos que se interponen en nuestro campo visual cuando dejamos tranquilo al ratón por un tiempo), con el nombre y tipo de los parámetros, según aparecen en la declaración de la rutina. Si se nos pierde la ventana, tenemos el comando de teclado CTRL+MAY+ESPACIO, para que reaparezca la ayuda:

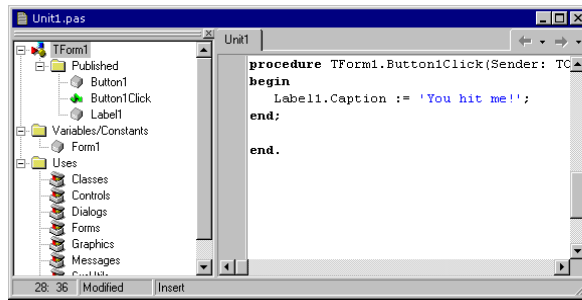


La misma combinación CTRL+ESPACIO, que nos ofrece ayuda para seleccionar métodos y propiedades, sirve para ayudarnos con los posibles valores de un parámetro cuando nos encontramos dentro de la lista de parámetros de un método, procedimiento o función:



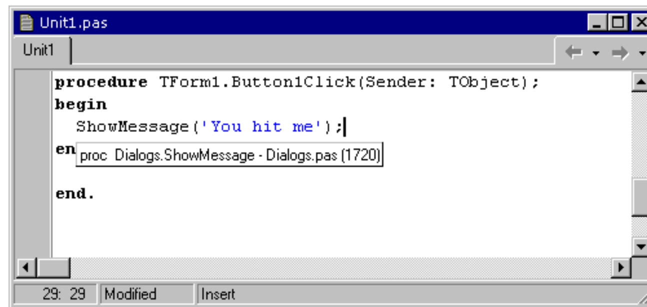
Hay que utilizar con precaución esta técnica, porque en la lista de posibilidades aparecen todas las constantes, variables y funciones de tipos compatibles, y esto incluye a las de tipo *Variant*, que salen hasta en la sopa.

Ahora mencionaremos las novedades de Delphi 4. La primera es la presencia del nuevo Explorador de Código:



Esta ventana muestra los símbolos definidos dentro de una unidad. Si está activa, podemos realizar búsquedas incrementales de símbolos utilizando el teclado. También es útil para cambiar el nombre a un símbolo. Tenga en cuenta, no obstante, que el cambio solamente se aplicará a la declaración, y que habrá que cambiar manualmente el código en los sitios donde se utiliza el símbolo. Ah, y si le molesta mucho esta ventana, recuerde que puede arrastrarla fuera del Editor, o simplemente cerrarla.

La segunda novedad se conoce como Navegación por el Código. Si colocamos el ratón (más bien, el puntero del mismo) sobre un símbolo, ya sea una variable, una clase, un método, un procedimiento, aparece una indicación acerca de dónde está definido tal símbolo.



Si además pulsamos la tecla CONTROL mientras estamos sobre el símbolo, éste adopta la apariencia de un enlace HTML (¡ah, la moda!), y nos permite ir inmediatamente al lugar donde ha sido definido o implementado. La historia de esta navegación se almacena en las listas desplegables asociadas al par de botones de la esquina superior derecha del Editor, que nos permiten movernos hacia delante y hacia atrás por la misma.

Por último, las combinaciones de teclas CTRL+MAY+ABAJO y CTRL+MAY+ARRIBA, sirven para alternar rápidamente entre la declaración de un método en la interfaz de una unidad, y su cuerpo, en la sección de implementación.

Class completion

Class completion es otra de las técnicas de mejora de la productividad introducidas por Delphi 4. Como el lector conoce, los métodos de una clase deben definir sus cabeceras dentro de la declaración de la misma, pero su cuerpo debe definirse por separado. Esto implica, casi siempre, navegar desesperadamente desde la sección de interfaz de la unidad a la sección de implementación, y un uso habilidoso de los comandos de copiar y pegar. Delphi nos releva de esta tortura cuando crea métodos de respuesta a eventos, pero nos deja abandonados cuando tenemos que definir otro tipo de métodos. Lo mismo sucede cuando queremos definir *propiedades* para una clase. La declaración de una propiedad debe ir precedida por la declaración de sus métodos de acceso. Estos, a su vez, llevan doble trabajo: la declaración de sus cabeceras y su posterior implementación. Es cierto que estas labores son más frecuentes durante la escritura de componentes ... pero todos tenemos derecho a una vida mejor.

Delphi 4 ofrece la combinación CTRL+MAY+C para ayudarnos en estos menesteres. Supongamos que tecleamos la siguiente declaración de clase:

```
type
   TYoQueSe = class(TCiertaClase)
   public
      constructor Create(AOwner: TComponent);
      destructor Destroy; override;
   published
      property Entero: Integer;
      property Vector[Index: Integer]: string;
   end;
```

Hemos dejado incompletas las dos declaraciones de propiedades. Si ahora tecleamos CTRL+MAY+C, Delphi modifica la clase de esta manera:

```
type
   TYoQueSe = class(TCiertaClase)
   private
      FEntero: Integer;
      function GetVector(Index: Integer): string;
      procedure SetEntero(const Value: Integer);
      procedure SetVector(Index: Integer; const Value: string);
   public
      constructor Create(AOwner: TComponent);
      destructor Destroy; override;
   published
      property Entero: Integer read FEntero write SetEntero;
      property Vector[Index: Integer]: string
         read GetVector write SetVector;
   end;
```

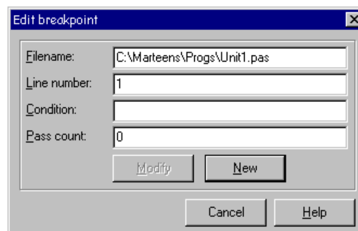
También añade, en la sección de implementación de la clase, un cuerpo para el constructor, el destructor y para los tres métodos de acceso que ha generado. Incluso, en el caso de los métodos de acceso, se atreve a sugerirnos una posible implementación:

```
procedure TYoQueSe.SetEntero(const Value: Integer);
begin
   FEntero := Value;
end;
```

Herramientas de depuración

Para desarrollar eficientemente aplicaciones con Delphi, es sumamente conveniente dominar las técnicas de depuración. Estas técnicas siguen siendo similares a las del depurador integrado en las primeras versiones de Turbo Pascal. Sin embargo, las diferencias quedan determinadas por la diferente estructura de los programas escritos para MS-DOS y Windows. En un programa para Windows es imposible seguir paso a paso la ejecución de todas las instrucciones del mismo. Hay un momento en que perdemos de vista el puntero de las instrucciones: cuando nos sumergimos en una rutina conocida como el *ciclo de mensajes*. A partir de este punto, recuperamos periódicamente el control, cuando Delphi, C++ Builder ó Visual Basic ejecutan algunas de las respuestas a *eventos*, o cuando C++ ejecuta alguna función de respuesta o método asociado a *mensajes*.

Por lo tanto, lo que debe saber el programador de Delphi es cómo interceptar al programa cuando pase por los sitios que le interesa, y para esto tiene dos posibilidades. La primera es establecer un *punto de ruptura* (ó *breakpoint*) en una instrucción, para lo cual puede pulsar la tecla F5 (CTRL+F8, si se utiliza a configuración de teclado tradicional de Borland), o ejecutar el comando de menú *Run|Add breakpoint*, estando seleccionada la línea en la cual nos queremos detener. Cuando el programa se ejecute y pase por esa línea, se detendrá, se activará el Entorno de Desarrollo, y podremos decidir qué hacer: evaluar el estado actual de las variables, seguir ejecutando paso a paso, detener el programa ... La otra posibilidad es utilizar el comando de menú *Run|Run to cursor*, o pulsar la tecla F4, estando situados sobre la línea deseada. El efecto es similar a colocar un punto de ruptura temporal, ejecutar el programa y liberar el punto una vez llegados al mismo.



A un punto de ruptura se le puede asociar una condición de parada, que debe cumplirse para que el programa detenga allí su ejecución, y un contador de pasadas, para activar el punto a las tantas veces que se pase por él. Los puntos de ruptura pueden

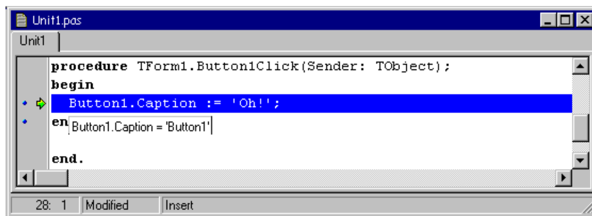
controlarse con el comando de menú *View|Breakpoints*. El margen izquierdo del Editor de Código muestra también los puntos de ruptura y el estado de los mismos.

Una vez que el programa se ha detenido temporalmente en un lugar determinado, comenzamos a seguirle la pista a las instrucciones por las que pasa. Tenemos a nuestra disposición los siguientes comandos del menú *Run*:

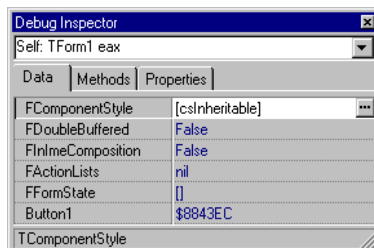
Trace into (F7)	Avanzar una instrucción. Si es una rutina y tenemos el código fuente, entramos en su interior.
Step over (F8)	Igual a la anterior, pero esquivando el interior de las rutinas.
Trace to next source line (MAY+F7)	Ir hasta la siguiente línea de código disponible.

También se puede pulsar F9, y dejar que el programa se ejecute hasta terminar o encontrar un punto de ruptura.

La otra técnica importante nos permite conocer el contenido de las variables, atributos y propiedades del espacio de datos del programa. Los comandos *Run|Evaluate* y *Add watch* nos permiten evaluar dinámicamente una expresión ó inspeccionar constantemente el valor, respectivamente.



Recuerde también que, si se trata de una variable o propiedad de un objeto, basta con situar el cursor sobre una referencia a la misma en el código del programa para obtener su contenido en una ventana de indicaciones. También podemos utilizar el comando de menú *Run|Inspect*, que solamente está disponible en tiempo de ejecución, para mostrar una ventana como la siguiente con los datos, métodos y propiedades del objeto que deseemos:

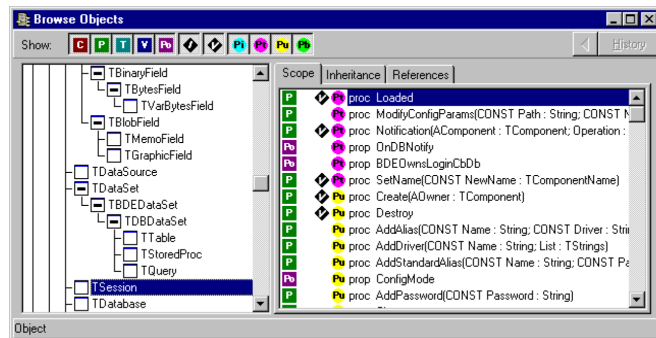


Por último, Delphi 4 añade una nueva ventana de depuración: el *event log*, o registro de eventos que almacena y muestra opcionalmente los siguientes sucesos: carga y descarga de módulos por la aplicación, mensajes de Windows recibidos por nuestra aplicación, activación de puntos de rupturas y mensajes generados con el procedimiento de Delphi *OutputDebugString*. ¿Recuerda que en la época de MS-DOS la técnica más potente de depuración era con frecuencia escribir en la consola mensajes del tipo *"Killroy was here"*? Ahora podemos retornar nuestra adolescencia gracias al registro de eventos.

Object Browser

Uno de los grandes desconocidos del Entorno de Desarrollo de Delphi es el *Explorador de Objetos*; en inglés: *Object Browser*. Es una herramienta integrada en el entorno, que se ejecuta desde el comando de menú *View|Browser*. La información que se muestra con la misma se extrae del ejecutable de la aplicación, por lo que el proyecto debe haber sido compilado previamente para poder activar la utilidad.

La siguiente figura muestra la ventana principal del Explorador. Consta de dos paneles; en el de la izquierda se muestra la jerarquía de las clases incluidas en el proyecto, mientras que sus métodos, eventos y propiedades se muestran en el panel derecho. La barra de herramientas de la porción superior de la ventana sirve para restringir la información visualizada. Una posibilidad interesante es la navegación por referencias.

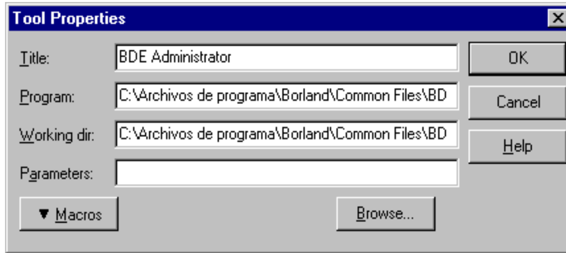


BDE Administrator

El Administrador del Motor de Datos (*BDE Administrator*) es el programa necesario para configurar las opciones de acceso a bases de datos de Delphi. Esta aplicación también se instala cuando distribuimos el Motor de Datos con nuestras aplicaciones. Puede ejecutarse desde el grupo de programas de Delphi. También puede configurarse el menú *Tools* del entorno de desarrollo de Delphi para incluir el Administrador,

si es que vamos a utilizarlo con frecuencia. Para configurar *Tools* ejecute el comando de menú *Tools|Configure tools*. En el cuadro de diálogo que aparece, pulse el botón *Add* para añadir la nueva opción. El ejecutable del Administrador, en mi ordenador, corresponde al fichero:

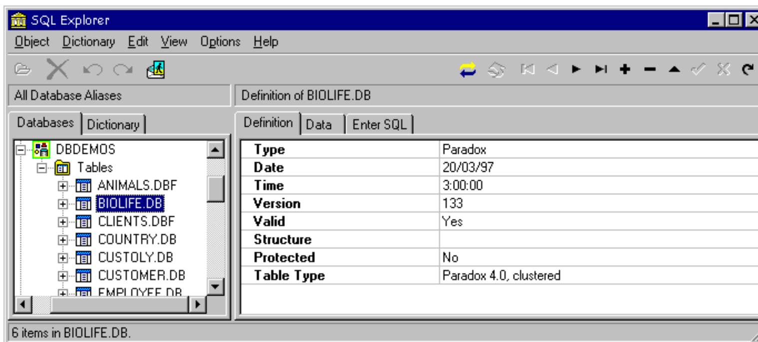
```
C:\Archivos de programa\Borland\Common Files\BDE\Bdeadmin.exe
```



Para más detalles sobre esta herramienta, lea el capítulo sobre el Motor de Bases de Datos.

Database Explorer

Para conocer la estructura de una base de datos, saber qué tablas contiene, qué columnas, índices y restricciones se han definido para cada tabla, y para visualizar los datos almacenados en las mismas, Delphi ofrece *Database Explorer*: el Explorador de Bases de Datos. Para activar este programa desde Delphi, basta con seleccionar el comando de menú *Database|Explore*, pero también puede ejecutarse como utilidad independiente desde el grupo de programas de Delphi. En la versión cliente/servidor, en la que se incluye, como es lógico, el acceso a bases de datos SQL, la utilidad se nombra *SQL Explorer*.



La ventana principal de la utilidad contiene dos páginas: *Databases* y *Dictionary*. En la primera es donde aparecen las distintas bases de datos asociadas a los alias persis-

tentes del Motor de Datos. El árbol del panel de la izquierda puede expandirse para mostrar jerárquicamente los objetos de la base de datos. Se pueden realizar modificaciones en todos estos objetos en el panel de la derecha.

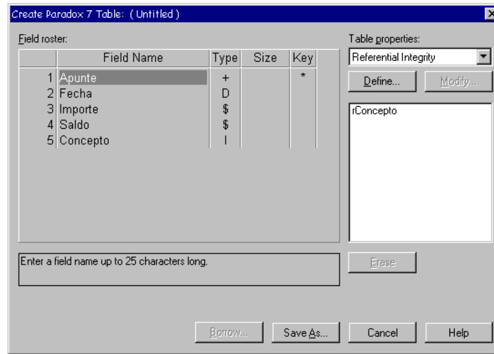
El Explorador de Bases de Datos también sirve para configurar el *Diccionario de Datos*. Esta entidad es utilizada por Delphi en tiempo de diseño; no tiene función alguna en tiempo de ejecución. En el Diccionario de Datos pueden almacenarse, en primer lugar, definiciones de *conjuntos de atributos* de campos. Los conjuntos de atributos se pueden asociar a los campos de las tablas al ser creados desde Delphi, y permiten uniformizar las propiedades de visualización, edición y validación de los mismos. También se consultan cuando arrastramos un campo sobre un formulario y Delphi crea un control de edición; el tipo del control puede determinarse de acuerdo a los atributos asociados.

La otra operación importante con el Diccionario de Datos es la *importación de bases de datos* al diccionario. Cuando importamos una base de datos, se copian sus definiciones de tablas, campos e índices al Diccionario. Se pueden entonces asociar conjuntos de atributos a los campos, de modo tal que estos atributos se asocian automáticamente cuando se crean los campos. Cuando una base de datos ha sido importada, pueden también importarse las restricciones de integridad a determinadas propiedades de los campos y tablas de Delphi. Esto es útil en la programación cliente/servidor, pues permite evaluar las verificaciones en los ordenadores clientes antes de ser enviadas, disminuyendo el tráfico en la red.

Los datos del Diccionario por omisión se almacenan en formato Paradox, en la tabla *bdesdd.db*. Sin embargo, pueden crearse nuevos diccionarios en diferentes formatos, incluso en bases de datos cliente/servidor, y ser compartidos por todos los miembros de un equipo de desarrollo.

Database Desktop

Sin embargo, mi herramienta preferida para crear tablas de Paradox y dBase sigue siendo *Database Desktop*, que acompaña a Delphi desde su primera versión. A pesar de la apariencia más moderna de Database Explorer, determinadas tareas se ejecutan más cómodamente y con mayor eficiencia con la "vieja" interfaz del Database Desktop. Por ejemplo, es muy fácil copiar tablas, modificar su estructura y editar su contenido. La siguiente figura muestra el diálogo de creación de una tabla en el formato de Paradox 7:

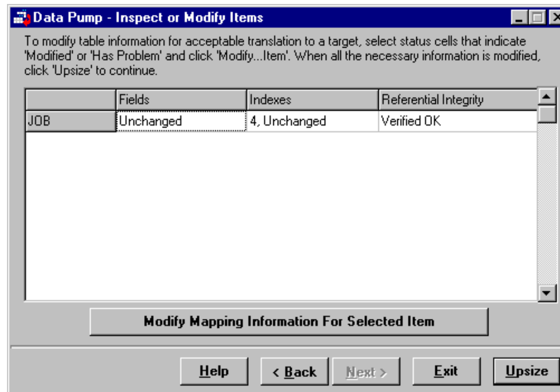


De todos modos, es recomendable que la administración de la estructura de las bases de datos SQL se realice, en general, utilizando las herramientas propias del sistema de bases de datos.

> Desgraciadamente, el Database Desktop que acompaña a Delphi 4 no permite aprovechar las nuevas posibilidades del formato DBF7. Ha sido un lamentable descuido que esperamos se corrija pronto.

Data Migration Wizard

Las versiones cliente/servidor de Delphi incluyen el programa *Data Migration Wizard*, que sirve para copiar tablas de una base de datos a otra. Este es un programa asistente, en el que se van presentando secuencialmente páginas en un cuadro de diálogo para ayudar paso a paso a realizar determinada operación.

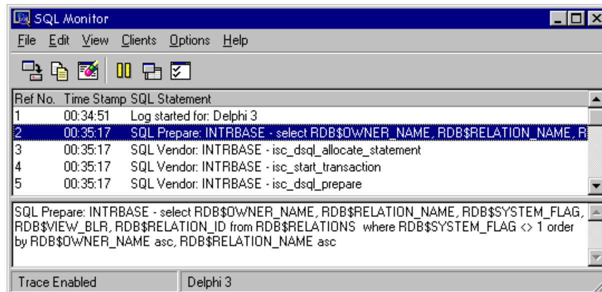


Aunque es muy fácil copiar una tabla utilizando el componente *TBatchMove* de Delphi, esta utilidad permite mover también las restricciones de integridad, índices, et-

cétera, algo que no está previsto para *TBatchMove*. Se puede también modificar la "traducción" que se genera para tipos de datos diferentes, como se puede ver en la figura anterior.

SQL Monitor

SQL Monitor es una utilidad incluida en las versiones cliente/servidor de Delphi, que sirve para seguir la pista de las instrucciones SQL generadas por el BDE que se envían al servidor.

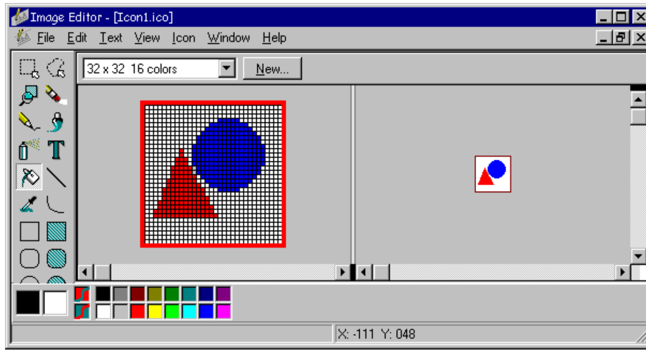


Cuando el Motor de Datos tiene que conectarse a una base de datos SQL, toda la comunicación transcurre enviando instrucciones SQL a la misma. El Motor trata con una interfaz uniforme a las tablas locales y a las tablas situadas en un servidor. Por este motivo, a veces la traducción a SQL de determinada rutina de alto nivel del BDE no es muy evidente. Gracias a esta utilidad, podemos saber qué órdenes le da el Motor de Datos al servidor SQL, y qué le responde éste.

SQL Monitor puede ejecutarse tanto desde el grupo de programas de Delphi, como desde Delphi, mediante el comando de menú *Database|SQL Monitor*. El contenido del capítulo 27 está basado completamente en el análisis de los datos que ofrece SQL Monitor.

Image Editor

Por último, Delphi ofrece un sencillo editor de imágenes, *Image Editor*, para los gráficos nuestros de cada día. Este programa permite editar imágenes en formato de mapas de bits e iconos. Pero también nos permite abrir y modificar ficheros de recursos y los ficheros de *recursos compilados de Delphi* (*Delphi Compiled Resources*), de extensión *dcr*.



Un fichero *dcr* es simplemente un fichero de recursos binarios que almacena solamente mapas de bits e iconos. Se utiliza fundamentalmente para definir los iconos asociados a los componentes de la Paleta de Componentes.

Image Editor no puede trabajar con cadenas contenidas en recursos. A partir de Delphi 3, se podía utilizar declaraciones **resourcestring** para que el compilador generara y enlazara los recursos correspondientes dentro de nuestros ejecutables. Pero no se podía extraer o modificar estas definiciones a partir de ejecutables o DLLs existentes. Delphi 4, por fin, ha añadido un editor de recursos de tablas de cadenas al entorno de desarrollo.

Capítulo

3

Unidades, proyectos y paquetes

PASCAL ES UN LENGUAJE CON BASTANTE HISTORIA, quizás con demasiada. Comenzó siendo un proyecto docente para la enseñanza de la entonces reciente Programación Estructurada; ya sabemos: eliminar los saltos a etiquetas dentro del código y sustituirlos por elegantes instrucciones de control. Era un lenguaje de pequeño tamaño y poca complejidad, fácil de aprender, pero inadecuado para la programación real, sobre todo por carecer de la posibilidad de descomponer el programa en módulos independientes capaces de ser compilados por separado. Todo programa en el Pascal original se escribía en un mismo fichero, y se compilaba y enlazaba en una sola operación.

Al convertirse Pascal en un lenguaje para la programación "real" tuvo que transformarse, incluyendo nuevas estructuras sintácticas para la compilación por separado y el trabajo en equipo. La implementación que trazó el camino a seguir fue la de UCSD Pascal, un experimento interesante que integraba lenguaje más sistema operativo (¡quiera Dios que a Microsoft no se le ocurra repetir la experiencia!). Luego Borland y la compañía de Redmond utilizaron variantes de la sintaxis establecida por esta implementación y la incorporaron a sus productos: estoy hablando de las *units* o *unidades*.

El otro cambio importante que sufrió Pascal se produjo cuando se popularizó la Programación Orientada a Objetos. No se introdujeron cambios en la estructura de proyectos, pero cambió bastante la metodología de trabajo, que se tuvo que adaptar a la forma histórica de organización del lenguaje. Por ejemplo, como veremos al estudiar las técnicas de Programación Orientada a Objetos, los mecanismos de control de acceso de las unidades y los de las declaraciones de clase se interfieren mutuamente, lo cual no es muy elegante. Y mientras menos elegante sea un lenguaje de programación es más difícil de aprender.

En este capítulo describiré a grandes rasgos los ficheros y formatos que se utilizan en los proyectos de Delphi, y los tipos de proyectos que se pueden crear.

La estructura de un proyecto de aplicación

El proyecto más frecuente en Delphi es el proyecto de aplicación. Su objetivo: la creación de un ejecutable. Delphi divide las aplicaciones en módulos relativamente independientes. Aunque más adelante veremos toda la flexibilidad del esquema de partición, en principio se utiliza un sistema muy sencillo: cada ventana se coloca en una unidad separada, y la integración de las unidades se produce gracias a un fichero especial, el *fichero de proyecto*. Todo el esqueleto básico del proyecto de aplicación es creado automáticamente por Delphi, y muy pocas veces es necesario modificarlo explícitamente, pero es aconsejable conocer qué está sucediendo detrás del telón.

El *fichero de proyecto* tiene la extensión *dpr*: *Delphi PRoject*, es decir, Proyecto de Delphi. El contenido de este fichero es Pascal, y su estructura es similar a la que tenían los programas escritos en el lenguaje Pascal original:

```
program NombreProyecto;
```

```
uses
        Lista de Unidades;
```

```
        Declaraciones
```

```
begin
        Lista de instrucciones
end.
```

En primer lugar, tenemos el nombre del proyecto; este nombre debe ser un *identificador* de Pascal: debe comenzar con un carácter alfabético y el resto de sus caracteres deben ser dígitos o letras sin acentos[2]. El nombre utilizado para el proyecto debe coincidir con el nombre del fichero en que se almacena. De hecho, este nombre cambia cuando guardamos el fichero de proyecto bajo otro nombre. La principal consecuencia de la regla anterior es que no se pueden generar en Delphi, al menos de forma directa, aplicaciones cuyos nombres contengan acentos, espacios en blanco y otros caracteres válidos como nombres de fichero, pero no permitidos en identificadores Pascal.

La siguiente sección es la cláusula **uses**, en la que se nombran las *unidades* que forman parte del proyecto. En el esqueleto generado por Delphi, todas las unidades integradas en el proyecto son mencionadas en esta cláusula, aunque no es estrictamente necesario. Mediante la sección **uses** del fichero de proyecto, el enlazador de Delphi sabe qué unidades deben enlazarse para crear el ejecutable. En contraste, otros lenguajes requieren de la presencia por separado de un fichero de opciones de enlace en

---

[2] Bueno, sí, vale ... y también el carácter subrayado, ¡al cual detesto de todo corazón!

el cuál se especifiquen explícitamente las dependencias entre los módulos. Toda esta información la deduce Delphi automáticamente.

No obstante, el objetivo primordial de la inclusión de una unidad en la cláusula **uses** es proporcionar acceso a las declaraciones públicas efectuadas dentro de la misma. En particular, estas declaraciones se utilizan en el proyecto por omisión de Delphi para la creación automática de los objetos de ventanas durante la inicialización de la aplicación. Más adelante, en este capítulo, estudiaremos más detalles de las cláusulas **uses** y de las declaraciones en unidades. La creación automática de formularios se estudia en el capítulo 13.

A continuación de la cláusula **uses** pueden incluirse declaraciones de constantes, tipos, variables y rutinas; todas para uso exclusivo del fichero de proyecto. Por último, es obligatoria la presencia de una lista de instrucciones delimitada por las palabras reservadas **begin** y **end**. La ejecución de la aplicación comienza por esta lista de instrucciones. En contraste, en otros lenguajes de programación es más difícil determinar a priori por dónde comienza la ejecución; en C y C++, por ejemplo, hay que buscar en qué módulo o fichero se ha definido una función de nombre *main*, en MS-DOS, o *WinMain*, en Windows.

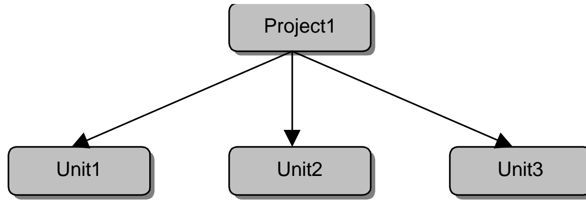
El siguiente listado muestra un fichero de proyecto típico de Delphi. Este proyecto contiene tres ventanas que se crean automáticamente al principio de la aplicación:

```
program Project1;
uses
   Forms,
   Unit1 in 'Unit1.pas' {Form1},
   Unit2 in 'Unit2.pas' {Form2},
   Unit3 in 'Unit3.pas' {Form3};

{$R *.RES}

begin
   Application.Initialize;
   Application.CreateForm(TForm1, Form1);
   Application.CreateForm(TForm2, Form2);
   Application.CreateForm(TForm3, Form3);
   Application.Run;
end.
```

Observe la ausencia de la lista de declaraciones. La línea encerrada entre llaves será explicada más adelante.

¿Para qué se necesitan unidades? En realidad, todo el código y los datos del programa pudiera colocarse teóricamente en el mismo fichero de proyecto. Pero en la práctica se haría imposible el mantenimiento de la aplicación resultante. Por eso se reparten las rutinas y los datos de la aplicación entre diversos ficheros, actuando el proyecto como coordinador de la estructura de ficheros generada. El diagrama a

continuación ilustra las relaciones establecidas en el ejemplo de proyecto anterior. Las flechas representan la mención de las unidades en la cláusula **uses**. Este diagrama es una simplificación, pues pueden existir referencias entre las propias unidades. Más adelante estudiaremos la sintaxis de las unidades, la forma de establecer referencias entre ellas y los motivos que pueden existir para estas referencias.



¿Qué se puede declarar?

En la sección anterior he mencionado la palabra "declaraciones". Pascal tiene una regla básica: todo identificador que utilicemos debe haber sido previamente declarado. Ahora bien, ¿qué entidades pueden utilizarse en un programa Pascal que deban ser declaradas? En primer lugar, están las declaraciones de constantes, como las siguientes:

```
const
   PiSobre2 = 1.57079637;
   Lenguaje = 'Pascal';
   Delphi = 'Object' + Lenguaje;
```

Luego tenemos la posibilidad de que el programador defina *tipos de datos*. Pascal ofrece de entrada un amplio conjunto de tipos básicos predefinidos, y en sus bibliotecas se incluyen multitud de tipos formados a partir de los básicos, para resolver los problemas comunes más frecuentes de la programación. En particular, cuando diseñamos una ventana en Delphi, estamos definiendo realmente un tipo especial: una *clase*. Las *clases* y *objetos* son característicos de la Programación Orientada a Objetos; los capítulos 6 y 7 de este libro se dedican al estudio de la misma. Por ejemplo, una ventana vacía genera automáticamente la siguiente declaración de tipo:

```
type
   TForm1 = class(TForm)
   private
      { Private declarations }
   public
      { Public declarations }
   end;
```

Tendremos que esperar un poco para poder explicar el significado del fragmento de código anterior.

A continuación, están las declaraciones de variables. Estas declaraciones reservan memoria para almacenar información. En Pascal, cada variable debe pertenecer a un tipo determinado, ya sea predefinido o creado anteriormente por el programador. En una unidad creada por Delphi para contener un formulario, a continuación de la declaración del tipo de formulario (la definición mostrada anteriormente de *TForm1*), se declara una variable de ese tipo:

```
var
   Form1: TForm1;
```

Por último, tenemos los procedimientos y funciones, pues antes de llamar a una rutina el compilador necesita conocer su prototipo. Las declaraciones de rutinas en un fichero de proyecto van acompañadas por el cuerpo de las mismas; en las siguientes secciones veremos que las declaraciones de procedimientos y funciones en las interfaces de unidades pueden prescindir temporalmente de su implementación. Esta es la sintaxis, grosso modo, de una declaración de procedimiento; la de una función es similar:

```
procedure Nombre [(Parámetros)];
Declaraciones
begin
   ListaDeInstrucciones
end;
```

Tome nota de que dentro de un procedimiento vuelve a repetirse la sintaxis de las declaraciones. De este modo, un procedimiento puede utilizar internamente a otras rutinas que solamente estarán a su disposición. Este era el estilo de programación en boga cuando Wirth diseñó el lenguaje Pascal; el estilo sigue siendo válido, pero existen alternativas modernas para la organización del código de un programa.

He dejado aparte, intencionalmente, a las declaraciones de *etiquetas*, que utilizan la palabra clave **label**. Estas etiquetas se utilizan en conjunción con las políticamente incorrectas instrucciones de salto **goto**, como en el siguiente ejemplo:

```
function BuscarCuadricula(const M: TMatrizNombres;
   const S: string): Boolean;
label
   TERMINAR;
var
   I, J: Integer;
begin
   BuscarCuadricula := False;
   for I := 1 to M.Filas do
      for J := 1 to M.Columnas do
         if M.Cuadricula[I, J] = S then
         begin
            BuscarCuadricula := True;
            goto TERMINAR;
         end;
```

```
TERMINAR:
   // Ponga aquí su código favorito
end;
```

En el Pascal moderno hubiéramos empleado la instrucción *Exit* para terminar ambos bucles saliendo de la función. Las instrucciones *Break* y *Continue* también ayudan a evitar estas situaciones, pero no estaban disponibles en la primera definición de Pascal.

El orden de las secciones de declaraciones original de Pascal era rígido:

```
label - const - type - var - procedure/function
```

La teoría era que las constantes podían utilizarse en la declaración de tipos, los tipos podían ser utilizados en la declaración de variables, y las funciones y procedimientos podían aprovecharse entonces de todas las declaraciones anteriores. ¿Y las etiquetas de salto? Bueno, si hay que declararlas en algún lugar, ¿qué más da si es al principio o al final? Turbo Pascal permitió intercambiar arbitrariamente el orden de las secciones de declaración, e incluso escribir varias secciones independientes del mismo tipo.

Sintaxis de las unidades

En realidad, los ladrillos básicos con los que se construye una aplicación, y en los que se escribe casi todo el código de la misma, son las *unidades*. Una unidad es una colección de declaraciones que puede ser compilada por separado, y de esta forma puede volver a utilizarse en otros proyectos. Por omisión, el código fuente de una unidad Pascal se almacena en un fichero de extensión *pas*. Cada unidad está dividida en dos secciones principales: la *interfaz* (*interface*) y la *implementación* (*implementation*). Todas las declaraciones de la interfaz son públicas, y están a disposición de cualquier otra unidad o proyecto interesado en las mismas. Por el contrario, las declaraciones realizadas en la sección de implementación son privadas para la unidad. En contraste, otros lenguajes de programación requieren que cada declaración indique explícitamente su carácter público o privado. En la época oscura de la Programación Tradicional, las declaraciones consistían fundamentalmente en funciones y procedimientos; con el advenimiento de la Programación Orientada a Objetos, son las clases las declaraciones más frecuentes. Esta es la estructura de una unidad de Delphi:

```
unit NombreUnidad;
```

```
interface

        Declaraciones
```

```
implementation

       Más declaraciones y cuerpos de rutinas
```

```
initialization

       Instrucciones de inicialización
```

```
finalization

       Instrucciones de finalización
```

```
end.
```

Las cláusulas de inicialización y finalización son opcionales; la palabra reservada **begin** puede utilizarse en lugar de **initialization**[3]. Las unidades de Delphi 1 no tienen la cláusula de finalización.

El siguiente listado corresponde a una unidad que contiene la declaración de una ventana con un botón; todo el código que se muestra es generado automáticamente por Delphi:

```
unit Unit1;
interface
uses
   Windows, Messages, SysUtils, Classes, Graphics, Controls,
   Forms, Dialogs;

type
   TForm1 = class(TForm)
      Button1: TButton;
   private
      { Private declarations }
   public
      { Public declarations }
   end;

var
   Form1: TForm1;

implementation

{$R *.DFM}

end.
```

La compilación de una unidad produce un fichero de extensión *dcu* (*Delphi Compiled Unit*), que contiene tanto el código de las funciones traducido a código nativo del

[3] El origen americano de Delphi se evidencia en la grafía **initialization**, en contraste con la forma inglesa **initialisation**.

procesador, como las declaraciones públicas de la unidad en forma binaria. Para generar un fichero ejecutable, es necesario que todas las unidades referidas por el mismo estén en ese formato binario. El compilador se encarga automáticamente de producir los ficheros *dcu* según sea necesario. Para asegurar que los cambios realizados en el código fuente se reflejen en la unidad compilada, se utiliza la fecha de los ficheros: si el fichero *pas* es posterior al *dcu*, se compila nuevamente la unidad.

La cláusula uses y las referencias circulares

Como se puede apreciar en el listado de la sección anterior, una unidad puede necesitar las declaraciones realizadas en otras unidades, por lo cual puede también utilizar cláusulas **uses**. Las unidades admiten hasta dos cláusulas, una en la interfaz y otra en la sección de implementación. ¿Qué diferencia existe entre ambas? Aquí entra en juego el viejo principio de Pascal: todo símbolo que se utilice debe declararse con anterioridad. Si hace falta utilizar, en la interfaz de la unidad *A*, algún tipo de dato o constante declarados en la unidad *B*, tenemos que mencionar a esta última en la cláusula **uses** de la interfaz de *A*. Pero si las declaraciones de *B* sólo se utilizan en la implementación de *A*, podemos (y de hecho, es recomendable) postergar la referencia hasta la sección de implementación.

Aclaremos un error de interpretación frecuente. Muchos lenguajes de programación, C++ incluido, utilizan un mecanismo de *inclusión* para referirse a declaraciones externas. Típicamente, para tener acceso a funciones programadas en C++ es necesario "insertar" al principio de un fichero un listado con los prototipos de las mismas; a este listado se le conoce como fichero de *cabecera* (*header file*), y su extensión casi siempre es *h* ó *hpp*. Por ejemplo:

```
// Fichero de cabecera de C++
// declara.hpp

int unaFuncion();
```

```
// Fichero que utiliza a declara.hpp
#include <declara.hpp>

// Código que utiliza las declaraciones
// incluidas
```

La línea del segundo fichero que comienza con la directiva *#include* hace que el compilador procese en ese punto todas las líneas del fichero mencionado. El problema ocurre cuando el fichero de cabecera, *declara.hpp*, requiere declaraciones realizadas en otro fichero, digamos que *declara2.hpp*. El primero debe especificar entonces una segunda directiva de inclusión: *#include <declara2.hpp>*. Pero cuando alguien incluya ahora a *declara.hpp*, estará también incluyendo de forma automática a *declara2.hpp*. Y lo malo es que posiblemente el programador no es consciente de esta inclusión, por lo que erróneamente puede producirse la inclusión múltiple del fichero.

Esta situación no se produce en Delphi, pues la cláusula **uses** solamente da acceso a las declaraciones realizadas exactamente en la unidad mencionada:

```
unit A;
```

```
unit B;
interface
uses A;
```

```
unit C;
interface
uses B;
```

En el ejemplo anterior, la unidad *C* no tiene acceso en su interfaz a las declaraciones realizadas en la unidad *A*. Precisamente, el comportamiento de la cláusula **uses** en comparación con el de la directiva *#include* es uno de los motivos por los que Pascal es un lenguaje que, intrínsecamente, puede compilarse a mayores velocidades que los actuales C y C++.

Un caso particular de relación de referencia entre unidades ocurre cuando existen *ciclos* dentro del diagrama de referencias: la unidad *A* referencia a la unidad *B*, la cual necesita a la unidad *C*, que finalmente hace referencia a *A*. No hay problema alguno con la relación anterior, y este tipo de situaciones se produce con frecuencia, incluso directamente entre pares de unidades. Por ejemplo, la unidad *Rejilla* contiene la definición de una tabla y de una rejilla de datos para explorar la misma. Las altas y modificaciones sobre esta tabla, en cambio, se realizan en una ventana declarada en la unidad *Dialogo*. *Rejilla* debe hacer referencia a *Dialogo*, para poder lanzar la ventana de edición en respuesta a comandos del usuario. Pero *Dialogo* posiblemente tenga que referirse a su vez a *Rejilla*, para poder acceder a los datos de la tabla con la que trabajamos.

Cuando existen referencias circulares entre pares de unidades, sólo hay que tener en cuenta una regla: debido a problemas técnicos de compilación, no se admiten referencias circulares cuando ambas unidades se refieren entre sí desde sus respectivas interfaces. Una de las unidades debe ser mencionada en la otra mediante una cláusula **uses** ubicada en la sección de implementación.

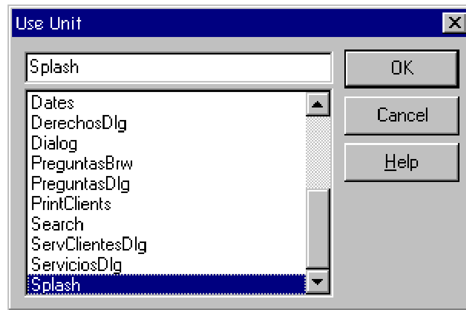
La inclusión automática de unidades

Todas las versiones de Delphi manejan automáticamente la actualización de la cláusula **uses** de la interfaz de una unidad visual, mencionando las unidades necesarias para poder utilizar los componentes que el programador va agregando al diseño. Por ejemplo, si el programador agrega una rejilla de datos a un formulario, en la unidad asociada se añade la unidad *DbCtrls* a su primera cláusula **uses**. Es necesario aclarar que esta cláusula se actualiza solamente cuando se compila o guarda el proyecto, para optimizar el proceso.

Delphi va más allá a partir de la versión 2, pues en ciertos casos puede actualizar de forma automática la cláusula **uses** de la implementación. Esto se produce cuando desde una unidad se hace referencia a una variable de formulario definida en otra unidad del mismo proyecto. Supongamos que desde *Unit1* se utiliza la variable *Form2*, que se refiere a un formulario definido en la unidad *Unit2*; la variable se utiliza dentro

del código de algún procedimiento o función, por supuesto. Al compilar el proyecto, si la segunda unidad no es mencionada desde *Unit1*, Delphi reporta el error, pero nos pregunta si queremos añadir automáticamente la referencia. Si contestamos afirmativamente, se modifica la cláusula **uses** de la implementación, como hemos dicho antes. ¿Por qué precisamente esta cláusula? En primer lugar, porque solamente necesitamos las declaraciones de la misma en el código de la implementación. Y en segundo lugar, para evitar posibles referencias circulares incorrectas.

El proceso de inclusión automática solamente se lleva a cabo para las variables de formulario de las interfaces de las unidades del proyecto. Otras declaraciones realizadas en estas unidades no disparan el proceso. Por ejemplo, mencionar el tipo de datos de un formulario no produce la inclusión automática.



A partir de la versión 2 también se incluye el comando de menú *File|Use unit*. Al ejecutar este comando, Delphi presenta la lista de unidades del proyecto que no están en las cláusulas **uses** de la unidad activa. Si seleccionamos una de ellas y aceptamos el cuadro de diálogo, la unidad seleccionada se añade en la cláusula **uses** de la implementación de la unidad que se encuentra en la página activa el Editor de Código. Esto es casi siempre más conveniente que buscar manualmente la posición de la cláusula, recordar exactamente el nombre de la unidad, teclearlo y regresar a la posición en la que estamos editando.

Los ficheros dfm

¿Dónde, en qué parte del proyecto, se dice que el título del botón *Button1* debe ser *'Púlsame y verás'*? ¿Dónde está escrito que al pulsar este mismo botón se ejecute el método *DestrozarDiscoDuro*? ¿Acaso la respuesta está en el viento? Por supuesto que no: toda esta información se almacena, en tiempo de diseño, en los ficheros de formularios de Delphi, de extensión *dfm* (*Delphi Form*).

El programador diseña una ventana o formulario añadiendo componentes al mismo y especificando valores iniciales para sus propiedades; estos valores iniciales son los

que se almacenan en los ficheros *dfm*. Delphi trata a los eventos de los componentes como propiedades de tipo especial. Por lo tanto, también se colocan en estos ficheros las direcciones de los métodos que dan tratamiento a los eventos. Bueno, en realidad se almacenan los nombres de los métodos y Delphi, al leer el fichero durante la creación del formulario, los convierte en direcciones.

Los ficheros de formularios no son necesarios para la ejecución de la aplicación. Cuando Delphi compila un proyecto, estos ficheros se traducen a un formato binario aceptable por Windows, y se incorporan al ejecutable (o paquete, o DLL), como un *recurso* de Windows. Los *recursos* son datos estáticos que se acoplan a los módulos de códigos, y que luego pueden leerse en tiempo de ejecución mediante funciones estándar de la Interfaz de Programación de Aplicaciones del sistema operativo.

Los ficheros *dfm* no son ficheros de texto, pero pueden convertirse a un formato legible abriéndolos en el Editor de Código de Delphi. Sólo tenemos que activar el diálogo *File|Open*, y elegir el tipo de archivo **.dfm*, o pulsar el botón derecho del ratón sobre un formulario en pantalla, y ejecutar el comando *View as text*. Por ejemplo, este es el código *dfm* correspondiente a un formulario de Delphi 4, con un botón solitario en su interior:

```
object Form1: TForm1
  Left = 200
  Top = 108
  Width = 544
  Height = 375
  Caption = 'Form1'
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  OldCreateOrder = False
  PixelsPerInch = 96
  TextHeight = 13
  object Button1: TButton
    Left = 231
    Top = 162
    Width = 75
    Height = 25
    Caption = 'Button1'
    TabOrder = 0
  end
end
```

Si modificamos este texto, al grabarlo Delphi lo compila nuevamente al formato *dfm*. Sin embargo, no es recomendable utilizar esta técnica a no ser en circunstancias excepcionales. El autor, en particular, sólo la ha empleado para portar formularios de nuevas versiones de Delphi a las anteriores.

La utilidad *convert.exe*, que se encuentra en el directorio de ficheros ejecutables de Delphi, permite convertir un fichero *dfm* a su representación textual, y viceversa.

Bibliotecas de Enlace Dinámico

Otro tipo de proyecto de Delphi son las *Bibliotecas de Enlace Dinámico*, conocidas como DLLs (*Dynamic Link Libraries*). Las DLLs son un componente esencial de la arquitectura de los sistemas operativos Windows, pues permiten el enlace de rutinas en tiempo de carga y ejecución, y posibilitan que varios procesos compartan concurrentemente una misma imagen en memoria del código de la biblioteca. Otra posibilidad es utilizar una DLL como fuente de datos estáticos e inicializados, en particular, mapas de bits, iconos, ficheros de animación y sonido, etc. En principio, además, una DLL puede se utilizada desde una aplicación escrita en un lenguaje diferente al lenguaje original en que fue programada. Son, por lo tanto, un puente natural para la comunicación entre lenguajes.

Desde el punto de vista sintáctico, la diferencia fundamental de una DLL de Delphi con respecto a un proyecto ejecutable está en el fichero de proyecto, cuya sintaxis es ahora la siguiente:

```
library NombreDLL;

uses
        Lista de Unidades;

        Declaraciones

exports
        Lista de rutinas exportadas

begin
        Lista de instrucciones
end.
```

La cláusula **exports**, al final del fichero de proyecto, es fundamental, pues es la que determina las funciones que serán visibles fuera de la DLL. Estas funciones pueden ser declaradas en el mismo fichero de proyecto, en su sección de declaraciones, o pueden estar declaradas en cualquiera de las unidades incluidas en la cláusula **uses**. Ahora bien, si estamos trabajando con Delphi 1, una función necesita para poder ser exportada que haya sido declarada previamente utilizando la directiva **export**. Esta directiva se encarga de generar el código de inicio y fin de la rutina necesario para tratar con el cambio de contexto de ejecución, principalmente, el cambio de dirección base del segmento de código entre el ejecutable (o biblioteca) que llama a la función y la biblioteca donde reside la misma. La siguiente declaración muestra un

ejemplo sencillo de DLL, diseñada para ser utilizada como una ampliación de Inter-Base:

```
library IMUdfs;

type
  TIBDate = record
    Days: LongInt;
    Frac: LongInt;
  end;

function DateDiff(var Date1, Date2: TIBDate): Integer; cdecl;
begin
  Result := Abs(Date1.Days - Date2.Days);
end;

exports
  DateDiff name 'DATEDIFF';
begin
end.
```

En el capítulo 36 trataremos con más detalles la creación y utilización de ficheros DLL con Delphi. Mencionemos también que los proyectos de controles ActiveX, los formularios activos (*ActiveForms*) y las aplicaciones servidoras Web (ISAPI/NSAPI) son casos particulares de bibliotecas dinámicas.

Paquetes

¿Tiene a mano alguna de las versiones de Delphi? En caso afirmativo, cree una nueva aplicación, que contenga únicamente un formulario en blanco, guarde el proyecto y compile. Luego, con el Explorador de Windows busque el fichero ejecutable y apunte el tamaño del mismo. Las cifras que he obtenido son las siguientes:

Delphi 1	Delphi 2	Delphi 3	Delphi 4
187 KB	154 KB	176 KB	274 KB

¡Ciento y tantos miles de bytes para mostrar una ventana vacía en pantalla! ¿Qué es lo que consume tanto espacio? La respuesta es: el núcleo de rutinas básicas de la biblioteca de componentes de Delphi, la VCL. El problema es que, al estar trabajando con clases y objetos, el enlazador no puede eliminar código superfluo con la misma eficacia que en un lenguaje que no sea orientado a objetos. Para disculpar a Delphi digamos que, una vez que comenzamos el desarrollo de nuestro código propio, el crecimiento del fichero ejecutable es bastante mesurado.

Si dos aplicaciones escritas en Delphi se están ejecutando en memoria al mismo tiempo, la parte correspondiente al núcleo de la biblioteca se repite en la memoria RAM del ordenador. "Bueno" – piensa el lector – "¿qué posibilidad tengo de ven-

derle dos aplicaciones en Delphi a un cliente, y que éste las ejecute a la vez?" Dos aplicaciones quizás no, es la respuesta, pero el problema se agrava si estamos trabajando con controles ActiveX. Es muy probable que si estamos utilizando controles ActiveX en una aplicación, necesitemos varios controles de este tipo. Si han sido programados con Delphi, ¡tendremos varias copias del *runtime* del lenguaje activas a la vez! Un problema similar ocurre cuando se incluye uno de estos controles en una página HTML. Cada vez que un cliente se conecta a la página debe cargar vía Internet una y otra vez el entorno de ejecución de Delphi, lo cual es inaceptable en términos de eficiencia.

La solución es relativamente simple, y consiste en mover el código de la biblioteca de componentes, que es común a todos los proyectos, a una biblioteca dinámica, ó DLL. A estas bibliotecas dinámicas de componentes es a lo que Delphi llama *packages*, o *paquetes*. Delphi, para distinguir los paquetes de las bibliotecas dinámicas de otros lenguajes, utiliza la extensión *bpl* (*Borland Package Library*); en Delphi 3 la extensión utilizada era *dpl*. Un fichero con extensión *bpl* no es más que una DLL que contiene el código y los datos necesarios para cierto conjunto de componentes.

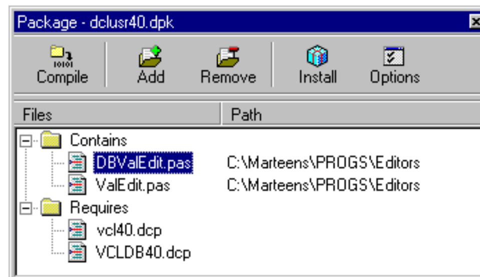
El código fuente de un paquete se almacena en un fichero de extensión *dpk*. La sintaxis de un paquete es muy sencilla, pues solamente consta de un encabezamiento y las cláusulas **contains** y **requires**. El siguiente listado, por ejemplo, corresponde al paquete *dclusr30.dpk* en mi ordenador. Este paquete es el que Delphi ofrece para instalar componentes de prueba en el entorno de desarrollo, por lo cual las unidades mencionadas en la cláusula **contains** no resultarán familiares al lector:

```
package Dclusr40;

{$R *.RES}
{$ALIGN ON}
{ ASSERTIONS ON}
{$BOOLEVAL OFF}
{ DEBUGINFO OFF}
{$EXTENDEDSYNTAX ON}
{$IMPORTEDDATA ON}
{$IOCHECKS ON}
{ LOCALSYMBOLS OFF}
{$LONGSTRINGS ON}
{$OPENSTRINGS ON}
{$OPTIMIZATION ON}
{$OVERFLOWCHECKS OFF}
{$RANGECHECKS OFF}
{$REFERENCEINFO OFF}
{$SAFEDIVIDE OFF}
{$STACKFRAMES OFF}
{$TYPEDADDRESS OFF}
{$VARSTRINGCHECKS ON}
{$WRITEABLECONST ON}
{$MINENUMSIZE 1}
{$IMAGEBASE $00400000}
{$DESCRIPTION 'Borland User''s Components'}
```

```
{$DESIGNONLY}
{$IMPLICITBUILD ON}

requires
  vcl40,
  VCLDB40;

contains
  LeadLib_TLB,
  LImgs,
  HolaXP_TLB,
  CalendarXControl_TLB;
end.
```

La cláusula **contains** menciona las unidades cuyo código va a situarse en el paquete, mientras que **requires** especifica otros paquetes necesarios para la ejecución del actual. Por ejemplo, el paquete *vcldb40*, que contiene el código de los componentes de bases de datos, necesita para su funcionamiento los componentes estándar de Delphi, que se encuentran en *vcl40*; el texto fuente de *vcldb40.dpk* hace referencia en su cláusula **requires** al paquete *vcl40*. La mayor parte del listado anterior, sin embargo, la consumen las directivas de compilación, que han sido situadas explícitamente en el mismo.

Aunque podemos crear directamente un fichero *dpk*, editarlo manualmente y compilarlo, es mas sencillo utilizar el editor especializado de Delphi, que se muestra a continuación:



Esta ventana aparece cada vez que creamos un nuevo paquete, o abrimos uno existente. Para añadir unidades al paquete basta con pulsar el botón *Add*. Para modificar las opciones, tenemos el botón *Options*. Además, se puede pulsar el botón derecho del ratón para acceder al menú local que, entre otros comandos, nos permite almacenar en disco el texto generado para el paquete, o activar el Editor de Código para trabajar directamente sobre el *dpk*.

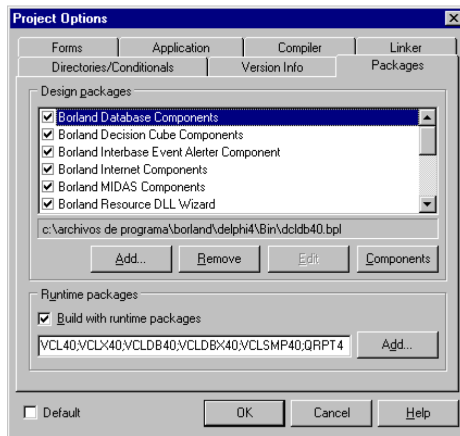
La compilación de un paquete produce, además del paquete en sí (el fichero de extensión *bpl*) otro fichero con el mismo nombre que el fuente y con extensión *dcp* (*Delphi Compiled Package*). Este fichero contiene, sencillamente, todas las unidades

compiladas en formato *dcu* concatenadas. De este modo, para compilar un código que hace uso de paquetes no necesitamos todos los ficheros *dcu* sueltos de cada unidad contenida, lo cual es conveniente.

> ¿Por qué Delphi recurre a los paquetes, en vez de utilizar sencillamente DLLs? La razón consiste en que si no tenemos cuidado, la información de clases y tipos en tiempo de ejecución puede duplicarse en los distintos módulos (ejecutables y bibliotecas), dando lugar a una serie de anomalías que describiremos en el capítulo 36. Los paquetes evitan este problema en tiempo de compilación, al exigir que cada unidad aparezca solamente en un paquete.

El ABC del uso de paquetes

Con esta nueva característica, podemos seguir generando aplicaciones "a la vieja usanza", con todo el código enlazado estáticamente, o podemos elegir el uso de paquetes, para producir ejecutables de menor tamaño. Si queremos esto último, solamente tenemos que activar el comando de menú *Project|Options*, buscar la página *Packages*, y marcar la casilla *Build with runtime packages*, en la parte inferior de la página. Con la opción anterior activada, logramos adicionalmente un menor tiempo de enlace, porque el fichero generado en cada compilación ocupa mucho menos espacio.



Debajo de la casilla mencionada, hay un cuadro de edición con la lista de los paquetes que la aplicación va a utilizar. Podemos eliminar paquetes de esta lista, o añadir alguno nuevo. En cuanto a la lista de paquetes que aparece en la parte superior de la página, explicaremos su uso en el siguiente epígrafe.

El mayor problema del uso de paquetes es cómo identificar a cuál de ellos pertenece un componente determinado, y cómo incluirlo en el programa de instalación de

nuestra aplicación. Lo primero tiene una solución razonablemente sencilla: los manuales de Borland y la ayuda en línea nos dicen claramente dónde se encuentra cada componente de los que trae Delphi. En cuanto a la segunda dificultad, no es un problema en Delphi 4, pues la versión de InstallShield que acompaña a este producto nos permite automatizar esta tarea. Para Delphi 3, en cambio, las noticias no son tan buenas, pues la correspondiente versión de InstallShield no soporta la instalación automática de paquetes. Por el contrario, hay que incluir los ficheros *dpl* (recuerde que esta era la extensión utilizada por esta versión de Delphi) explícitamente en el proyecto de instalación. El último capítulo de este libro lo dedicaremos a InstallShield, la herramienta de creación de instalaciones que viene con Delphi, y explicaremos como realizar una instalación con paquetes.

Paquetes de tiempo de diseño y de ejecución

Delphi realiza una clara separación entre paquetes de tiempo de ejecución y de diseño, aunque físicamente ambos son ficheros de extensión *bpl*. La razón para esta distinción es la peculiar arquitectura de componentes de Delphi. Con frecuencia, para poder utilizar un componente en el diseño de una aplicación es necesario disponer, además del código asociado al componente, de editores de propiedades, expertos y otras rutinas para la asistencia durante el desarrollo. Estas rutinas, sin embargo, no son necesarias para la ejecución posterior de la aplicación.

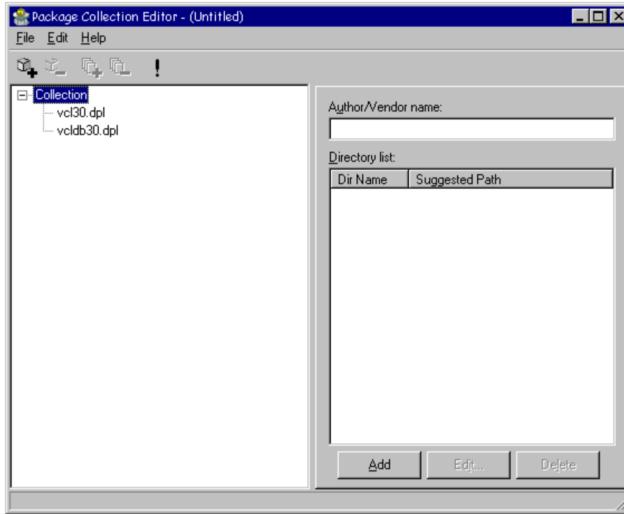
Las siguientes directivas de compilación controlan la clasificación de un paquete:

```
{$RUNONLY ON}
{$DESIGNONLY ON}
```

El propio Delphi es, por lo tanto, el primer beneficiario de la arquitectura de paquetes, pues el Entorno de Desarrollo basa su funcionamiento en los mismos. En versiones anteriores del producto, la biblioteca de componentes que se utilizaba durante el desarrollo residía en un fichero monolítico, llamado *complib.dcl* ó *complib32.dcl*, en dependencia de la versión. Por lo tanto, instalar o eliminar un componente provocaba la reconstrucción de este enorme fichero. Ahora para agregar un nuevo componente basta con registrar el paquete de tiempo de diseño en el cuál viene el mismo. Si solamente poseemos el paquete de tiempo de ejecución, podemos añadir la unidad en que se encuentra el componente al paquete *dclusr40.dpk*, cuyo código fuente ofrece Delphi. Este paquete es precisamente el que hemos listado con anterioridad.

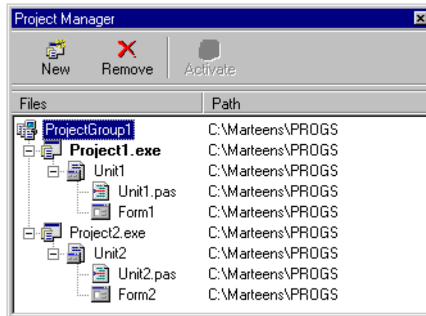
Incluso podemos eliminar temporalmente de la Paleta de Componentes los paquetes que son innecesarios para un proyecto dado. Esto lo podemos realizar en la página *Packages* del cuadro de diálogo de opciones del proyecto, utilizando la lista que se encuentra en la parte superior de la página.

Para facilitar la distribución de paquetes a otros programadores, Delphi ofrece la herramienta *Package Collection Editor*, que permite comprimir varios paquetes en un mismo fichero de extensión *dpc* (*Delphi Package Collection*). Estas colecciones pueden después instalarse automáticamente desde el comando de menú *Components|Install packages*.



Los grupos de proyectos

Mientras más se complican las cosas, más organizados debemos ser. Muchas veces no basta con un fichero ejecutable para echar a andar una aplicación. Podemos necesitar, por ejemplo, otros ejecutables auxiliares, bibliotecas dinámicas (también conocidas como DLLs), servidores de automatización, paquetes de componentes, ficheros de recursos, etc. ¿No sería apropiado poder organizar todos estos elementos dentro de una sola entidad? Esto es lo que intentan hacer los *grupos de proyectos* de Delphi 4. La siguiente imagen muestra el nuevo Administrador de Proyectos, con un grupo de proyectos activo que contiene dos programas ejecutables:



El grupo de proyectos se almacena en un fichero, de extensión *bpg* (*Borland Project Group*), cuyo contenido corresponde aproximadamente al de un fichero *make* de otros lenguajes:

```
#------------------------------------------------------------------------------
VERSION = BWS.01
#------------------------------------------------------------------------------
!ifndef ROOT
ROOT = $(MAKEDIR)\..
!endif
#------------------------------------------------------------------------------
MAKE = $(ROOT)\bin\make.exe -$(MAKEFLAGS) -f$**
DCC = $(ROOT)\bin\dcc32.exe $**
BRCC = $(ROOT)\bin\brcc32.exe $**
#------------------------------------------------------------------------------
PROJECTS = Project1.exe Project2.exe
#------------------------------------------------------------------------------
default: $(PROJECTS)
#------------------------------------------------------------------------------

Project1.exe: Project1.dpr
  $(DCC)

Project2.exe: Project2.dpr
  $(DCC)
```

Siempre hay un proyecto activo dentro de un grupo de proyectos, por lo cual podemos olvidarnos de la existencia del grupo y seguir trabajando igual que antes. Tampoco hay problemas con obviar totalmente los grupos de proyectos, pues Delphi permite que sigamos trabajando directamente con los ficheros *dpr* de toda la vida.

Capítulo

4

Sistemas de bases de datos

ESTE ES 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 recomiendo 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 siguiente capítulo. 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 hubieran permitido terminar mucho antes la aplicación, y que ésta se ejecutara más eficientemente.

Bases de datos relacionales

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.

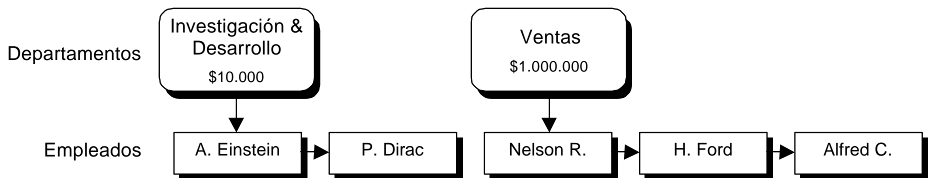
El modelo de redes era más flexible. Un registro podía contener un conjunto de otros registros. Y cada uno de estos registros podía pertenecer a más de un conjunto. La implementación más frecuente de esta característica se realizaba mediante punteros. El sistema más famoso que seguía este modelo, curiosamente, fue comprado por cierta compañía tristemente conocida por comprar sistemas de bases de datos y 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 cuantos 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 quizás menos popular *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[4].

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*.

[4] 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.

¡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 vínculo[5] 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 paso siguiente son 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

---

[5] 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.

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.

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 desígnela 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 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 alguien que aprovecha inteligentemente la existencia de claves primarias: el Motor de Datos de Borland. De hecho, estas claves primarias son la forma en que el Motor de Datos puede simular el concepto de registro activo en una tabla SQL. Pero tendremos que esperar un poco para desentrañar este ingenioso mecanismo.

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 siste-

mas 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 prohibe la modificación de la clave primaria de un registro que tenga filas de detalles asociadas.
- Alternativamente, la modificación de la clave se propaga 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 más adelante, en el capítulo 21).

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 opera-

ciones. Supongamos que tenemos un registro de empleados 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.

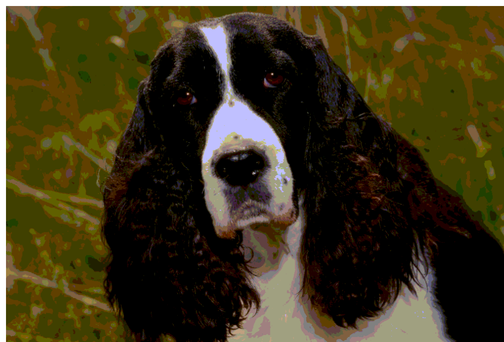


Ilustración 1 El perro de Codd

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

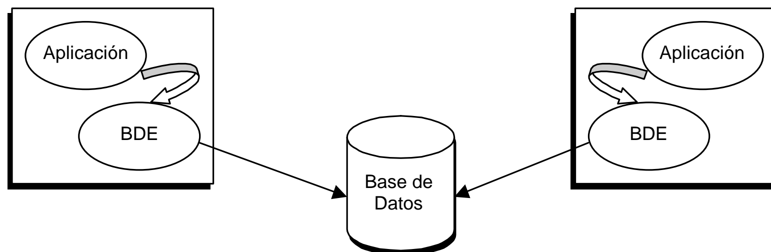
en su vida anterior, de no ser por la manía que tiene de rascarse periódicamente la oreja. No obstante, por haber destrozado la tapicería del sofá de su dueño y orinarse un par de veces en la alfombra del salón, fue condenado a programar dos largos años en Visual Basic, hasta que Delphi llegó a su vida 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 de bases de datos 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":

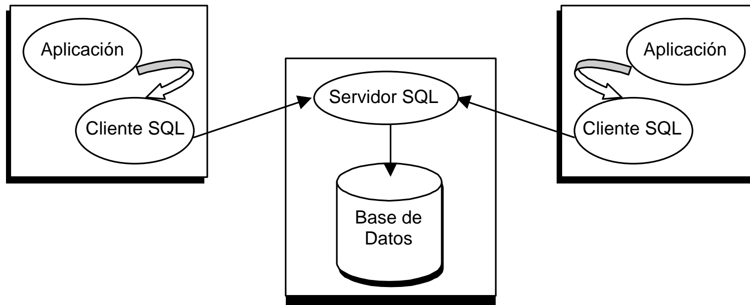


Aunque la segunda burbuja dice "BDE", el Motor de Datos de Borland, sustituya estas siglas por DAO, y podrá aplicar el diagrama a Access. He representado la aplicación y el motor de datos en dos burbujas separadas, pero en realidad, constituyen un mismo ejecutable. Lo más importante, sin embargo, es 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, olvídese 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 pedírselo 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; en cualquiera de estos dos casos, los *triggers* que se hayan programado se activarán.

Estos recursos se estudiarán más adelante, cuando exploremos el lenguaje SQL.

El formato Paradox

Comenzaremos nuestra aventura explicando algunas características de los sistemas de bases de datos de escritorio que pueden utilizarse directamente con Delphi. De estos, mi actual favorito es Paradox. Pongo el adjetivo "actual" porque el nuevo formato de Visual dBase VII comienza a aproximarse a la riqueza expresiva soportada desde hace mucho por Paradox. Dedicaremos a dBase la siguiente sección.

Paradox no reconoce directamente el concepto de *base de datos*, sino que administra tablas en ficheros independientes. Cada tabla se almacena en un conjunto de ficheros, todos con el mismo nombre, pero con las extensiones que explicamos a continuación:

Extensión	Explicación
.db	Definición de la tabla y campos de longitud máxima fija
.mb	Campos de longitud variable, como los memos y gráficos
.px	El índice de la clave primaria
.Xnn, .Ynn	Indices secundarios
.val	Validaciones e integridad referencial

En el fichero *.db* se almacena una cabecera con la descripción de la tabla, además de los datos de los registros que corresponden a campos de longitud fija. Este fichero está estructurado en bloques de idéntico tamaño; se pueden definir bloques de 1, 2, 4, 8, 16 y 32KB. El tamaño de bloque se determina durante la creación de la tabla mediante el parámetro *BLOCK SIZE* del controlador de Paradox del BDE (por omisión, 2048 bytes); en el próximo capítulo aprenderemos a gestionar éste y muchos otros parámetros. El tamaño de bloque influye en la extensión máxima de la tabla, pues Paradox solamente permite 2^{16} bloques por fichero. Si se utiliza el tamaño de bloque por omisión tendríamos el siguiente tamaño máximo por fichero de datos:

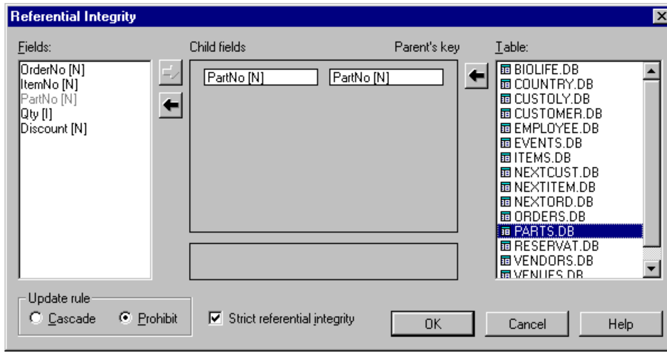
$$2048 \times 65536 = 2^{11} \times 2^{16} = 2^{27} = 2^{7} \times 2^{20} = 128\text{MB}$$

Un registro debe caber completamente dentro de un bloque, y si la tabla tiene una clave primaria (¡recomendable!) deben existir al menos tres registros por bloque; esto se refiere a los datos de longitud fija del registro, excluyendo campos de texto largos, de imágenes y binarios. Dentro de cada bloque, los registros se ordenan según su clave primaria, para agilizar la búsqueda una vez que el bloque ha sido leído en la caché. Un registro, además, siempre ocupa el mismo espacio dentro del fichero *.db*, incluso si contiene campos alfanuméricos que, a diferencia de lo que sucede en dBase, no se rellenan con blancos hasta ocupar el ancho total del campo.

Como explicamos anteriormente, el índice primario de la tabla se almacena en el fichero de extensión *.px*. No es necesario que una tabla tenga índice primario, pero es altamente ventajoso. Al estar ordenados los registros dentro los bloques de datos, sólo se necesita almacenar en el índice la clave del primer registro de cada bloque. Esto disminuye considerablemente los requerimientos de espacio del índice, y acelera las búsquedas.

Es interesante conocer cómo Paradox implementa los índices secundarios. En realidad, cada índice secundario es internamente una tabla, con su propio fichero de datos, de extensión *.Xnn*, y su índice asociado *.Ynn*. La numeración de los índices sigue un esquema sencillo: los dos últimos caracteres de la extensión indican el número del campo en hexadecimal, si es un índice sobre un solo campo. Si es un índice compuesto, se utiliza una pseudo numeración hexadecimal, pero comenzando desde el "valor" *G0* y progresando en forma secuencial. Desde el punto de vista del usuario, los índices secundarios tienen un nombre asociado. El índice primario, en cambio, no tiene nombre.

Paradox permite la definición de relaciones de integridad referencial. La siguiente imagen muestra el diálogo de Database Desktop que lo hace posible:



El motor de datos crea automáticamente un índice secundario sobre las columnas de la tabla dependiente que participan en una restricción de integridad. Si la restricción está basada en una sola columna, el índice recibe el nombre de la misma. La tabla que incluyo a continuación describe la implementación en Paradox del comportamiento de la integridad referencial respecto a actualizaciones en la tabla maestra:

	Borrados	Modificaciones
Prohibir la operación	Sí	Sí
Propagar en cascada	No	Sí
Asignar valor por omisión	No	-

No se permiten los borrados en cascada. No obstante, este no es un impedimento significativo, pues dichos borrados pueden implementarse fácilmente en las aplicaciones clientes. En cambio, trate el lector de implementar una propagación en cascada de un cambio de clave cuando existe una integridad referencial...

No fue hasta la aparición de la versión 3.0 del BDE, que acompañó a Delphi 2, que Paradox y dBase contaron con algún tipo de soporte para transacciones. No obstante, este soporte es actualmente muy elemental. Utiliza un *undo log*, es decir, un fichero en el que se van grabando las operaciones que deben deshacerse. Si se desconecta la máquina durante una de estas transacciones, los cambios aplicados a medias quedan grabados, y actualmente no hay forma de deshacerlos. La independencia entre transacciones lanzadas por diferentes procesos es la mínima posible, pues un proceso puede ver los cambios efectuados por otro proceso aunque éste no haya confirmado aún toda la transacción.

Finalmente, estas transacciones tienen una limitación importante. La contención entre procesos está implementada mediante bloqueos. Actualmente Paradox permite hasta 255 bloqueos por tabla, y dBase solamente 100. De modo que una transacción en Paradox puede modificar directamente un máximo de 255 registros.

El formato DBF7

¿Quién no ha tenido que trabajar, en algún momento y de una forma u otra, con los conocidos ficheros DBF? La popularidad de este formato se desarrolló en paralelo con la aparición de los PCs y la propagación de MS-DOS. En el principio, era propiedad de una compañía llamada Ashton-Tate. Ashton fue el empresario que compró el software a su desarrollador, y tenía un loro llamado Tate, ¿o era al revés? Después a esta empresa le fueron mal las cosas y Borland adquirió sus productos. Aunque muchos interpretaron esta acción como un reconocimiento a los "méritos" técnicos de dBase, se trataba en realidad de adquirir otro producto de bases de datos, que en aquel momento era un perfecto desconocido: InterBase. Hay que advertir que InterBase había sido a su vez comprado por Ashton-Tate a sus desarrolladores originales, una efímera compañía de nombre Groton Database Systems, cuyas siglas aún perduran en la extensión por omisión de los ficheros de InterBase: *gdb*.

El formato DBF es muy sencillo ... y muy malo. Fue diseñado como "la base de datos de los pobres". Cada tabla, al igual que sucede en Paradox, se representa en un conjunto de ficheros. El fichero de extensión *dbf* contiene a la vez la descripción de los campos y los registros, estos últimos de longitud fija. Los registros se almacenan de forma secuencial, sin importar las fronteras de bloques y la pérdida de eficiencia que esto causa. Los tipos de datos originales se representaban en formato ASCII legible, en vez de utilizar su codificación binaria, más compacta. De este modo, las fechas se representaban en ocho caracteres, con el formato *AAAAMMDD*, y un valor entero de 16 bits ocupaba 40 bits, o 48 si se representaba el signo. Por otra parte, hasta fechas recientes el único tipo de cadena implementado tenía longitud fija, obligando a los programadores a usar exhaustivamente la función *Trim*.

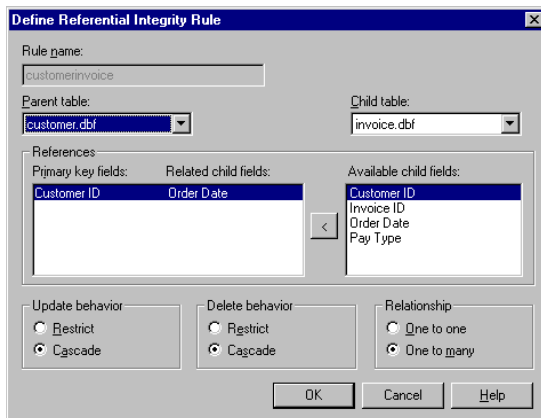
Se pueden utilizar campos de longitud variable, que en principio estuvieron limitados al tipo texto, o *memo*. Los valores de estos campos se almacenan en un fichero paralelo de extensión *dbt*. En este fichero los datos sí respetan la frontera de bloque. Por omisión, el tamaño de estos bloques es de 1024 bytes, pero este parámetro es configurable. Los índices, por su parte, se almacenan todos en un fichero de extensión *mdx*. Aunque, en principio, una tabla puede tener varios ficheros *mdx* asociados, solamente los índices que se encuentran en el fichero que tiene el mismo nombre que la tabla se actualizan automáticamente, por lo que en la práctica solamente cuenta este fichero, denominado *índice de producción*.

Una de las peculiaridades de dBase es la forma en que se implementan los índices sobre varias columnas. dBase no tiene índices compuestos, al estilo de Paradox y de los sistemas SQL. Como alternativa ofrece los índices basados en expresiones: si queremos un índice compuesto por las columnas *Nombre* y *Apellidos*, tenemos que definir un índice por la expresión *Nombre+Apellidos*. Puede parecer que ésta es una opción muy potente, pero en la práctica el uso de los índices sobre expresiones se reduce a sustituir los índices compuestos de otros sistemas. Además, veremos que

trabajar con estos índices en Delphi es bastante penoso. ¿Claves primarias, integridad referencial? Nada de eso tenía el formato DBF. Cuesta más trabajo rediseñar una aplicación escrita en Delphi con dBase para que se ejecute en un sistema cliente/servidor que si hubiera sido desarrollada para Paradox.

¿Algo a favor de este formato? Las dos cualidades que mencionaré son consecuencia de la simplicidad de sus ficheros. En primer lugar, las operaciones de actualización son un poco más rápidas en dBase que en Paradox. Y, lo más importante, al existir "menos cosas para romperse", hay más estabilidad en este formato y más tolerancia respecto a fallos en el sistema operativo.

Con la aparición de Visual dBase 7 junto a la versión 4.50 del BDE, el formato DBF fue renovado en gran medida. Una de las novedades más importantes es la incorporación de relaciones de integridad referencial, que incluyen los borrados en cascada, como se muestra en el siguiente cuadro de diálogo del Administrador de Bases de Datos de Visual dBase 7:



También se han añadido tipos de datos representados en forma binaria, condiciones de verificación, que consisten en expresiones lógicas que deben satisfacer siempre las filas de una tabla, valores por omisión, etc. Lamentablemente, seguimos con los índices de expresiones como sustitutos de los índices compuestos, y el límite de registros modificados por transacción sigue siendo 100.

Criterios para evaluar un servidor SQL

¿Cómo saber cuál es el sistema de bases de datos cliente/servidor más adecuado para nuestros objetivos? En esta sección trato de establecer criterios de comparación para estos sistemas. No hay dos aplicaciones con las mismas necesidades, así que el hecho

de que un sistema no ofrezca determinada opción no lo invalida como la solución que usted necesita. Recuerde que no siempre más es mejor.

- **Plataformas soportadas**
  ¿En qué sistema operativo debe ejecutarse el servidor? La respuesta es importante por dos razones. La primera: nos da una medida de la flexibilidad que tenemos a la hora de seleccionar una configuración de la red. Si vamos a instalar una aplicación en una empresa que tiene toda su red basada en determinado servidor con determinado protocolo, no es recomendable que abandonen todo lo que tienen hecho para que el recién llegado pueda ejecutarse.
  En segundo lugar, no todos los sistemas operativos se comportan igual de eficientes actuando como servidores de bases de datos. Esto tiene que ver sobre todo con la implementación que realiza el SO de la concurrencia. Mi favorito en este sentido es UNIX: un InterBase, un Oracle, un Informix, ejecutándose sobre cualquier "sabor" de UNIX (HP-UX, AIX, Solaris).
  Puede suceder, por supuesto, que el mantenimiento de la red no esté en manos de un profesional dedicado a esta área. En tal caso, hay que reconocer que es mucho más sencillo administrar un Windows NT.

- **Soporte de tipos de datos y restricciones**
  En realidad, casi todos los sistemas SQL actuales tienen tipos de datos que incluyen a los especificados en el estándar de SQL, excepto determinados casos patológicos. De lo que se trata sobre todo es la implementación de los mismos. Por ejemplo, no todas los formatos de bases de datos implementan con la misma eficiencia el tipo *VARCHAR*, que almacena cadenas de longitud variable. La longitud máxima es uno de los parámetros que varían, y el formato de representación: si siempre se asigna un tamaño fijo (el máximo) para estos campos, o si la longitud total del registro puede variar.
  Más importante es el soporte para validaciones declarativas, esto es, verificaciones para las cuales no hay que desarrollar código especial. Dentro de este apartado entran las claves primarias, claves alternativas, relaciones de integridad referencial, dominios, chequeos a nivel de fila, etc. Un elemento a tener en cuenta, por ejemplo, es si se permite o no la propagación en cascada de modificaciones en la tabla maestra de una relación de integridad referencial. En caso positivo, esto puede ahorrarnos bastante código en *triggers* y permitirá mejores modelos de bases de datos.

- **Lenguaje de triggers y procedimientos almacenados**
  Este es uno de los criterios a los que concedo mayor importancia. El éxito de una aplicación, o un conjunto de aplicaciones, en un entorno C/S depende en gran medida de la forma en que dividamos la carga de la aplicación entre el servidor de datos y los clientes. En este punto es donde podemos encontrar mayores diferencias entre los sistemas SQL, pues no hay dos dialectos de este lenguaje que sean exactamente iguales.

Debemos fijarnos en si el lenguaje permite *triggers* a nivel de fila o de operación, o de ambos niveles. Un *trigger* a nivel de fila se dispara antes o después de modificar una fila individual. Por el contrario, un *trigger* a nivel de operación se dispara después de que se ejecute una operación completa que puede afectar a varias filas. Recuerde que SQL permite instrucciones como la siguiente:

```
delete from Clientes
where Ciudad in ("Pompeya", "Herculano")
```

Si la base de datos en cuestión sólo ejecuta sus *triggers* al terminar las operaciones, será mucho más complicada la programación de los mismos, pues más trabajo costará restablecer los valores anteriores y posteriores de cada fila particular. También debe tenerse en cuenta las posibilidades de extensión del lenguaje, por ejemplo, incorporando funciones definidas en otros lenguajes, o el poder utilizar servidores de automatización OLE o CORBA, si se permiten o no llamadas recursivas, etc.

- **Implementación de transacciones: recuperación y aislamiento**
  Este es mi otro criterio de análisis preferido. Las transacciones ofrecen a las bases de datos la consistencia necesaria para que operaciones parciales no priven de sentido semántico a los datos almacenados. Al constituir las unidades básicas de procesamiento del servidor, el mecanismo que las soporta debe encargarse también de aislar a los usuarios entre sí, de modo que la secuencia exacta de pasos concurrentes que realicen no influya en el resultado final de sus acciones. Por lo tanto, hay dos puntos en los que centrar la atención: cómo se implementa la atomicidad (la forma en que el sistema deshace operaciones inconclusas), y e método empleado para aislar entre sí las transacciones concurrentes.

- **Segmentación**
  Es conveniente poder distribuir los datos de un servidor en distintos dispositivos físicos. Al situar tablas en distintos discos, o segmentos de las propias tablas, podemos aprovechar la concurrencia inherente a la existencia de varios controladores físicos de estos medios de almacenamiento. Esta opción permite, además, superar las restricciones impuestas por el tamaño de los discos en el tamaño de la base de datos.

- **Replicación**
  Uno de los usos principales de la replicación consiste en aislar las aplicaciones que realizan mantenimientos (transacciones OLTP) de las aplicaciones para toma de decisiones (transacciones DSS). Las aplicaciones con fuerte base OLTP tienden a modificar en cada transacción un pequeño número de registros. Las aplicaciones DSS se caracterizan por leer grandes cantidades de información, siendo poco probable que escriban en la base de datos. En sistemas de bases de datos que implementan el aislamiento de transacciones mediante bloqueos, ambos ti-

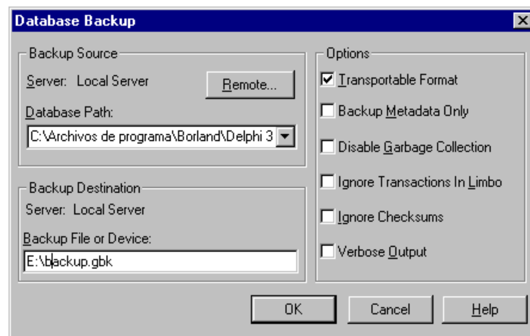
pos de transacciones tienen una coexistencia difícil. Por lo tanto, es conveniente disponer de réplicas de la base de datos sólo para lectura, y que sea a estos clones a quienes se refieran las aplicaciones DSS.

Me da un poco de vergüenza añadir el último factor a la lista anterior: el precio. Todos queremos lo mejor, pero no siempre estamos dispuestos a pagar por eso. Así que muchas veces una decisión de compra representa un balance entre la calidad y el precio ... como en todo.

InterBase

Los antiguos griegos representaban a la Fama y a la Fortuna como caprichosas diosas que otorgaban sus favores muchas veces a quienes menos lo merecían; estos griegos de antes, la verdad, eran bastante misóginos, entre otras cosas. En cualquier caso, InterBase debe haber tenido algún altercado con la señorita Fama, porque no tiene la popularidad que merecería por sus muchas virtudes.

Una de estas virtudes es que se puede instalar el servidor en muchos sistemas operativos: Windows NT y 9x, NetWare, Solaris, HP-UX, SCO, Linux... Otra es que en cualquiera de estos sistemas ocupa muy poco espacio, del orden de los 10 ó 20 MB. El mantenimiento de un servidor, una vez instalado, es también mínimo. Cada base de datos se sitúa en uno o más ficheros, casi siempre de extensión *gdb*. Estos ficheros crecen automáticamente cuando hace falta más espacio, y no hay que preocuparse por reservar espacio adicional para registrar los cambios efectuados por las transacciones, como en otros sistemas. Se pueden realizar copias de seguridad de una base de datos "en caliente", mientras que otros sistemas requieren que no existan usuarios activos para efectuar esta operación. Y la recuperación después de un fallo de hardware es sencilla e inmediata: vuelva a encender el servidor.



Tal sencillez tiene un precio, y es que actualmente InterBase no implementa directamente ciertas opciones avanzadas de administración, como la segmentación y la replicación. Esta última debe ser implementada manualmente, por medio de *triggers*

definidos por el diseñador de la base de datos, y con la ayuda de un proceso en segundo plano que vaya grabando los datos del original a la réplica. No obstante, InterBase permite definir directamente copias en espejo (*mirrors*) de una base de datos, de forma tal que existan dos copias sincronizadas de una misma base de datos en discos duros diferentes del mismo servidor. De este modo, si se produce un fallo de hardware en uno de los discos o controladores, la explotación de la base de datos puede continuar con la segunda copia.

En lo que atañe a los tipos de datos, InterBase implementa todos los tipos del estándar SQL con una excepción. Este lenguaje define tres tipos de campos para la fecha y la hora: *DATE*, para fechas, *TIME*, para horas, y *TIMESTAMP* para almacenar conjuntamente fechas y horas. InterBase solamente tiene *DATE*, pero como equivalente al *TIMESTAMP* del estándar. Esto en sí no conlleva problemas, a no ser que haya que escribir una aplicación que acceda indistintamente a bases de datos de InterBase y de cualquier otro sistema SQL. Pero cuando estudiemos el uso del Diccionario de Datos de Delphi, veremos cómo resolver esta nimiedad.

InterBase tiene, hoy por hoy, una de las implementaciones más completas de las restricciones de integridad referencial, pues permite especificar declarativamente la propagación en cascada de borrados y modificaciones:

	Borrados	**Modificaciones**
Prohibir la operación	Sí	Sí
Propagar en cascada	Sí	Sí
Asignar valor por omisión	Sí	-

Por supuesto, tenemos todas las restricciones de unicidad, claves primarias, validaciones a nivel de registro (cláusulas **check**). Estas últimas son más potentes que en el resto de los sistemas, pues permiten realizar comprobaciones que involucren a registros de otras tablas.

Los *triggers* de InterBase se ejecutan antes o después de cada operación de actualización, fila por fila, que es como Dios manda. Por otra parte, tiene un lenguaje de procedimientos almacenados muy completo, que permite llamadas recursivas, y la definición de procedimientos de selección, que devuelven más de un registro de datos por demanda. Todo ello se integra con un mecanismo de excepciones muy elegante, que compagina muy bien con las técnicas transaccionales. Es posible, además, extender el conjunto de funciones del lenguaje mediante módulos dinámicos, DLLs en el caso de las versiones para Windows y Windows NT.

Pero la característica más destacada de InterBase es la forma en la que logra implementar el acceso concurrente a sus bases de datos garantizando que, en lo posible, cada usuario no se vea afectado por las acciones del resto. Casi todos los sistemas existentes utilizan, de una forma u otra, bloqueos para este fin. InterBase utiliza la

denominada *arquitectura multigeneracional*, en la cual cada vez que una transacción modifica un registro se genera una nueva versión del mismo. Teóricamente, esta técnica permite la mayor cantidad de acciones concurrentes sobre las bases de datos. En la práctica, la implementación de InterBase es muy eficiente y confiable. Todo esto lo estudiaremos en el capítulo 31, que trata acerca de las transacciones.

La última versión de InterBase, en el momento en que escribo estas líneas, es la 5, aunque existe una versión de actualización que corrige ciertos pequeños problemas en las herramientas de diagnósticos y administración.

Microsoft SQL Server

Toda semejanza en la explicación de las características de Microsoft SQL Server con las de Sybase será algo más que una simple coincidencia. Realmente MS SQL Server comenzó como un derivado del servidor de Sybase, por lo que la arquitectura de ambos es muy parecida. De modo que gran parte de lo que se diga en esta sección sobre un sistema, vale para el otro.

Es muy fácil tropezarse por ahí con MS SQL Server. Hay incluso quienes lo tienen instalado y aún no se han dado cuenta. Microsoft tiene una política de distribución bastante agresiva para este producto, pues lo incluye en el paquete BackOffice, junto a su sistema operativo Windows NT y unos cuantos programas más. Como puede imaginar el lector, SQL Server 6.5 puede ejecutarse en Windows NT y en Windows NT. Por fortuna, la próxima versión 7.0 ofrece un servidor para Windows 9x ... aunque obliga al usuario a instalar primeramente Internet Explorer 4[6]. Está previsto que la versión 7.0 comience a comercializarse a partir de septiembre/octubre del 98, cuando este libro ya esté en la calle. La información que ofrecemos sobre la nueva versión ha sido obtenida de las versiones Beta, así que júzguela adecuadamente.

Lo primero que llama la atención es la cantidad de recursos del sistema que consume una instalación de SQL Server. La versión 6.5 ocupa de 70 a 90MB, mientras que la beta que he examinado llega a los 180MB de espacio en disco. ¿La explicación? Asistentes, y más asistentes: en esto contrasta con la austeridad espartana de InterBase.

Aunque la instalación del servidor es relativamente sencilla, su mantenimiento es bastante complicado, sobre todo en la versión 6.5. Cada base de datos reside en uno o más *dispositivos físicos*, que el fondo no son más que vulgares ficheros. Estos dispositivos ayudan a implementar la segmentación, pero no crecen automáticamente, por lo que el administrador del sistema deberá estar pendiente del momento en que los datos están a punto de producir un desbordamiento. A diferencia de InterBase, para

---

[6] Al menos en versiones Beta.

cada base de datos hay que definir explícitamente un *log*, o registro de transacciones, que compite en espacio con los datos verdaderos, aunque este registro puede residir en otro dispositivo (lo cual se recomienda).

Aunque los mecanismos de *logging*, en combinación con los bloqueos, son los más frecuentes en las bases de datos relacionales como forma de implementar transacciones atómicas, presentan claras desventajas en comparación con la arquitectura multigeneracional. En primer lugar, no se pueden realizar copias de seguridad con usuarios conectados a una base de datos. Los procesos que escriben bloquean a los procesos que se limitan a leer información, y viceversa. Si se desconecta físicamente el servidor, es muy probable que haya que examinar el registro de transacciones antes de volver a echar a andar las bases de datos. Por último, hay que estar pendientes del crecimiento de estos ficheros. Hay un experimento muy sencillo que realizo con frecuencia en InterBase, poblar una tabla con medio millón de registros, que nunca he logrado repetir en SQL Server, por mucho que he modificado parámetros de configuración.

Hay que reconocer que esta situación mejora un poco con la versión 7, pues desaparece el concepto de dispositivo, siendo sustituido por el de fichero del propio sistema operativo: las bases de datos se sitúan en ficheros de extensión *mdf* y *ndf*, los registros de transacciones, en ficheros *ldf*. Este nuevo formato permite que los ficheros de datos y de transacciones crezcan dinámicamente, por demanda.

Otro grave problema de versiones anteriores que soluciona la nueva versión es la granularidad de los bloqueos. Antes, cada modificación de un registro imponía un bloqueo a toda la página en que se encontraba. Además, las páginas tenían un tamaño fijo de 2048 bytes, lo que limitaba a su vez el tamaño máximo de un registro. En la versión 6.5 se introdujo el bloqueo a nivel de registro ... pero únicamente para las inserciones, que es cuando menos hacen falta. Finalmente, la versión 7 permite siempre bloqueos de registros, que pueden escalarse por demanda a bloqueos de página o a nivel de tablas, y aumenta el tamaño de página a 8192 bytes. No obstante, este tamaño sigue sin poder ajustarse.

Microsoft SQL Server ofrece extrañas extensiones a SQL que solamente sirven para complicarnos la vida a usted y mí. Por ejemplo, aunque el SQL estándar dice que por omisión una columna admite valores nulos, esto depende en SQL Server del estado de un parámetro, ¡que por omisión produce el efecto contrario! La implementación de la integridad referencial en este sistema es bastante pobre, pues solamente permite restringir las actualizaciones y borrados en la tabla maestra; nada de propagación en cascada y otras alegrías. También es curioso que SQL Server no crea automáticamente índices secundarios sobre las tablas que contienen claves externas.

	Borrados	**Modificaciones**
Prohibir la operación	Sí	Sí
Propagar en cascada	No	No
Asignar valor por omisión	No	-

Otro de los aspectos negativos de SQL Server es su lenguaje de *triggers* y procedimientos almacenados, llamado Transact-SQL, que es bastante excéntrico respecto al resto de los lenguajes existentes y a la propuesta de estándar. Uno puede acostumbrarse a soberanas tonterías tales como obligar a que todas las variables locales y parámetros comiencen con el carácter @. Pero es bastante difícil programar determinadas reglas de empresa cuando los *triggers* se disparan solamente después de instrucciones completas.

El aislamiento de transacciones deja bastante que desear, pues el nivel superior solamente se alcanza en bases de datos abiertas en modo sólo lectura.

Oracle

Oracle es uno de los abuelos en el negocio de las bases de datos relacionales; el otro es DB2, de IBM[7]. Este sistema es otra de las apuestas seguras en el caso de tener que elegir un servidor de bases de datos. ¿Su principal desventaja? Resulta que no es de carácter técnico, sino que tiene que ver con una política de precios altos, alto coste de la formación y del mantenimiento posterior del sistema. Pero si usted puede permitirse el lujo...

Piense en una plataforma de hardware ... ¿ya?, pues Oracle funciona en la misma. Los ejemplos para Oracle de este libro han sido desarrollados, concretamente, con Personal Oracle, versiones 7.3 y 8.0, para Windows 95. Este es un servidor muy estable, quizás algo lento en establecer la conexión a la base de datos, que a veces cuesta un poco instalar adecuadamente (sobre todo por las complicaciones típicas de TCP/IP), pero una vez en funcionamiento va de maravillas. Así que con Oracle no tiene pretextos para no llevarse trabajo a casa.

Oracle permite todas las funciones avanzadas de un servidor SQL serio: segmentación, replicación, etc. Incluso puede pensarse que tiene demasiados parámetros de configuración. La parte principal del control de transacciones se implementa mediante bloqueos y registros de transacciones, aunque el nivel de aislamiento superior se logra mediante copias sólo lectura de los datos. Por supuesto, el nivel mínimo de granularidad de estos bloqueos es a nivel de registro.

[7] Debe haber una explicación (nunca me la han contado) a esta pulsión freudiana del gigante azul para que sus productos siempre sean "segundos": DB2, OS/2, PS/2 ...

¿Tipos de datos? Todos los que usted desee. ¿Restricciones **check**? No tan generales como las de InterBase, pero quedan compensadas por la mayor abundancia de funciones predefinidas. Hasta la versión 7.3, Oracle implementaba solamente la propagación en cascada de borrados para la integridad referencial, como muestra la siguiente tabla:

	Borrados	**Modificaciones**
Prohibir la operación	Sí	Sí
Propagar en cascada	Sí	No
Asignar valor por omisión	Sí	-

Las extensiones procedimentales a SQL, denominadas PL/SQL, conforman un lenguaje potente, que permite programar *paquetes* (*packages*) para la implementación de tipos de datos abstractos. Con la versión 8, incluso, se pueden definir tipos de clases, u *objetos*. Esta última extensión no es, sin embargo, lo suficientemente general como para clasificar a este sistema como una base de datos orientada a objetos, en el sentido moderno de esta denominación. Uno de los puntos fuertes de la versión 4.0 de Delphi es la posibilidad de trabajar con las extensiones de objetos de Oracle 8.

Como pudiera esperarse, el lenguaje de *triggers* es muy completo, y permite especificarlos tanto a nivel de fila como de operación. Hay montones de funciones utilizables desde SQL, y curiosas extensiones al lenguaje consulta, como la posibilidad de realizar determinados tipos de clausuras transitivas.

Otros sistemas de uso frecuente

Evidentemente, es imposible hablar con autoridad acerca de todos los formatos de bases de datos existentes en el mercado, y en las secciones anteriores me he limitado a presentar aquellos sistemas con los que he trabajado con mayor frecuencia. Sin embargo, gracias a las particularidades de mi actual ocupación, he podido ver en funcionamiento a muchos de los restantes sistemas SQL con los que Delphi nos permite trabajar directamente.

Por ejemplo, DB2, de IBM. Antes mencioné que este sistema y Oracle eran los dos sistemas que más tiempo llevaban en este negocio, y los frutos de esta experiencia se dejan notar también en DB2. Existen actualmente versiones de DB2 para una amplia gama de sistemas operativos. El autor lo ha visto funcionar sobre OS/2, Windows NT y Windows 95, teniendo una versión de evaluación sobre este último sistema. Por supuesto, estas no son las únicas plataformas sobre las que puede ejecutarse.

La arquitectura de DB2 es similar a la de Oracle, a la que se parece la de MS SQL Server, que es similar a la de Sybase SQL Server... En realidad, la concepción de estos

sistemas está basada en un proyecto experimental de IBM, denominado System-R, que fue la primera implementación de un sistema relacional. En este proyecto se desarrollaron o perfeccionaron técnicas como los identificadores de registros, los mecanismos de bloqueos actuales, registros de transacciones, índices basados en árboles balanceados, los algoritmos de optimización de consultas, etc. Así que también podrá usted esperar de DB2 la posibilidad de dividir en segmentos sus bases de datos, de poder realizar réplicas y de disponer de transacciones atómicas y coherentes. El mantenimiento de las bases de datos de DB2 puede ser todo lo simple que usted desee (sacrificando algo el rendimiento), o todo lo complicado que le parezca (a costa de su cuero cabelludo). El lenguaje de *triggers* y procedimientos almacenados es muy completo, y similar al de Oracle e InterBase, como era de esperar. La única pega que le puedo poner a DB2 es que la instalación de clientes es bastante pesada, y para poder conectar una estación de trabajo hay que realizar manualmente un proceso conocido como *catalogación*. Pero esto mismo le sucede a Oracle con su SQL Net.

Otro sistema importante es Informix, que está bastante ligado al mundo de UNIX, aunque en estos momentos existen versiones del servidor para Windows NT. Su arquitectura es similar a la de los sistemas antes mencionados.

Finalmente, quiero referirme aunque sea de pasada a otras bases de datos "no BDE" (en el siguiente capítulo veremos qué quiere decir esta clasificación). Tenemos, por ejemplo, la posibilidad de trabajar con bases de datos de AS/400. Aunque el motor de datos que viene con Delphi no permite el acceso directo a las mismas, podemos programar para estas bases de datos colocando una pasarela DB2 como interfaz. No obstante, el producto Delphi/400 sí que nos deja saltarnos las capas intermedias, logrando mayor eficiencia a expensas de la pérdida de portabilidad. También está muy difundido Btrieve, una base de datos que inicio su vida como un sistema navegacional, pero que en sus últimas versiones ha desarrollado el producto Pervasive SQL, que es un verdadero motor de datos cliente/servidor relacional. Lamentablemente, tampoco está soportado directamente por el motor de datos de Delphi (por el momento).

Capítulo

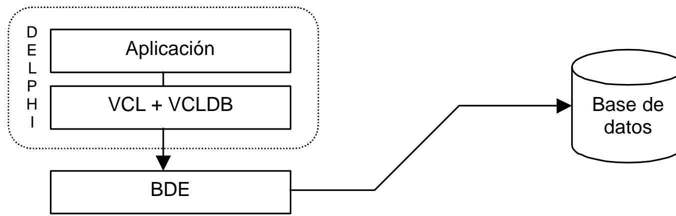
5

El Motor de Datos de Borland

DELPHI 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.

Algo de razón había, no obstante, para evitar la programación de bases de datos con estos lenguajes de propósito general. Y era la pobre integración de la mayor parte de las bibliotecas de acceso a bases de datos que existían en el mercado. Sin embargo, a partir de entonces los lenguajes tradicionales han experimentado cambios revolucionarios. En el prólogo a este libro he mencionado la introducción de la orientación a objetos, el uso de componentes para la comunicación bidireccional y el tratamiento de errores por medio de excepciones. Estos avances han mejorado notablemente la integración y facilidad de uso de bibliotecas creadas por terceros.

Delphi, en particular, utiliza una arquitectura estructurada en dos niveles. En el nivel inferior se encuentra el *Motor de Datos de Borland*, ó *Borland Database Engine*, más conocido por las siglas BDE, que es un conjunto de funciones agrupadas en bibliotecas dinámicas (DLLs). Esta biblioteca no es orientada a objetos, no permite eventos, y los errores se notifican del modo tradicional: un valor de retorno de la función que falla. Pero el segundo nivel de la arquitectura se encarga de corregir estos "fallos": el programador de Delphi no utiliza directamente las funciones del BDE, sino por mediación de objetos definidos en la VCL, que es la biblioteca de componentes de Delphi.



En este capítulo estudiaremos la estructura y filosofía del Motor de Datos, su instalación y configuración. No veremos, en cambio, la forma de programar con este Motor, pues en muy pocas ocasiones necesitaremos profundizar en este nivel de funcionalidad. Para terminar, mencionaremos alternativas al uso del BDE.

Qué es, y cómo funciona

Explicábamos en el capítulo anterior las diferencias entre las bases de datos de escritorios y los sistemas SQL. Debido al modelo de comunicación entre las aplicaciones y los datos, una de las implicaciones de estas diferencias es la forma en que se implementa la navegación sobre los datos y, en consecuencia, el estilo de programación que queda determinado. Para las bases de datos de escritorio, por ejemplo, está muy claro el concepto de *posición de registro*. Además, al abrirse una tabla, cuesta casi el mismo tiempo ir al registro 10 que al 10.000. Sin embargo, para las bases de datos SQL la posición de un registro no es una invariante: el orden de inserción es un concepto ajeno a este tipo de formato. Y, posiblemente, buscar el registro número 10.000 de una tabla tarde mil veces más que buscar el décimo registro.

Las operaciones de navegación se implementan muy naturalmente en bases de datos de escritorio. Cuando abrimos una tabla podemos trabajar con sus registros igual que si estuvieran situados en un vector o *array*; es fácil retroceder y avanzar, buscar el primero y el último. En cambio, en una base de datos SQL esta operación puede complicarse. Cuando abrimos una tabla o consulta en estos sistemas normalmente obtenemos un *cursor*, que puede imaginarse un tanto aproximadamente como un fichero de punteros a registros; la implementación verdadera de los cursores depende del sistema SQL concreto y de la petición de apertura realizada. Algunos sistemas, por ejemplo, solamente ofrecen cursores unidireccionales, en los cuales no se permite el retroceso.

En compensación, las bases de datos SQL permiten tratar como tablas físicas al resultado de consultas SQL. Podemos pedir el conjunto de clientes que han comprado los mismos productos que las empresas situadas en Kamchatka, y recibir un cursor del mismo modo que si hubiéramos solicitado directamente la tabla de clientes. Los sistemas basados en registros no han ofrecido, tradicionalmente y hasta fechas recientes, estas facilidades.

Uno de los objetivos fundamentales del diseño de BDE es el de eliminar en lo posible la diferencia entre ambos estilos de programación. Si estamos trabajando con un servidor que no soporta cursores bidireccionales, el Motor de Datos se encarga de implementarlos en el ordenador cliente. De este modo, podemos navegar por cualquier tabla o consulta SQL de forma similar a como lo haríamos sobre una tabla de dBase. Para acercar las bases de datos orientadas a registros a las orientadas a conjuntos, BDE ofrece un intérprete local para el lenguaje SQL.

El Motor de Datos ofrece una arquitectura abierta y modular, en la cual existe un núcleo de funciones implementadas en ficheros DLLs, al cual se le añaden otras bibliotecas dinámicas para acceder a los distintos formatos de bases de datos soportados. El BDE trae ya unos cuantos controladores específicos, aunque recientemente Borland ha puesto a disposición del público el llamado *Driver Development Kit* (*DDK*) para el desarrollo de nuevos controladores.

Controladores locales y SQL Links

Para trabajar con bases de datos locales, BDE implementa controladores para los siguientes formatos:

- dBase
- Paradox
- Texto ASCII
- FoxPro (a partir de Delphi 3)
- Access (a partir de Delphi 3)

Los formatos anteriores están disponibles tanto para las versiones Standard, Professional o Client/Server de Delphi. Además, se pueden abrir bases de datos de InterBase Local. El servidor local de InterBase viene con las versiones Professional y Client/Server de Delphi, pero las versiones de 32 bits de este servidor requieren el pago de un pequeño *royalty* para su distribución.

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
- Informix
- DB2 (a partir de Delphi 2)

- Sybase
- MS SQL Server

Los SQL Links pueden adquirirse por separado. Aunque la publicidad es algo confusa al respecto, sin embargo, no basta con instalar este producto sobre un Delphi Professional para convertirlo en un Delphi Client/Server. Por lo tanto, si aún no ha comprado su Delphi y tiene en mente trabajar con servidores remotos SQL, vaya directamente a la versión cliente/servidor y no cometa el error de adquirir una versión con menos prestaciones.

Acceso a fuentes de datos ODBC

ODBC (*Open Database Connectivity*) es un estándar desarrollado por Microsoft que ofrece, al igual que el BDE, una interfaz común a los más diversos sistemas de bases de datos. Desde un punto de vista técnico, los controladores ODBC tienen una interfaz de tipo SQL, por lo cual son intrínsecamente inadecuados para trabajar eficientemente con bases de datos orientadas a registros. La especificación también tiene otros defectos, como no garantizar los cursores bidireccionales si el mismo servidor no los proporciona. Sin embargo, por ser una interfaz propuesta por el fabricante del sistema operativo, la mayoría de las compañías que desarrollan productos de bases de datos se han adherido a este estándar.

BDE permite 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. Por suerte, a partir de la versión 4.0 del BDE que venía con Delphi 3, se han mejorado un poco los tiempos de acceso vía ODBC.

¿Dónde se instala el BDE?

Para distribuir una aplicación de bases de datos escrita en Delphi necesitamos distribuir también el Motor de Datos. En la primera versión de Delphi se incluían los discos de instalación de este producto en un subdirectorio del CD-ROM; solamente había que copiar estos discos y redistribuirlos. A partir de Delphi 2, no se incluye una instalación prefabricada del BDE, pues siempre podremos generarla mediante InstallShield, que ahora acompaña a Delphi.

En dependencia del modelo de aplicación que desarrollemos, la instalación del Motor de Datos tendrá sus características específicas. Los distintos modelos de aplicación son los siguientes:

- *Aplicación monopuesto*: La base de datos y la aplicación residen en el mismo ordenador. En este caso, por supuesto, también debe haber una copia del BDE en la máquina.
- *Aplicación para bases de datos locales en red punto a punto*: Las tablas, normalmente de Paradox ó dBase, residen en un servidor de ficheros. La aplicación se instala en cada punto de la red, o en un directorio común de la red, pero se ejecuta desde cada uno de los puestos. Opcionalmente, la aplicación puede ejecutarse también desde el ordenador que almacena las tablas. En este caso, cada máquina debe ejecutar una copia diferente del BDE. Lo recomendable es instalar el BDE en cada puesto de la red, para evitar problemas con la configuración del Motor de Datos.
- *Aplicaciones cliente/servidor en dos capas*[8]: Hay un servidor (UNIX, NetWare, WinNT, etcétera) que ejecuta un servidor de bases de datos SQL (InterBase, Oracle, Informix...). Las aplicaciones se ejecutan nuevamente desde cada puesto de la red y acceden al servidor SQL a través del cliente SQL instalado en el puesto. Nuevamente, el BDE debe ejecutarse desde cada estación de trabajo.
- *Aplicaciones multicapas*: En su variante más sencilla (tres capas), es idéntica a la configuración anterior, pero las estaciones de trabajo no tienen acceso directo al servidor SQL, sino por medio de un servidor de aplicaciones. Este es un ordenador con Windows NT ó Windows 95 instalado, y que ejecuta una aplicación que lee datos del servidor SQL y los ofrece, por medio de comunicación OLEnterprise, TCP/IP, DCOM o CORBA, a las aplicaciones clientes. En este caso, hace falta una sola copia del BDE, en el servidor de aplicaciones. Este modelo sólo funciona a partir de Delphi 3.

El lector se dará cuenta que estas son configuraciones simplificadas, pues una aplicación puede trabajar simultáneamente con más de una base de datos, incluso en diferentes formatos. De este modo, ciertos datos pueden almacenarse en tablas locales para mayor eficiencia, y otros ser extraídos de varios servidores SQL especializados. También, como veremos al estudiar la tecnología Midas, se pueden desarrollar aplicaciones "mixtas", que combinen accesos en una, dos, tres o más capas simultáneamente.

Una de las preguntas más frecuentes acerca del BDE es la posibilidad de instalarlo en un solo puesto de la red, con la doble finalidad de evitar la instalación en cada una de las estaciones de trabajo y las necesarias actualizaciones cada vez que cambie la versión del motor, o alguno de los parámetros de instalación. Bien, esto es técnicamente posible, pero no lo recomiendo. El problema consiste en que es necesario dejar una instalación mínima en los clientes, de todos modos, además que de esta forma se aumenta el tráfico en la red.

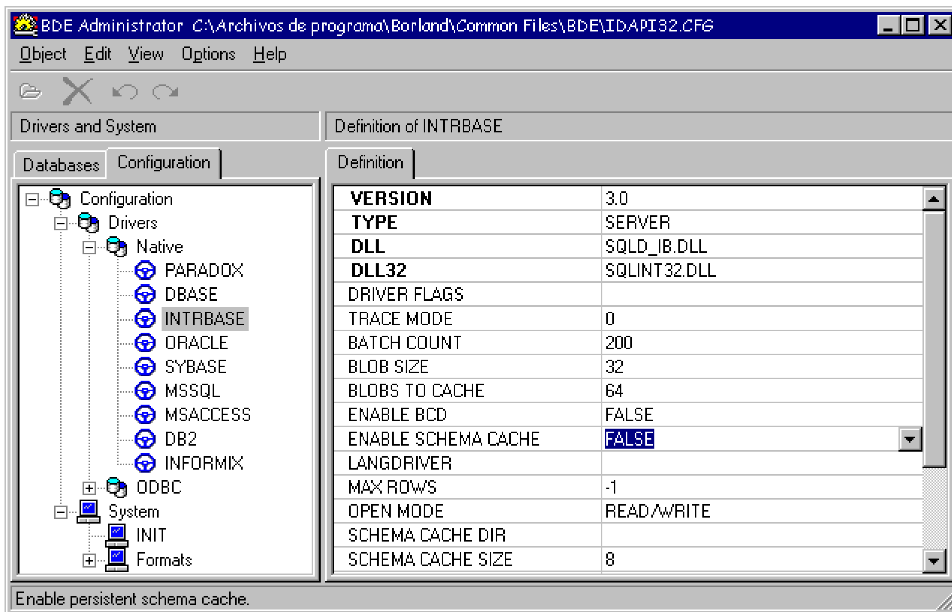
---

[8] "Capa" es mi traducción de la palabra inglesa de moda *tier* (no confundir con *tire*). Así, traduciré *multi-tier* como *multicapas*.

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.

Existen diferencias entre las implementaciones de este programa para Delphi 1 y 2, por una parte, y Delphi 3 y 4, por la otra. Pero se trata fundamentalmente de diferencias visuales, de la interfaz de usuario, pues los parámetros a configurar y la forma en que están organizados son los mismos. Para simplificar, describiré la interfaz visual del Administrador que viene con Delphi 4. La siguiente figura muestra el aspecto de este programa:



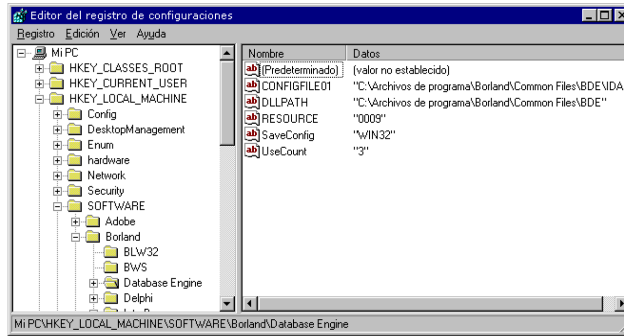
La ventana principal del Administrador está dividida en dos páginas, *Databases* y *Configuration*; comenzaremos por esta última, que permite ajustar los parámetros de los controladores instalados y los parámetros globales de la biblioteca.

Configuración del registro e información de versión

¿Dónde se almacena la información de configuración del BDE? Resulta que se almacena en dos lugares diferentes: la parte principal se guarda en el registro de Windows, mientras que la información de los alias y un par de parámetros especiales va en el

fichero *idapi32.cfg*, cuya ubicación se define en el propio registro. La clave a partir de la cual se encuentran los datos 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.

Para saber la versión del BDE que estamos utilizando, se utiliza el comando de menú *Object|Version information*, que aparece en el Administrador cuando nos encontramos en la página *Databases*:

BDE Version Information

DLL Name	Version Number	Date	Time	Size (bytes)
IDQBE32.DLL	5.0.0.27 (0)	30/04/98	5:00:00	422400
IDSQL32.DLL	5.0.0.27 (0)	30/04/98	5:00:00	461824
SQLORA32.DLL	5.0.0.27 (0)	30/04/98	5:00:00	414208
SQLORA8.DLL	5.0.0.27 (1)	30/04/98	5:00:00	440832
SQLINT32.DLL	5.0.0.27 (0)	30/04/98	5:00:00	378880
SQLINF32.DLL	5.0.0.27 (0)	30/04/98	5:00:00	394240
SQLDB232.DLL	5.0.0.27 (0)	30/04/98	5:00:00	424960
SQLMSS32.DLL	5.0.0.27 (0)	30/04/98	5:00:00	409088
SQLSYB32.DLL	5.0.0.27 (0)	30/04/98	5:00:00	408576
SQLSSC32.DLL	5.0.0.27 (0)	30/04/98	5:00:00	408064
IDPROV32.DLL	5.0.0.27 (0)	30/04/98	5:00:00	110080
BLW32.DLL	3.0.0.1 (0)	30/04/98	5:00:00	65536
DBCLIENT.DLL	5.0.0.27 (2)	30/04/98	5:00:00	214104

OK Help

La siguiente tabla resume la historia de las últimas versiones del BDE:

BDE 4/4.01	Versión original distribuida con Delphi 3/3.01, que incluye por primera vez el acceso directo a Access 95 y a FoxPro
BDE 4.5/4.51	Acompaña a Visual dBase 7, e introduce el nuevo formato DBF7 y el acceso directo a Access 97
BDE 5	Acompaña a Delphi 4, e introduce el soporte para Oracle 8

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. Delphi facilita esta tarea mediante componentes de alto nivel, en concreto mediante la clase *TDatabase*.

Los alias ofrecen a la aplicación que los 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.

A partir de la versión 4.0 del BDE, se introducen los alias *virtuales*, que corresponden a los nombres de fuentes de datos de ODBC, conocidos como DSN (*data source names*). Una aplicación puede utilizar entonces 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 subnodos, *INIT* y *Formats*, para los parámetros de funcionamiento y los formatos de visualización por omisión. Aunque estos últimos pueden cambiarse para controlar el formato visual de fechas, horas y valores numéricos, es preferible especificar los formatos desde nuestras aplicaciones Delphi, para evitar dependencias con respecto a instalaciones y actualizaciones del BDE. La forma de hacerlo será estudiada en el capítulo 15.

La mayor parte de los parámetros de sistema tienen que ver con el uso de memoria y otros recursos del ordenador. Estos parámetros son:

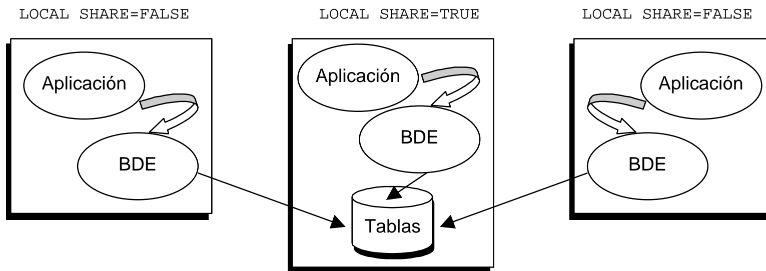
Parámetro	**Explicación**
MAXFILEHANDLES	Máximo de *handles* de ficheros admitidos.
MINBUFSIZE	Tamaño mínimo del *buffer* de datos, en KB.
MAXBUFSIZE	Tamaño máximo del *buffer* de datos, en KB (múltiplo de 128).
MEMSIZE	Tamaño máximo, en MB, de la memoria consumida por BDE.
LOW MEMORY USAGE LIMIT	Memoria por debajo del primer MB consumida por BDE.
SHAREDMEMSIZE	Tamaño máximo de la memoria para recursos compartidos.
SHAREDMEMLOCATION	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.

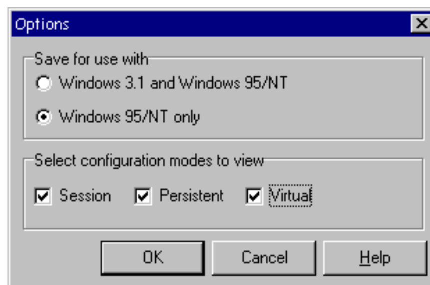
Cuando se va a trabajar con dBase y Paradox es importante configurar correctamente el parámetro *LOCAL SHARE*. Para acelerar las operaciones de bloqueo, BDE normalmente utiliza una estructura en memoria para mantener la lista de bloqueos impuestos en las tablas pertenecientes a los discos locales; esta 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 *pdoxusrs.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, 2048 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 menos problemática. 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.

A partir de la versión 4 del BDE (Delphi 3) se ha incluido un controlador para Access. Este controlador actúa como interfaz con el Microsoft Jet Engine, que tiene que estar presente durante el diseño y ejecución de la aplicación. La versión del BDE que acompañaba a Delphi 3, solamente permitía utilizar el motor que venía con

Office 95: el DAO 3.0. Si se había instalado Access 97 sobre una instalación previa del 95 no habría problemas, pues la versión anterior se conservaba. No obstante, desde la aparición de la versión 4.5 del BDE, que acompañaba a Visual dBase 7, se ha incluido también el acceso mediante el motor DAO 3.5, que es el que viene con Office 97.

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
```

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**
VERSION	Número interno de versión del controlador
TYPE	Siempre debe ser *SERVER*
DLL	DLL de acceso para 16 bits
DLL32	DLL de acceso para 32 bits

En realidad, *DLL* y *DLL32* se suministran para el caso especial de Informix y Sybase, que admiten dos interfaces diferentes con el servidor. Solamente en estos casos deben modificarse estos parámetros.

La siguiente lista muestra los parámetros relacionados con la apertura de bases de datos:

Parámetro	**Significado**
SERVER NAME	Nombre del servidor
DATABASE NAME	Nombre de la base de datos
USER NAME	Nombre del usuario inicial
OPEN MODE	Modo de apertura: *READ/WRITE* ó *READ ONLY*
DRIVER FLAGS	Modifíquelo sólo según instrucciones de Borland
TRACE MODE	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, 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. Afortunadamente, 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
SCHEMA CACHE DIR	El directorio donde se copian los esquemas
SCHEMA CACHE SIZE	Cantidad de tablas cuyos esquemas se almacenan
SCHEMA CACHE TIME	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 Delphi "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
BLOB SIZE	Tamaño máximo de un BLOB a recibir
BLOBS TO CACHE	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
BATCH COUNT	Registros que se transfieren de una vez con *BatchMove*
ENABLE BCD	Activa el uso de *TBCDField* por Delphi
MAX ROWS	Número máximo de filas por consulta
SQLPASSTHRU MODE	Interacción entre SQL explícito e implícito
SQLQRYMODE	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 para Delphi con mayor facilidad. En primer lugar, la instalación del software propio de InterBase en el cliente es elemental: basta con instalar el propio BDE que necesitamos para las aplicaciones Delphi. Sólo necesitamos un pequeño cambio si queremos utilizar TCP/IP como protocolo de comunicación con el servidor: hay que añadir una entrada en el ficheros *services*, situado en el directorio de Windows, para asociar un número de puerto al nombre del servicio de InterBase:

```
gds_db          3050/tcp
```

Por supuesto, también podemos tomarnos el trabajo de instalar el software cliente y las utilidades de administración en cada una de las estaciones de trabajo, pero si usted tiene que realizar una instalación para 50 ó 100 puestos no creo que le resulte una opción muy atractiva.

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 utilizar 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 este 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.

Para terminar, podemos 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. También se puede probar con el valor 512, que induce el nivel de aislamiento superior para las transacciones implícitas, el nivel de lecturas repetibles; en el capítulo 31, donde nos ocupamos de las transacciones, veremos en qué consiste esto del nivel de aislamiento. Si quiere combinar ambas constantes, 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.

Configuración de MS SQL Server

También es muy sencillo configurar un ordenador para que pueda acceder a un servidor de MS SQL Server. Aunque existe un software para instalar en cada cliente, y que contiene herramientas de administración y consulta, basta con colocar un par de DLLs en el directorio de sistema de Windows. Por ejemplo, si queremos utilizar *named pipes* para la comunicación, necesitamos estas dos DLLs, que podemos extraer del servidor:

ntwdblib.dll	La biblioteca DB-Library de programación.
dbnmpntw.dll	Necesaria para *named pipes*.

En MS SQL Server, a diferencia de InterBase, *SERVER NAME* representa el nombre del servidor del cual vamos a situar las bases de datos. Normalmente, este nombre coincide con el nombre del ordenador dentro del dominio, pero puede también ser diferente. Por ejemplo, si el nombre del ordenador contiene acentos o caracteres no válidos para un identificador SQL, la propia instalación del servidor cambia el

nombre que identificará al servidor SQL para evitar conflictos. Si vamos a utilizar un único servidor de MS SQL Server, es posible configurar *SERVER NAME* con el nombre del mismo, de modo tal que quede como valor por omisión para cualquier acceso a estas bases de datos. 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*.

Sin embargo, en contraste con InterBase, el controlador de MS SQL Server tiene muchos parámetros que pueden ser configurados. La siguiente tabla ofrece un resumen de los mismos:

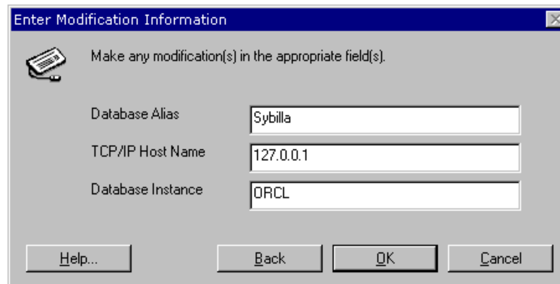
Parámetro	Significado
CONNECT TIMEOUT	Tiempo máximo de espera para una conexión, en segundos
TIMEOUT	Tiempo máximo de espera para un bloqueo
MAX QUERY TIME	Tiempo máximo de espera para la ejecución de una consulta
BLOB EDIT LOGGING	Desactiva las modificaciones transaccionales en campos BLOB
APPLICATION NAME	Identificación de la aplicación en el servidor
HOST NAME	Identificación del cliente en el servidor
DATE MODE	Formato de fecha: 0=*mdy*, 1=*dmy*, 2=*ymd*
TDS PACKET SIZE	Tamaño de los paquetes de intercambio
MAX DBPROCESSES	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.

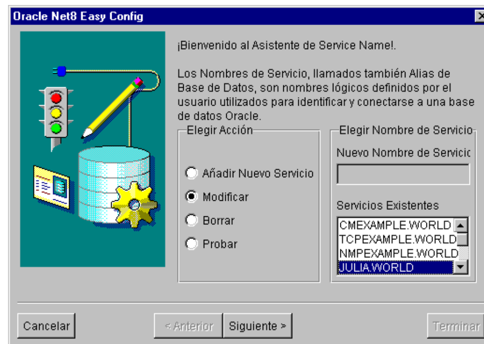
Por su parte, el parámetro *TDS PACKET SIZE* está relacionado con la opción *network packet size* del procedimiento *sp_configure* de la configuración del servidor. Se puede intentar aumentar el valor del parámetro para una mayor velocidad de transmisión de datos, por ejemplo, a 8192. TDS quiere decir *Tabular Data Stream*, y es el formato de transmisión utilizado por MS SQL Server.

Configuración de Oracle

Para configurar un cliente de Oracle, necesitamos instalar obligatoriamente el software SQL Net que viene con este producto, y configurarlo. 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 para Windows 95. 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. A pesar de las diferencias en formato, el procedimiento de conexión sigue siendo básicamente el mismo:



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 hemos creado con SQL Net.

Parámetro	Significado
VENDOR INIT	Nombre de DLL suministrada por Oracle
NET PROTOCOL	Protocolo de red; casi siempre TNS
ROWSET SIZE	Número de registros que trae cada petición

Parámetro	Significado
ENABLE INTEGERS	Traduce columnas de tipo *NUMERIC* sin escala a campos enteros de Delphi
LIST SYNONYMS	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 para configurar en el controlador de Oracle. El primero 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 para sus diferentes versiones. El segundo nuevo parámetro es *OBJECT MODE*, que permite activar las extensiones de objetos de Oracle para que puedan ser utilizadas desde Delphi.

Configuración de otros sistemas

Mencionaré a continuación algunos de los parámetros de configuración de los restantes formatos. Por ejemplo, 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. Este es el único parámetro especial del controlador.

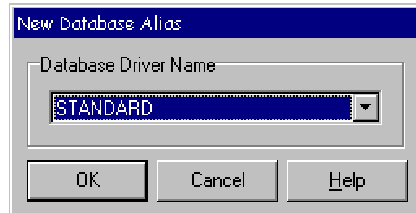
Informix tiene un mecanismo similar a MS SQL Server: es necesario indicar el nombre del servidor, *SERVER NAME*, y el de la base de datos, *DATABASE NAME*. El parámetro *LOCK MODE* indica el número de segundos que un proceso espera por la liberación de un bloqueo; por omisión, se espera 5 segundos. El formato de fechas se especifica mediante los parámetros *DATE MODE* y *DATE SEPARATOR*.

Por último, los parámetros de conexión a Sybase son lo suficientemente parecidos a los de MS SQL Server como para no necesitar una discusión adicional.

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.

Alternativas al Motor de Datos

De cualquier manera, como he explicado al principio del capítulo, BDE no es la única forma de trabajar con bases de datos en Delphi, aunque sea la más recomendable. Si no tenemos un SQL Link para nuestra base de datos, y no queremos utilizar una conexión ODBC, contamos con estas opciones:

- Utilizar un sustituto del BDE
- Utilizar componentes derivados directamente de *TDataset*

La primera opción va quedando obsoleta, pero era la única posibilidad en Delphi 1 y 2. Consistía en reemplazar tanto el motor de datos como las unidades de Delphi que lo utilizaban. Demasiado radical.

La segunda posibilidad aparece con Delphi 3, y consiste en desarrollar componentes derivados de la clase *TDataset*, que es la base de la jerarquía de los objetos de acceso a datos. De esta forma, podemos seguir utilizando el BDE para otros formatos, o no usarlo en absoluto, sin tener que cargar con el código asociado. En realidad existe una tercera alternativa: desarrollar un SQL Link con el DDK (*Driver Development Kit*) que ofrece Borland. Pero todavía no conozco ningún producto en este estilo.

Existen, a nivel comercial, alternativas al BDE más o menos exitosas. Por ejemplo, para trabajar con bases de datos de Btrieve, Regatta Systems ofrece Titan, que se comunica directamente con el motor de datos de Btrieve. En realidad, Titan es una *suite* de productos de bases de datos que ofrece también acceso directo a bases de datos de Access y SQL Anywhere. En la misma línea se sitúa Apollo, de Success-Ware, ofreciendo acceso a bases de datos de Clipper, FoxPro y un formato nativo de la compañía. Recuerde que Delphi 2 no permitía trabajar con FoxPro ó Access.

Algunas de estas alternativas las comercializa el propio Borland, como es el caso de Delphi/400. Este es un conjunto de componentes que realizan acceso directo a bases de datos en AS/400 mediante SNA. Delphi/400 está disponible a partir de Delphi 3, pues el producto original pertenecía a una empresa francesa que lo desarrolló para Delphi 2.

2

Fundamentos del lenguaje

- Encapsulamiento
- Herencia y polimorfismo
- Elementos de programación con Windows
- Propiedades
- Eventos
- Excepciones
- Tipos de datos de Delphi
- Técnicas de gestión de ventanas
- Los módulos de datos y el Depósito de Objetos

Parte

Capítulo

6

Programación orientada a objetos: encapsulamiento

SI SE LE PREGUNTA A ALGUIEN QUE PASEA POR LA CALLE qué es la Teoría de la Relatividad es muy posible que nos conteste: "Sí, cómo no, es muy sencillo, todo es relativo…". Algo parecido sucede con la Programación Orientada a Objetos. Todos, más o menos, saben que *tiene que ver* con objetos. Otros pocos recitan, como un mantra incomprensible, la palabra "reusabilidad". A veces, cuando imparto cursos de programación, hago a los asistentes una pregunta bastante inocente: "¿Cómo convencerías a tu jefe para que cambie su lenguaje de programación tradicional por uno orientado a objetos … comprando, por supuesto, compilador, manuales y libros?" Y no crea, las respuestas suelen ser divertidas.

El problema de la división en módulos

Todo comienza a partir de una observación sencilla. Tenemos frente a nosotros un listado de un programa de 5.000 líneas, y otro de 10.000 líneas. ¿Se puede decir que el segundo listado es probablemente el doble de "complicado" que el primero? ¡No, posiblemente es cuatro veces más "complicado"!

Precisemos ahora esta afirmación. En primer lugar, he puesto entre comillas la palabra "complicado"; la razón es que es difícil dar una definición exacta de lo que constituye la complejidad de un programa. En segundo lugar, he asumido que este hipotético programa ha sido desarrollado en un solo bloque, como en los primeros tiempos de la Informática. Esta es una suposición extrema y poco realista, pero el ejemplo nos servirá, como veremos. Mi hipótesis se puede escribir de forma matemática. Si un programa desarrollado en este estilo obsoleto tiene n líneas de código, la "complejidad subjetiva" del mismo debe ser $\boldsymbol{k}n^2$, donde $\boldsymbol{k}$ es una constante arbitraria. Si el número de líneas crece al doble, la complejidad crece cuatro veces; si el número de líneas es diez veces mayor, nos costará cien veces más esfuerzo comprender el programa.

¿Y qué pasa si dividimos el programa en dos partes aproximadamente iguales? ¿Y si estas partes son relativamente independientes entre sí? Entonces la complejidad total será la suma de las complejidades individuales de cada bloque:

$$\text{Complejidad total} = \mathbf{k}(n/2)^2 + \mathbf{k}(n/2)^2 = \tfrac{1}{2}\mathbf{k}n^2$$

¡Resulta que la complejidad total se ha reducido a la mitad después de la división! Pero no podemos perder de vista que esta reducción ha sido posible porque hemos dividido el programa en dos bloques relativamente independientes. Este es el problema de la división en módulos: ¿cómo fraccionar un programa en trozos cuya comunicación mutua sea la menor posible?

Los módulos en la programación estructurada

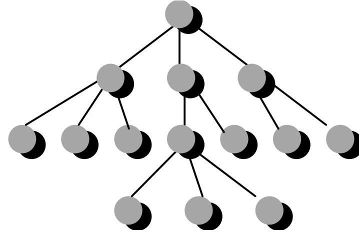
En la Programación Tradicional, nos enseñan que la forma de descomponer una aplicación en módulos debe basarse en los algoritmos que ejecuta. Cada algoritmo se codifica mediante un procedimiento o función. Si nuestro programa controla un cohete que viaja a la Luna, el proceso se divide en tres subprocesos: el despegue, el vuelo libre y el presunto aterrizaje. A su vez, el despegue se subdivide en despedirse de los familiares, sacar al gato de la cabina, arrancar los motores y rezar ...y así sucesivamente, hasta que al descomponer un proceso nos encontremos con operaciones tan simples como asignar algo a una variable o ejecutar una rutina del sistema operativo.

De acuerdo a esta metodología, cada algoritmo se representa mediante una función o procedimiento, y el concepto de módulo corresponde a estos. Una rutina se comunica en dos sentidos: con la rutina que la llama y con las rutinas a las que llama, y el canal de comunicación en ambos casos son los parámetros de la llamada. Este diseño satisface nuestro requerimiento de tener canales de comunicación "estrechos" entre los módulos, para garantizar su independencia, y que la reducción de complejidad sea efectiva.

¿Cómo se puede representar la relación entre módulos concebidos bajo esta filosofía?[9] Un programador ingenuo puede pensar enseguida en un diagrama de árbol, en el cual el nodo superior representa el código inicial de la aplicación (la función *main* de C/C++, o el programa principal de Pascal), del cual brotan ramas para las funciones que son llamadas desde el mismo. El árbol es *estrecho*, en el sentido de que de cada nodo se derivan pocas ramas; un principio básico de estilo de programación dice que el texto de una rutina debe caber en una pantalla. Por el contrario, estos árboles tienden a ser *profundos*: no hay límite en la longitud de cada rama del grafo.

---

[9] *There are more things in heaven and earth, Horatio, than are dreamt of in your philosophy.*



Sin embargo, aunque los diagramas de árbol son realmente útiles, no expresan toda la complejidad de las relaciones que se establecen entre las rutinas. La función *DescorcharChampagne* se ejecuta tanto después de un despegue exitoso, como cuando al astronauta, ya en pleno vuelo, le anuncian una subida salarial. Si intentamos representar las llamadas en tiempo de ejecución, necesitaremos un par de nodos para la función anterior. El esquema verdadero que muestra todas las relaciones entre rutinas es mucho más complicado.

Reusabilidad y el principio "abierto/cerrado"

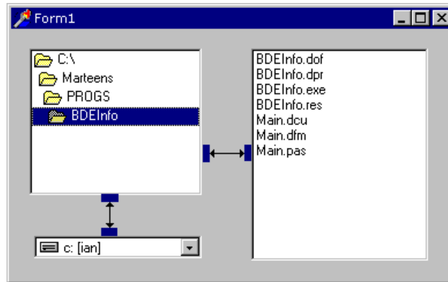
Una vez que hemos logrado reducir la complejidad dentro de una aplicación mediante una adecuada descomposición en módulos, es lógico plantearse la posibilidad de "transplantar" estos módulos a la siguiente aplicación que nos toque desarrollar. Es algo insensato comenzar una aplicación desde cero, por lo cual, desde que existe la Informática, se han desarrollado técnicas para ahorrarle al programador el trabajo de reinventar la rueda una y otra vez.

La idea es que los módulos obtenidos en la descomposición explicada anteriormente, puedan utilizarse del mismo modo que un componente electrónico. Una "pastilla", resistencia o transistor cumple con una tarea bien determinada dentro de un circuito, y cuenta con una serie de patillas que sirven para conectarla con el resto de los componentes; lo que sucede dentro de la pastilla no es asunto nuestro.

Si los módulos que hemos logrado son funciones, las patillas de conexión son sus canales de comunicación: los parámetros. En este punto se complica la analogía electrónica, pues el uso de parámetros como medio de conexión entre módulos obliga a la propagación de los valores de los mismos. Ahora el lector puede sorprenderse, pues si no conoce en absoluto la Programación Orientada a Objetos, le parecerá natural e inevitable esto de la propagación de valores de parámetros. Le pido, por favor, que espere un poco para que vea cómo los objetos ofrecen una mejor alternativa al asunto.

La primera técnica para permitir la reutilización de componentes de software fue agrupar las rutinas en *bibliotecas*. Para crear las bibliotecas, cada lenguaje desarrolló sus propios mecanismos sintácticos que permitían la compilación por separado de los

módulos. En particular, Pascal adoptó, después de una larga y tortuosa evolución, las *unidades* (*units*) como técnica de desarrollo de módulos. Posteriormente, se inventó la metodología de los *tipos de datos abstractos*, que consistía básicamente en agrupar las definiciones de tipos de datos junto a las operaciones que trabajaban con estos tipos, y en definir axiomas para formalizar la semántica de estas operaciones. La investigación sobre tipos de datos abstractos impulsó el desarrollo de técnicas de *encapsulamiento*, que más tarde probaron su utilidad en la programación orientada a objetos.



Un concepto importante en el arte de la descomposición en módulos es el denominado *principio abierto/cerrado* (*the open/closed principle*). De acuerdo a este principio, los componentes de software deben estar *cerrados*: actúan como cajas negras, y la única forma de comunicación con ellos debe ser mediante los canales establecidos, que a su vez deben tener la menor anchura de banda posible. Pero, por otra parte, deben estar *abiertos* a extensiones y especializaciones, de modo que con ligeras adaptaciones pueda utilizarse en aplicaciones imprevistas. En este sentido, las técnicas de descomposición funcional fallan. La única forma de modificar el comportamiento de una función es a través de sus parámetros; las variaciones de comportamiento deben estar estrictamente previstas. Es cierto que pueden utilizarse punteros a rutinas para lograr un poco más de flexibilidad, pero esto complica aún más las cosas: aumenta el número de parámetros necesario, ensanchándose el canal de comunicaciones con el componente.

Llegan los objetos

Pero el verdadero fallo de la Descomposición Funcional está dado por el propio criterio de descomposición. ¿Qué es lo que define a una aplicación? Su funcionalidad, por supuesto: las operaciones que realiza el programa. La próxima aplicación que programemos tendrá una funcionalidad completamente diferente de la actual. Las funciones de más bajo nivel, las que calculan logaritmos, muestran un mensaje en el monitor y lanzan un bit por el puerto serie, sí podrán volverse a utilizar. Pero todas las rutinas específicas del proyecto caducarán en poco tiempo. Es difícil que en el programa que controle el descenso de un batiscafo a la Fosa de las Marianas tengamos que sacar el gato de la cabina del cohete antes de despegar ... espere ... ¿he men-

cionado un gato? Resulta que los “objetos” o “actores” de una aplicación tienen una altísima probabilidad de sobrevivir al proyecto: tenemos también que sacar un gato del batiscafo antes de sumergirnos. ¿Por qué no probar entonces a utilizar los “objetos” como componentes de software básicos para la construcción de programas?

El primer paso de la Programación Orientada a Objetos no es nada espectacular: comenzaremos por definir qué es un objeto. Y, por el momento, un objeto será para nosotros un conglomerado de datos. Por ejemplo, un gato debe tener un nombre (digamos, *'Sylvester'*), un color de pelo (*clBlack*), una cola de cierta longitud (20 cm.), y un alimento preferido (por supuesto, *'Tweety'*). Cada lenguaje de programación tradicional tiene tipos de datos para almacenar datos de distintos tipos. C tiene el tipo **struct**, Pascal tiene **record**.

Si utilizamos el tipo **record** de Pascal, los gatos pueden representarse mediante variables del siguiente tipo:

```
type
   TGato = record
      Nombre: string;
      Color: TColor;   // TColor está definido por Delphi
      Cola: Integer;
      Alimento: string;
   end;
```

TGato es ahora un tipo de datos de Pascal, al igual que los predefinidos: *Integer*, **string**, *Boolean*, etc. La letra *T* inicial es sólo un convenio de Borland para los nombres de tipos de datos. Se pueden declarar variables pertenecientes a este tipo, y utilizarlas también en parámetros:

```
var
   Sylvester, Tom: TGato;

function LlevarGatoAlAgua(Gato: TGato): Boolean;
```

El acceso a las variables almacenadas en el **record** se realiza del siguiente modo:

```
function CrearGato(const Nombre: string): TGato;
begin
   // Dentro de la función, Result es de tipo TGato
   Result.Nombre := Nombre;
   // No hay confusión: el nombre del gato tiene
   // una variable de tipo TGato delante
   Result.Color := clBlack;
   Result.Cola := 18;
   // Sylvester la tiene más larga
   Result.Alimento := 'Whiskas';
end;
```

Sin embargo, las posibilidades de un **record** no van más allá de lo mostrado, mientras que la saga de los objetos acaba de comenzar. Por esto, cambiaremos la definición

anterior, utilizando la nueva palabra reservada **class**. Por ahora, puede considerar a las clases y a los registros como tipos similares; más adelante veremos que hay diferencias significativas, incluso en el formato de representación.

```
type
   TGato = class
      Nombre: string;
      Color: TColor;  // TColor está definido por Delphi
      Cola: Integer;
      Alimento: string;
   end;
```

Necesitamos un poco de terminología. Una *clase* (*class*) es un tipo de datos definido del modo anterior. Un *objeto*, o *instancia* (*object*, *instance*) es, hasta que puntualicemos los temas de representación más adelante, una variable perteneciente a un tipo de clase. *Nombre*, *Color*, *Cola* y *Alimento* serán denominados *campos* o *atributos* (*fields*, *attributes*), aunque utilizaremos *campo* lo menos posible, para evitar confusiones con los campos de las tablas de bases de datos.

El próximo paso de la Programación Tradicional sería empezar a definir funciones y procedimientos que trabajen con este tipo de dato. También es el siguiente paso de la Programación Orientada a Objetos, pero la forma en que se da marca la diferencia entre estilos.

Operaciones sobre un tipo de datos: funciones

Una vez que tenemos definido el tipo de datos, tenemos que crear procedimientos y funciones para realizar operaciones sobre las variables del tipo. Sin estas operaciones, el tipo de datos es solamente un recipiente inanimado de datos, un muñeco de arcilla esperando a que algún dios insufle aliento de vida en sus narices. En un lenguaje de programación no orientado a objetos, declararíamos las funciones y procedimientos que actúan sobre un gato en forma parecida a la siguiente:

```
procedure CargarGatoDesdeFichero(var Gato: TGato;
   const Fichero: string);
procedure GuardarGatoEnFichero(const Gato: TGato;
   const Fichero: string);
procedure CopiarGatoAlPortapapeles(const Gato: TGato);
function LlevarGatoAlAgua(const Gato: TGato): Boolean;
```

Normalmente, Delphi pasa los parámetros procedimientos y funciones por *valor*. Esto quiere decir que la rutina cuenta con una copia local del valor o variable utilizado en la llamada, y que se pueden realizar modificaciones en el parámetro sin que se modifique la variable original. Existen otras dos formas de traspaso de parámetros. *CargarGatoDesdeFichero* pasa el parámetro del gato por *referencia*, lo que se indica con el prefijo **var** antes del parámetro; la acción de **var** llega hasta el punto y coma. Delphi

utiliza el traspaso por referencia para lograr parámetros de entrada y salida; en este modo de traspaso se suministra al procedimiento la dirección de la variable. Los restantes procedimientos y funciones pasan al gato como *valor constante*: físicamente se pasa una dirección, pero el compilador no permite modificaciones en el parámetro. Delphi permite también declarar parámetros exclusivamente de salida, con el prefijo **out**, que son utilizados en el modelo COM, pero no utilizaremos este tipo de parámetros por ahora.

Los procedimientos definidos se utilizan como se muestra a continuación:

```
var
   Gato: TGato;       // TGato es aquí un record
begin
   CargarGatoDesdeFichero(Gato, 'C:\GARFIELD.CAT');
   Gato.Alimento := 'Caviar';
   CopiarGatoAlPortapapeles(Gato);
   GuardarGatoEnFichero(Gato, 'C:\GARFIELD.CAT');
end;
```

Para completar el ejemplo, veamos cómo puede guardarse el alma inmortal de un felino en disco y recuperarse posteriormente, utilizando las rutinas de ficheros de Pascal:

```
procedure GuardarGatoEnFichero(const Gato: TGato;
   const Fichero: string);
var
   Txt: TextFile;
begin
   AssignFile(Txt, Fichero);
   Rewrite(Txt);
   WriteLn(Txt, Gato.Nombre);
   WriteLn(Txt, Gato.Color);
   WriteLn(Txt, Gato.Cola);
   WriteLn(Txt, Gato.Alimento);
   CloseFile(Txt);
end;

procedure CargarGatoDesdeFichero(var Gato: TGato;
   const Fichero: string);
var
   Txt: TextFile;
begin
   AssignFile(Txt, Fichero);
   Reset(Txt);
   ReadLn(Txt, Gato.Nombre);
   ReadLn(Txt, Gato.Color);
   ReadLn(Txt, Gato.Cola);
   ReadLn(Txt, Gato.Alimento);
   CloseFile(Txt);
end;
```

He utilizado ficheros de texto, sin preocuparme por los posibles errores; hasta que no sepamos utilizar excepciones, no vale la pena ocuparse de este asunto. Los procedimientos de ficheros utilizados son los siguientes:

Procedimiento	Objetivo
AssignFile	Asigna un nombre a una variable de fichero.
Reset	Abre un fichero en modo de lectura.
Rewrite	Crea un fichero para escritura.
ReadLn	Lee una línea desde un fichero de texto.
WriteLn	Escribe una línea en un fichero de texto.
CloseFile	Cierra un fichero abierto.

Operaciones sobre un tipo de datos: métodos

La POO nos hace definir las funciones y procedimientos *dentro* de la declaración de la clase, después de definir los campos o atributos. Un tipo **record** no permite declarar procedimientos dentro de su definición; por este motivo utilizamos el tipo **class** para declarar los gatos. A las rutinas definidas dentro de una clase se le denominan *métodos*:

```
type
   TGato = class
      Nombre: string;
      Color: TColor;
      Cola: Integer;
      Alimento: string;
      procedure CargarDesdeFichero(const Fichero: string);
      procedure GuardarEnFichero(const Fichero: string);
      procedure CopiarAlPortapapeles;
      function LlevarAlAgua: Boolean;
   end;
```

Las rutinas definidas para el gato han sufrido un par de transformaciones al convertirse en métodos. La primera es quizás la menos importante, pero es curiosa: la palabra *Gato* ha desaparecido de los nombres; más adelante explicaré el motivo. La segunda sí es fundamental: se ha esfumado el parámetro de tipo *TGato*. ¿Cómo hacemos ahora para indicar qué gato tenemos que llevar al agua? El siguiente código muestra cómo ejecutar los nuevos métodos:

```
// Hay una "pequeña" mentira en este código
var
   Gato: TGato;        // TGato es una clase
```

```
begin
   // … aquí falta algo …
   Gato.CargarDesdeFichero('C:\GARFIELD.CAT');
   Gato.Alimento := 'Caviar';
   Gato.CopiarAlPortapapeles;
   Gato.GuardarEnFichero('C:\GARFIELD.CAT');
end;
```

Ahora podemos ver que se trata "simplemente" de un truco sintáctico. En la declaración del método se ha omitido el gato pues el parámetro se asume, al estar la declaración dentro de la definición de la clase. Después, al ejecutar el método, el objeto se indica *antes* que el nombre del método, separándolo por un punto, de forma similar a la que utilizamos para acceder a los atributos. Si comparamos, línea a línea, las llamadas en el viejo y en el nuevo estilo, comprenderemos más fácilmente las equivalencias:

```
Gato.CargarDesdeFichero('TOM.CAT');  CargarGatoDesdeFichero(Gato,'TOM.CAT');
Gato.GuardarEnFichero('TOM.CAT');    GuardarGatoEnFichero(Gato, 'TOM.CAT');
Gato.CopiarAlPortapapeles;           CopiarGatoAlPortapapeles(Gato);
```

Cuando se ejecuta un método de un objeto, se dice también que el método se *aplica* sobre el objeto.

La primera ventaja: nombres más cortos

¿Por qué los nombres de las operaciones son más cortos ahora? En realidad tendríamos que preguntarnos por qué eran más largos antes. Y la culpa la tienen los ratones, ¿quién si no? Si la aplicación escrita en el estilo tradicional trabaja con gatos, es lógico que tenga en cuenta a los ratones, y que tengamos definido un tipo *TRaton*:

```
type
   TRaton = record
      // …
   end;
```

Por supuesto, los ratones también harán uso del disco duro, y existirán procedimientos *CargarRatonDesdeFichero* y *GuardarRatonEnFichero*. Si utilizamos los nombres más cortos y cómodos, tendremos conflictos con los procedimientos correspondientes de los gatos. Apartándonos del mundo de los felinos y roedores, esta práctica es frecuente en las bibliotecas de rutinas. Ejecute, por ejemplo, la ayuda en línea de la interfaz del Motor de Datos de Borland, que viene con Delphi, y se encontrará con funciones de nombres tan pintorescos como *DbiOpenTable*, *DbiAddAlias*, *DbiCloseSession*, etc.

En cambio, si *TRaton* y *TGato* son clases, ambas pueden declarar métodos llamados *CargarDesdeFichero*. No se produce conflicto al utilizar estos métodos, pues solamente

pueden ejecutarse aplicados sobre un objeto, y en este caso el compilador tiene suficiente información para saber qué método es el necesario:

```
var
   Jinks: TGato;
   Pixie, Dixie: TRaton;
begin
   Pixie.CargarDesdeFichero('ANIMAL.1');     // Cargar ratón
   Jinks.CargarDesdeFichero('ANIMAL.2');     // Cargar gato
   Dixie.CargarDesdeFichero('ANIMAL.3');     // Cargar ratón
end;
```

Esta característica de la programación orientada a objetos se conoce como *sobrecarga de métodos*. En el siguiente capítulo veremos como Delphi4, siguiendo a C++, ha introducido la directiva **overload** para permitir que un nombre sea compartido incluso por dos métodos de una misma clase, o por funciones y procedimientos en general.

La implementación de un método

Hemos visto, hasta el momento, cómo se declara un método y cómo se ejecuta. Pero no he dicho cómo implementar el cuerpo de estos métodos. Generalmente, las declaraciones de tipos de clases se realizan en la interfaz de una *unit* de Delphi, aunque también se pueden definir clases internas en la sección **implementation** de una unidad. En cualquier caso, el cuerpo del método se desarrolla más adelante, en la implementación de la unidad. Lo primero es indicar qué método estamos programando:

```
procedure TGato.GuardarEnFichero(const Fichero: string);
begin
   // …
end;
```

Cuando se vuelve a teclear el encabezamiento del método, hay que incluir, antes del nombre del procedimiento o función, el nombre de la clase. El motivo es evidente, pues si tenemos la clase *TRaton* con un método *GuardarEnFichero*, tendremos dudas sobre qué *GuardarEnFichero* estamos implementando.

Ya hemos visto cómo programar la grabación de los datos del gato: creamos un fichero, escribimos el nombre en la primera línea ... un momento, ¿de qué gato estamos hablando? No debemos perder de vista el hecho de que hemos perdido un parámetro, al menos aparentemente. El lector sabe que el parámetro anda por ahí, pues corresponde al objeto sobre el cual se aplica el método. Y no se equivoca: el método tiene un parámetro implícito, llamado *Self*, cuyo tipo es *TGato*:

```
procedure TGato.GuardarEnFichero(const Fichero: string);
var
   Txt: TextFile;
```

```
begin
   AssignFile(Txt, Fichero);
   Rewrite(Txt);
   WriteLn(Txt, Self.Nombre);
   WriteLn(Txt, Self.Color);
   WriteLn(Txt, Self.Cola);
   WriteLn(Txt, Self.Alimento);
   CloseFile(Txt);
end;
```

Si busca la implementación anterior de *GuardarGatoEnFichero* verá que la diferencia principal es que todas las apariciones de parámetro *Gato* han sido reemplazadas por *Self*. Como es tan frecuente que dentro de un método se utilicen los atributos del objeto activo, Delphi permite eliminar las referencias a *Self* y utilizar directamente los nombres de atributos:

```
procedure TGato.GuardarEnFichero(const Fichero: string);
var
   Txt: TextFile;
begin
   AssignFile(Txt, Fichero);
   Rewrite(Txt);
   WriteLn(Txt, Nombre);
   WriteLn(Txt, Color);
   WriteLn(Txt, Cola);
   WriteLn(Txt, Alimento);
   CloseFile(Txt);
end;
```

El uso de *Self* es también válido para la aplicación de métodos al objeto activo. Dentro de un método de la clase *TGato* pueden ejecutarse instrucciones como las siguientes:

```
procedure TGato.DucharAlGato;
begin
   if LlevarAlAgua then
      CopiarAlPortapapeles;
   // Realmente:
   // if Self.LlevarAlAgua then Self.CopiarAlPortapapeles;
end;
```

Todos los métodos tienen un parámetro *Self*; el tipo de este parámetro es el tipo de clase a la que pertenece el método. Así, dentro del método *TRaton.CargarDesdeFichero*, *Self* pertenece al tipo *TRaton*. Insisto, una y otra vez, en que *Self* es un parámetro, y en tiempo de ejecución corresponde al objeto sobre el cual se aplica el método:

```
Pixie.ComerQueso('Roquefort');
// El Self de ComerQueso se identifica con Pixie
Dixie.ComerQueso('Manchego');
// El Self de ComerQueso se identifica con Dixie
```

El estado interno de un objeto

Parece ser que la introducción de métodos en el lenguaje es un simple capricho sintáctico: simplemente se ha enfatizado el papel desempeñado por uno de los muchos parámetros de un conjunto de funciones y procedimientos, asignándole una sintaxis especial. No es así, por supuesto. Más adelante, cuando estudiemos la herencia y el polimorfismo descubriremos las significativas ventajas de este pequeño cambio de punto de vista. Pero ya podemos explicar una magnífica característica de los objetos y métodos: nos permiten disminuir el número de parámetros necesarios en las llamadas a funciones. Y no me estoy refiriendo solamente a la transformación del objeto en el parámetro implícito *Self*.

La mejor demostración práctica de la afirmación anterior la podemos encontrar examinando alguna biblioteca de funciones convencionales que haya sido transformada en una biblioteca de clases. Y uno de los mejores ejemplos es la transformación de las funciones del Motor de Datos de Borland (BDE) por la Biblioteca de Controles Visuales (VCL) de Delphi. Tomemos una función al azar, por ejemplo, la función *DbiOpenTable*, que se utiliza para abrir una tabla directamente con el BDE; esta es su declaración:

```
function DbiOpenTable(hDb: hDBIDb; pszTableName: PChar;
   pszDriverType: PChar; pszIndexName: PChar;
   pszIndexTagName: PChar; iIndexId: Word; eOpenMode: DBIOpenMode;
   eShareMode: DBIShareMode; exltMode: XLTMode;
   bUniDirectional: Bool; pOptParams: Pointer;
   var hCursor: hDBICur): DBIResult;
```

DbiOpenTable tiene doce parámetros, de tipos diferentes. Para llamar a esta función tenemos que recordar en qué posición se encuentra cada parámetro y cuáles son los valores más frecuentes que se pueden utilizar. Por ejemplo, casi siempre en *pOptParams* se pasa el puntero nulo, **nil**, y en *bUniDirectional*, el valor *False*. Por los motivos explicados, se hace muy engorroso llamar a este tipo de funciones.

El equivalente orientado a objetos de esta función es el método *Open* de la clase *TTable*. Para abrir una tabla en Delphi hacen falta las siguientes instrucciones:

```
Table1.DatabaseName := 'DBDEMOS';
Table1.TableName := 'EMPLOYEE.DB';
Table1.Open;
```

El método *Open* se encarga internamente de llamar a *DbiOpenTable*, pero en vez de pedir los parámetros en la propia llamada, los extrae de atributos almacenados en el objeto. Por ejemplo, el parámetro del nombre de la tabla *pszTableName*, se extrae del atributo *TableName*. ¿Y qué sucede, digamos, con el parámetro *pszIndexName*? Pues que se extrae de un atributo llamado *IndexName*; no hemos asignado valores a *IndexName* porque el valor por omisión asignado automáticamente durante la creación del

objeto (la cadena vacía) nos vale para este ejemplo. También podemos despreocuparnos del orden de traspaso de los parámetros; el ejemplo anterior puede también programarse como sigue[10]:

```
Table1.TableName := 'EMPLOYEE.DB';
Table1.DatabaseName := 'DBDEMOS';
Table1.Open;
```

Por otra parte, el valor de retorno de *DbiOpenTable* se guarda en un atributo interno de *TTable*. Cuando queremos leer el siguiente registro directamente con el BDE, tenemos que utilizar la función *DbiGetNextRecord*, cuyo primer parámetro es, precisamente, el valor retornado después de la apertura exitosa de una tabla. En cambio, con Delphi solamente tenemos que llamar al método *Next*, que llamará internamente a la función del BDE, pero que extraerá los parámetros de esta función de las variables almacenadas en el objeto.

Al agrupar los datos del objeto junto a las operaciones aplicables al mismo, hemos logrado que estas operaciones se comporten "inteligentemente", deduciendo la mayor parte de sus parámetros del *estado interno del objeto*. De esta manera, se logran unos cuantos objetivos:

- Se evita el pasar una y otra vez los parámetros comunes a llamadas relacionadas.
- Se evita la dependencia del orden de definición de los parámetros.
- Se aprovecha la existencia de valores por omisión en los atributos, para no tener que utilizar cada vez el ancho de banda completo del canal de comunicación con el objeto.

De todos modos, tenemos que establecer una distinción, pues no todos los parámetros son buenos candidatos a almacenarse como parte del estado interno de un objeto. Por ejemplo, el parámetro *pszTableName* de la función *DbiOpenTable* sí lo es, pues nos puede interesar, en una llamada posterior, saber cuál es el nombre de la tabla con la cual trabajamos. Sin embargo, no creo que sea interesante para un gato saber cuál es el fichero del cual se extrajeron sus datos. Por esto, el método *CargarDesdeFichero* de la clase *TGato* sigue manteniendo un parámetro.

Por último, una aclaración. He escrito anteriormente que *DatabaseName*, *TableName* e *IndexName* son atributos de la clase *TTable*. Es inexacto: son *propiedades*, aunque Delphi hace todo lo posible para que creamos lo contrario. En el capítulo 9 estudiaremos qué son las propiedades; mientras tanto, piense en ellas como si fueran atributos. La abundancia de propiedades en las clases de Delphi es uno de los motivos por

[10] En honor a la verdad, tenemos un nuevo problema: se nos puede olvidar asignar algún atributo imprescindible para el funcionamiento del método. Cuando esto sucede, el método debe lanzar una excepción.

los que escribo acerca de gatos y ratones en este capítulo y en el siguiente, en vez de utilizar tablas, sesiones y otros objetos más "serios".

Parámetros por omisión

Delphi 4 ha introducido una técnica que ayuda en forma complementaria a reducir el número de parámetros "visibles" en los llamados a función. Se trata de poder especificar valores por omisión para los parámetros de un método, función o procedimiento. Estos valores se especifican durante la declaración de la rutina. Por ejemplo:

```
procedure TTabla.Abrir(const NombreIndice: string = '');
```

Si normalmente usted abre tablas sin especificar un índice, puede utilizar la siguiente instrucción:

```
Tabla.Abrir;
```

Sin embargo, tenga en cuenta que Delphi lo traducirá internamente a:

```
Table.Abrir('');
```

Hay una serie de reglas sencillas que deben cumplirse. Entre ellas:

- Los parámetros solamente pueden pasarse por valor o con el prefijo **const**. Esto es, no se permiten valores por omisión para parámetros por referencia.
- Las expresiones deben ser de tipos simples: numéricos, cadenas, caracteres, enumerativos. Para clases, punteros e interfaces, solamente se admite **nil**.
- Los parámetros con valores por omisión deben estar agrupados al final de la declaración.

Público y privado

No todos los métodos y atributos de un objeto son de interés para el público en general. Recuerde lo que decíamos, al principio del capítulo, acerca de las condiciones necesarias para que una descomposición modular redujera efectivamente la complejidad: los módulos debían ser independientes entre sí y sus canales de comunicación debían ser, mientras más estrechos, mejor. Si dejamos todos los atributos y métodos de un objeto al alcance de cualquiera, las reglas de comunicación con el objeto pueden hacerse engorrosas, o imposibles de establecer.

La programación tradicional ha reconocido desde siempre la necesidad de "esconder" información a los programadores. Muchas veces esta necesidad se ha interpre-

tado mal, como si "escondiéramos" información privilegiada que el resto del mundo no merece conocer. El verdadero sentido de esta protección es la reducción de complejidad asociada con esta técnica. Recuerde que el primer beneficiario puede ser usted mismo: esta semana desarrolla una magnífica colección de rutinas para manejar el microondas desde un ordenador. Al cabo de un mes, tiene que desarrollar un programa que las utilice, pero ya no recuerda todos los detalles internos. Si la interfaz de las rutinas contiene estrictamente la información necesaria, será más fácil para usted utilizarlas.

El mecanismo de protección de acceso de las clases en Object Pascal es realmente un añadido sobre el mecanismo de control de acceso existente en Pascal antes de la POO: la división de las unidades en interfaz e implementación. Todo lo que va en la interfaz se puede utilizar desde otra unidad; lo que se sitúa en la implementación, es interno a la unidad. Pero las declaraciones de clases suelen colocarse en la sección de interfaz de las unidades. Si no hacemos algo al respecto, todos los métodos y atributos declarados serán de dominio público.

Para solucionar este problema, Object Pascal permite dividir la declaración de una clase en secciones. Cada sección comienza con una de las palabras reservadas **public** ó **private**. En realidad, existen tres tipos adicionales de secciones: **protected**, **published** y **automated**, que se estudiarán más adelante. Las declaraciones de las secciones **public** son accesibles desde cualquier otra unidad. Por el contrario, lo que se declara en una sección **private** no es accesible desde otra unidad, bajo ninguna circunstancia, aunque la declaración de la clase se encuentre en la interfaz de la unidad. Si declaramos atributos y métodos y no hemos utilizado todavía ninguna de las palabras **public** o **private**, Delphi asume que estamos en una sección **public**. El siguiente ejemplo muestra una clase con secciones públicas y privadas:

```
type
   TGato = class
   private
      TieneMiedoAlAgua: Boolean;
      procedure LavadoConLengua;
   public
      Nombre: string;
      Color: TColor;
      Cola: Integer;
      Alimento: string;
      procedure CargarDesdeFichero(const Fichero: string);
      procedure GuardarEnFichero(const Fichero: string);
      procedure CopiarAlPortapapeles;
      function LlevarAlAgua: Boolean;
   end;
```

Puede haber varias secciones **public** y **private** en una clase, definidas en el orden que más nos convenga. Eso sí, dentro de cada sección hay que respetar el orden siguiente: los atributos primero y después los métodos.

Las variables de objetos son punteros

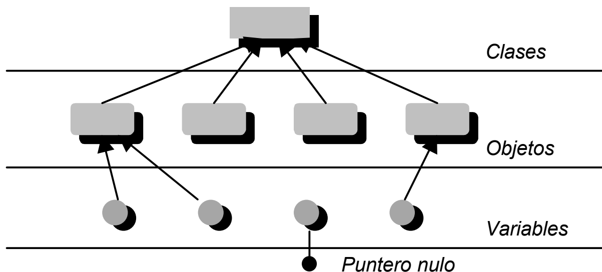
En todos los ejemplos con objetos presentados hasta ahora, hemos dejado deslizar una pequeña "imprecisión". Esta imprecisión se debe a que, en Delphi, las variables cuyo tipo es una clase no contienen al objeto en sí, sino un puntero al objeto.

```
var
   Gato: TGato;
```

La variable *Gato* ocupa solamente 4 bytes, los necesarios para el puntero a un gato. ¿Dónde almacenamos entonces el nombre, el color, la longitud de la cola y el alimento preferido? La respuesta es que, para poder trabajar con esta variable, hay que reservar memoria para el objeto en la zona de memoria dinámica de Delphi. Esta zona se conoce como *heap*; la traducción literal al castellano es "montón". En la siguiente sección veremos cómo utilizar métodos especiales, los *constructores* y *destructores*, para asignar y liberar esta memoria. Mientras el objeto no ha sido construido, la variable tiene un valor impredecible e inválido, y no debemos hacer referencias a sus atributos o aplicarle métodos. Es por este motivo que el siguiente ejemplo no funciona:

```
// Este código se ha mostrado antes
var
   Gato: TGato;
begin
   Gato.CargarDesdeFichero('C:\GARFIELD.CAT'); // ¡¡¡ERROR!!!
   // Gato no contiene un puntero válido
   // …
end;
```

Observe que no hace falta el operador de *derreferencia* de Delphi, el operador ^, para acceder a los atributos y métodos del objeto, pues el compilador se encarga automáticamente de asumirlo.



El hecho de que las variables de tipo clase contengan solamente el puntero al objeto tiene una importante consecuencia: la variable *no es* el objeto. Parece una distinción bizantina, pero no lo es. Dos variables de objeto pueden referirse a un mismo objeto,

del mismo modo que en la vida real "mi gato" se refiere al mismo objeto que "Edgar":

```
var
   MiGato, Edgar: TGato;
begin
   // Creamos un gato (ver la próxima sección)
   Edgar := TGato.Create;
   Edgar.CargarDesdeFichero('EDGAR.CAT');
   // Asignamos Edgar a MiGato
   MiGato := Edgar;
   // Después de la asignacion, sigue existiendo un solo objeto
   Edgar.Alimento := 'Tortilla de patatas';
   // También ha cambiado el gusto culinario de mi gato
   if MiGato.Alimento <> 'Tortilla de patatas' then
      ShowMessage('¡¡¡IMPOSIBLE!!!');
end;
```

Para comprender por qué Delphi exige que todos los objetos existan en el *heap* y se acceda a ellos mediante punteros, debemos conocer qué es el *polimorfismo*. Este concepto se estudiará en el siguiente capítulo.

Construcción y destrucción: el ciclo de la vida

En el último ejemplo de la sección anterior mostré cómo se reserva memoria para un objeto:

```
Edgar := TGato.Create;
```

Create es un constructor predefinido por Delphi, y todas las clases pueden hacer uso de él. En el caso de las clases creadas a partir de cero, como las que estamos examinando en este capítulo, *Create* no necesita parámetros. Para las clases más complejas, como las definidas en la VCL, *Create* puede necesitar parámetros. En el próximo capítulo, cuando estudiemos la *herencia*, volveremos a este tema.

Como se puede ver en el ejemplo, el constructor se llama como una función que retorna el puntero a un nuevo objeto de la clase especificada. A diferencia de lo que sucede con los métodos, la sintaxis habitual de las llamadas a constructores no es *objeto.método*, sino *clase.constructor*; esto es comprensible, pues cuando se ejecuta el constructor no tenemos el objeto a mano, sino que queremos crearlo.

El constructor predefinido *Create*, además de reservar la memoria necesaria, la inicializa automáticamente con ceros. Gracias a esto, los atributos del objeto de tipo numérico se inicializan con 0, los de tipo lógico con el valor *False*, las cadenas de caracteres con la cadena vacía, los valores enumerativos con el primer valor de la lista, etc.

Todas las clases cuentan también con un destructor predefinido: *Destroy*. Puede haber más de un destructor por clase, a diferencia de lo que sucede en C++. Esto se explica porque Pascal no llama automáticamente a los destructores cuando un objeto local sale de su alcance, como lo hace este lenguaje. Sin embargo, *Destroy* juega un importante papel en muchos procedimientos de la VCL, por lo que en la práctica es el destructor que siempre se utiliza.

Existe una forma alternativa de destruir un objeto: el método *Free*. Este método llama al destructor *Destroy* de la clase, pero sólo después de verificar que el puntero del objeto no es el puntero vacío. Se recomienda ejecutar *Free* para destruir objetos, en vez de llamar directamente a *Destroy*, por las siguientes razones:

- El código de la llamada ocupa menos espacio. Esto se explicará en el capítulo 7, en relación con la herencia y polimorfismo.
- Evitamos problemas asociados con las *construcciones parciales*. Al final del capítulo 11, que trata acerca de las excepciones, veremos el porqué.

Definiendo constructores y destructores

Si tenemos que crear muchos gatos en una aplicación, cada vez que lo hagamos nos encontraremos con un gato vacío en nuestras manos. De modo que, una y otra vez, tendremos que asignarle valores a sus atributos, aunque la cola del gato promedio sea de 15 centímetros, y su alimento preferido sea el pescado. También se nos puede olvidar inicializar el gato o asignarle valores inconsistentes. Por lo tanto, es deseable disponer de un constructor que nos libre de una vez por todas de estas tediosas inicializaciones:

```
type
   TGato = class
      // Atributos
      // …
      // Métodos
      constructor Create(const ANombre: string; AColor: TColor);
   end;
```

Un constructor se declara en Delphi como si fuera un procedimiento más de la clase, pero sustituyendo la palabra **procedure** por **constructor**. La declaración debe realizarse después de haber declarado los atributos de la sección; casi siempre el constructor se sitúa en la parte pública de la clase, pues un constructor privado tiene poca utilidad. Mi constructor de gatos se llama *Create* por uniformidad de estilo; para las clases simples, da lo mismo el nombre que se le dé. Pero para las clases de la VCL, como veremos en el siguiente capítulo, el nombre es importante.

Hemos quedado en que un constructor se ejecuta de forma similar a una función. Pero la implementación es similar a la de un procedimiento. Además, tampoco hay que hacer nada especial para reservar la memoria; a partir de la primera línea del cuerpo de constructor, se asume que la memoria del objeto ya está reservada:

```
constructor TGato.Create(const ANombre: string; AColor: TColor);
begin
   Nombre := ANombre;
   Color := AColor;
   Cola := 15;
   Alimento := 'Pescado';
end;
```

El nuevo constructor se utiliza del siguiente modo:

```
   GatoPromedio := TGato.Create('Micifuz', clInfoBk);
   // clInfoBk es el color de fondo de la Ayuda de Windows 95
```

¿Es frecuente que los datos de un gato se extraigan de un fichero? Definamos entonces un constructor que inicialice los datos de este modo. Una clase puede tener cuantos constructores haga falta, siempre que tengan nombres diferentes:

```
type
   TGato = class
      // …
      constructor CreateFromFile(const Fichero: string);
   end;

constructor TGato.CreateFromFile(const Fichero: string);
begin
   CargarDesdeFichero(Fichero);
end;
```

Ahora, también podemos crear gatos en el siguiente estilo:

```
   GatoConBotas := TGato.CreateFromFile('PERRAULT.CAT');
```

> Delphi 4 ha introducido la directiva **overload**, que permite definir métodos, constructores y funciones en general con el mismo nombre, siempre que el número de parámetros o sus tipos permita deducir a cuál método, constructor o función en particular se intenta llamar. Esta directiva será estudiada en el capítulo siguiente.

El lector se dará cuenta de que hemos dejado el método *CargarDesdeFichero*, por si necesitamos hacer caso omiso de los datos de un gato y sustituirlos por los almacenados en cierto fichero. Ahora bien, también podemos utilizar el constructor sobre un objeto ya creado; en ese caso, no se reserva la memoria del objeto, pues ya existe, pero se ejecutan las instrucciones de inicialización:

```
Gato := TGato.CreateFromFile('GATO.1');
// …
Gato.CreateFromFile('GATO.2');    // ¡¡¡NUEVO!!!
```

Hay que tener cuidado con el último estilo de ejecución de constructores. Si el objeto sobre el que se aplica el constructor no ha sido creado, se produce una excepción; las excepciones se verán más adelante. Es muy fácil confundirse y llamar al constructor sobre el objeto, como en la tercera línea del ejemplo, en vez de ejecutarlo como en la primera línea, sobre la clase, aunque lo que pretendamos sea reservar memoria: el compilador no va a protestar por nuestro error.

La definición de un destructor es similar a la de los constructores, excepto que la palabra clave **destructor** sustituye a **constructor**. Las instrucciones definidas en el destructor se ejecutan justo antes de liberar la memoria del objeto. En el próximo capítulo seguiremos viendo ejemplos de definición de constructores y destructores.

Capítulo

7

Herencia y polimorfismo

NADIE COMIENZA UN PROGRAMA PARTIENDO DE CERO. En la época de la Programación Estructurada, un programador abarrotaba la superficie de su escritorio con bibliotecas de funciones que resolvían ciertas tareas de la aplicación que se traía entre manos. En nuestros días, el mismo programador reúne las bibliotecas de clases de objetos que le ofrecen la funcionalidad que necesita. Con las técnicas analizadas en el capítulo anterior, quizás solamente necesite la mitad del espacio en el escritorio para almacenar las cajas de los productos de software.

Pero, ¿qué pasa si los componentes de software disponibles, sean rutinas o clases, no son exactamente lo que deseamos? Si nuestros componentes básicos son rutinas, lo que no esté previsto en los parámetros, no se puede lograr. Y si el autor de la rutina previó demasiado, tendremos un exceso de parámetros que dificultarán el uso normal de la misma. La solución la tiene, nuevamente, la Programación Orientada a Objetos. En el presente capítulo aprenderemos los mecanismos que disponemos para modificar o ampliar la funcionalidad básica de una clase.

Herencia = Extensión + Especialización

La *herencia* es una técnica de la POO que consiste en definir una nueva clase a partir de una clase ya existente, de modo que contenga inicialmente la misma estructura y comportamiento de la clase base. Posteriormente, al nuevo tipo se le pueden añadir nuevos atributos y métodos, y se puede cambiar la implementación de ciertos métodos. En Delphi, la herencia se indica sintácticamente al principio de la definición de una clase:

```
type
   TVentanaEdicion = class(TForm);
```

TVentanaEdicion es una clase que contiene exactamente los mismos atributos y métodos que *TForm*, la clase que representa los formularios de Delphi. La declaración anterior es equivalente a la siguiente:

```
type
   TVentanaEdicion = class(TForm)
   end;
```

Una vez que hemos heredado de una clase, tenemos tres posibilidades:

- Añadir nuevos datos al estado interno del objeto, definiendo nuevos atributos.
- Añadir nuevos métodos, extendiendo el comportamiento del objeto.
- Modificar el comportamiento del objeto, redefiniendo el cuerpo de métodos existentes.

La técnica más fácil es añadir nuevos atributos a una clase, para ampliar las capacidades de almacenamiento del objeto:

```
type
   TVentanaEdicion = class(TForm)
      Memo1: TMemo;
   private
      { Private declarations }
   public
      { Public declarations }
      NombreFichero: string;
   end;
```

En el ejemplo mostrado se añaden dos nuevos atributos: *Memo1*, que es un editor de textos, y *NombreDeFichero*, que es una cadena preparada para almacenar el nombre del fichero cargado en *Memo1*. Ambos atributos están declarados en secciones públicas. Todo el resto de la parafernalia de la división en secciones se debe al mecanismo empleado por el entorno de desarrollo de Delphi para la edición visual. *Memo1* ha sido añadido desde la Paleta de Componentes de Delphi, mientras que *NombreDeFichero* se ha tecleado directamente dentro del código fuente de la unidad.

Añadir nuevos atributos a una clase no es muy útil por sí mismo. ¿Para qué queremos nuevos atributos que no son utilizados por los métodos existentes? O definimos nuevos métodos que trabajen con los atributos añadidos, o modificamos los métodos heredados para que aprovechen la información de la que ahora dispone el objeto. Para mostrar cómo se definen nuevos métodos, definamos un método *LeerFichero* que cargue el contenido de un fichero de texto en el editor, asigne el nombre al atributo *NombreDeFichero* y modifique, finalmente, el título de la ventana:

```
type
   TVentanaEdicion = class(TForm)
      Memo1: TMemo;
   private
      { Private declarations }
```

```
    public
      { Public declarations }
      NombreFichero: string;
      procedure LeerFichero(const Nombre: string);
    end;
```

La implementación del método se realiza como de costumbre:

```
procedure TVentanaEdicion.LeerFichero(const Nombre: string);
begin
   Memo1.Lines.LoadFromFile(Nombre);
   Memo1.Modified := False;
   Caption := ExtractFileName(Nombre);
   NombreFichero := Nombre;
end;
```

Caption es una propiedad para acceder y modificar el título de la ventana; en el capítulo anterior recomendé considerarlas, por el momento, como si fueran atributos. Observe que tenemos acceso a *Caption* aunque no ha sido definida en *TVentanaEdicion*; está definida en *TForm*, y al heredar de esta clase tenemos derecho a considerar al atributo como si fuera nuestro. Más adelante precisaremos a qué tipo de atributos se nos otorga el derecho de acceso.

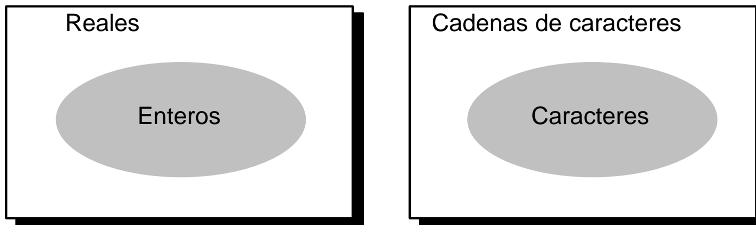
Un gato con botas sigue siendo un gato

Hay una regla en Programación Orientada a Objetos cuya importancia jamás podrá ser exagerada: un objeto derivado de una clase puede utilizarse en cualquier lugar en que sea aceptado un objeto de la clase original. En el Pascal original, no se podían mezclar tipos diferentes: a una variable de tipo "manzana" no se le podía asignar un valor de tipo "patata". La única excepción se hacía con los valores enteros y reales: a un real se le puede asignar un entero. Dicho con más exactitud, donde puede ir un valor real, se puede utilizar un entero, aunque no a la inversa:

```
var
   I: Integer;
   D: Double;
begin
   I := 0;
   D := I + 1;        // Correcto: se asigna un entero a un real
   D := Sin(I);       // Correcto, aunque Sin espera un valor real
   I := D - 1;        // ¡¡¡Incorrecto!!!
end;
```

También se aplica la misma regla de conversión para los caracteres, el tipo *Char*, y las cadenas de caracteres. Desde un punto de vista lógico, la regla puede aplicarse porque, en ambos casos, uno de los tipos es un subconjunto del otro. Este tipo de relaciones se ha representado desde siempre matemáticamente con los llamados *diagra-*

mas de Venn. Muestro a continuación los diagramas correspondientes a los tipos anteriores:



El primer diagrama informa, por ejemplo, que todos los enteros son, a la vez, valores reales, y que existen reales que no son valores enteros.

Resulta que este tipo de situación se produce ahora con las clases derivadas por herencia. Los objetos del tipo *TVentanaEdicion* pueden utilizarse en cualquier contexto en el que era válido un objeto *TForm*, y un objeto de clase *TGatoConBotas* puede sustituir a un simple *TGato*. Esto quiere decir que:

- A una variable de tipo *TGato* se le puede asignar un objeto de tipo *TGatoConBotas*.

```
var
   G: TGato;
begin
   G := TGatoConBotas.Create('Pepe', clBlack);
   // …
end;
```

- A una rutina que tenga un parámetro de tipo *TGato* se le puede pasar un objeto de tipo *TGatoConBotas* en ese parámetro.

```
procedure DucharAlGato(Gato: TGato);
begin
   // …
end;

var
  G: TGatoConBotas;
begin
   G := TGatoConBotas.Create('Pepe', clGray);
   DucharAlGato(G);
   // …
end;
```

- En particular, los métodos de la clase *TGato* tienen un parámetro *Self* declarado implícitamente de tipo *Gato*. Pero la definición de herencia exige que incluso los métodos heredados de su clase base por *TGatoConBotas* puedan aplicarse sobre objetos de este último tipo. Este es un caso particular del ejemplo anterior.

Es significativo que estas asignaciones no son, realmente, conversiones de tipos, como puede suceder con las asignaciones de enteros a variables reales. El objeto que se asigna o se pasa como parámetro sigue manteniendo su formato original. La razón de esta "libertad" puede comprenderse si examinamos someramente el formato de los objetos pertenecientes a un par de clases relacionadas mediante la herencia:

Gato
Nombre
Color
Cola
Alimento

Gato con Botas
Nombre
Color
Cola
Alimento
MarcaDeBotas
Talla

Partamos del hecho de que Delphi se refiere a todos los objetos mediante punteros. Si a una rutina que necesita un parámetro de tipo *TGato* le pasamos un objeto de tipo *TGatoConBotas*, la rutina no notará el cambio. Cuando necesite el nombre del animal, lo encontrará en su posición; lo mismo sucede con todos los atributos del gato común, hasta el alimento. Y también se aplica a los métodos.

En cambio, si una rutina necesita un *TGatoConBotas*, no se le puede pasar un gato normal, pues cuando vaya a buscar la talla de las botas se encontrará fuera del área de memoria del objeto. Ya sabe, fallos de protección general, tirones de pelos, grandes dosis de café y aspirina... Por eso, el compilador no permite este tipo de sustituciones:

```
var
   G: TGato;
   GB: TGatoConBotas;
begin
   G := TGatoConBotas.Create('Pepe', clSilver);     // ¡Correcto!
   GB := G;                                         // ¡Incorrecto!
   // …
end;
```

Al lector le puede resultar chocante que, aunque la variable *G* apunte a un objeto de la clase derivada, no se permita recuperar este objeto en la variable *GB*. El motivo de este tipo de prohibición se hace más claro si modificamos el código anterior del siguiente modo:

```
var
   G: TGato;
   GB: TGatoConBotas;
begin
   if DayOfWeek(Date) = 1 then                  // ¿Hoy es domingo?
      G := TGato.Create('Pepe', clWhite)
   else
      G := TGatoConBotas.Create('Pepe', clWhite);
```

```
   GB := G;                                      // ¡¡¡Incorrecto!!!
   // …
end;
```

El compilador no puede adivinar si el programa se va a ejecutar un lunes o un fin de semana (esto último, es siempre poco probable), de modo que no puede saber de antemano si la última instrucción es correcta o incorrecta. Ante la duda, opta por prohibir la asignación, aún dejando a algunos casos correctos "fuera de la ley". Más adelante, presentaremos los operadores **is** y **as**, que ayudan en estos casos.

Pero la consecuencia más importante, a la cual volveremos en breve es que, en tiempo de compilación, no es posible predecir qué tipo de objeto está asociado a una variable de clase:

```
var
   G: TGato;
begin
   if DayOfWeek(Date) = 1 then               // ¿Hoy es domingo?
      G := TGatoConBotas.Create('Pepe', clWhite)
   else
      G := TGatoMecanico.Create('Pepe', clWhite);
   G.Maullar;
   // ¿Qué tipo de gato contiene G en la instrucción anterior?
end;
```

Solamente podemos afirmar que el objeto al cual se refiere *G* sabe maullar. Hay una dualidad, entonces, cuando examinamos una variable de clase: tiene un tipo declarado para el compilador, pero el objeto al que apunta, si es que hay alguno, puede pertenecer al tipo declarado o a cualquier tipo derivado de él.

La clase TObject

Otro aspecto de importancia en relación con la herencia es que todas las clases de Delphi descienden, directa o indirectamente, de la clase *TObject*. Incluso cuando declaramos clases sin ancestros aparentes, se asume que la clase base es *TObject*:

```
type
   TMiClase = class
      // …
   end;
```

… equivale a …

```
type
   TMiClase = class(TObject)
      // …
   end;
```

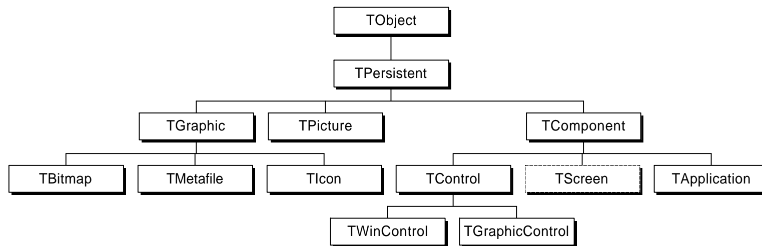
TObject ofrece la funcionalidad mínima exigible a un objeto de Delphi. La mayor parte de esta funcionalidad está dedicada a suministrar información de tipos en tiempo de ejecución (*Runtime Type Information*, ó *RTTI*), por lo cual tendremos que postergar un poco su estudio. Por otra parte, en la clase *TObject* se definen el constructor *Create*, el destructor *Destroy* y el método *Free*, que hemos mencionado al finalizar el capítulo anterior:

```
type
   TObject = class
      constructor Create;
      destructor Destroy; virtual;
      procedure Free;
      // …
   end;

procedure TObject.Free;
begin
   if Self <> nil then Destroy;
end;
```

Como todas las clases de Delphi deben heredar directa o indirectamente de *TObject*, no se definen atributos en esta clase. Un atributo definido en *TObject* sería heredado por todos los demás objetos de Delphi, ocupando espacio innecesariamente.

Jerarquías de herencia

La forma más extendida de mostrar la relación de herencia entre las distintas clases definidas en un lenguaje o biblioteca es el esquema denominado *jerarquía de herencia*. En un diagrama de jerarquía, de cada clase parten líneas que la enlazan a las clases que heredan de forma directa de ella. En Delphi, además, como todas las clases descienden de *TObject*, el nodo correspondiente actúa como la *raíz* del árbol. La siguiente figura muestra las relaciones entre algunas clases de Delphi escogidas al azar:



Expliquemos un poco quién es quién en este diagrama, examinando las bifurcaciones mas significativas:

- *TObject* es la base de toda la jerarquía. Incluso cuando se declara una clase sin ancestros, *TObject* está detrás de la declaración.
- De *TPersistent* descienden los tipos que Delphi puede almacenar en un fichero *dfm*, y restaurar posteriormente. Casi todos los objetos de Delphi pertenecen a la rama del árbol encabezada por *TPersistent*. Los objetos internos a una aplicación creados por el programador, sin embargo, son candidatos a heredar directamente de *TObject*, sin pasar por esta clase. Tal es el caso del gato, de las listas de punteros y de los flujos de datos (*streams*) para trabajar con ficheros.

- Todos los componentes descienden de *TComponent*, que añade la posibilidad de colocar a la clase en la Paleta de Componentes de Delphi, y poder trabajar con sus objetos en tiempo de diseño.
- Los componentes visuales descienden de *TControl*; el resto, son componentes no visuales, como los relacionados con el acceso a bases de datos (*TTable*, *TQuery*, *TDatabase*).
- Los componentes visuales, o *controles*, se dividen a su vez en dos grandes ramas: los controles de ventanas, derivados de *TWinControl*, y los controles gráficos, que descienden de *TGraphicControl*. Los primeros controlan objetos definidos por el sistema operativo: *TEdit*, *TButton*, *TCheckBox*, y las ventanas y diálogos: *TForm*. Son los controles más potentes y generales, pero consumen los recursos internos de Windows. Los controles gráficos, en cambio, son gestionados completamente por Delphi. Consumen menos recursos, pero sufren de limitaciones, como la de no poder recibir el foco del teclado.

Para el desarrollador de componentes de Delphi, es importante tener bien claras estas relaciones entre clases: saber quién desciende de quién. Sin embargo, para la programación nuestra de cada día, no hay que tomarse muy a pecho el aprendizaje de la jerarquía de clases. Para el desarrollo de aplicaciones en Delphi, el programador normalmente utiliza los componentes de la Paleta *tal como son*, sin preocuparse por sus antecedentes. Sólo en casos especiales debe saber, por ejemplo, que tanto *TTable* (tablas de bases de datos) como *TQuery* (consultas SQL) heredan indirectamente de la clase *TDataSet*, y que si vemos en un método con un parámetro de tipo *TDataSet*, puede que corresponda lo mismo a una tabla que a una consulta.

Herencia múltiple, ¿sí o no?

El modelo de herencia que hemos estudiado se conoce como *herencia simple*: cada clase tiene, como máximo, un solo ancestro. En cambio, ciertos lenguajes, como Eiffel y C++, permiten que una clase se derive de varias clases simultáneamente; a esto se le llama *herencia múltiple*. La nueva clase hereda todas las características de sus padres, y la regla de compatibilidad de tipos polimórfica permite utilizar objetos de la clase hija en cualquier lugar en que se acepta un objeto de alguno de los ancestros.

Existe una enconada y aparentemente interminable disputa entre los partidarios (👍) y detractores (👎) de la herencia múltiple. He aquí algunos de los argumentos:

👎 La herencia múltiple es un síntoma de un mal diseño de las relaciones entre clases.
👍 Las jerarquías basadas en la herencia simple son utopías académicas.
👎 La implementación de la herencia múltiple por un compilador es complicada.

- 👍 Existen, desde hace ya bastante tiempo, implementaciones exitosas y eficientes de lenguajes con herencia múltiple, especialmente C++.
- 👎 No está claro cuál es la mejor política a seguir cuando una clase hereda, directa o indirectamente, otra clase más de una vez (*herencia repetida*). Las reglas que utiliza C++, por ejemplo, son complicadas y difíciles de explicar (*clases virtuales*). Las reglas adoptadas por Eiffel son completamente diferentes y, para mi gusto, más racionales. Pero complican sobremanera la implementación y el control de tipos por el compilador.
- 👍 (… silencio total …)

En realidad, hay dos contextos en los que surge con mayor frecuencia la necesidad de utilizar herencia múltiple. El primero es cuando tengo una biblioteca de clases de ventanas y otra de clases de listas. Si necesito una ventana que sea también una lista, puedo "mezclar" ambas bibliotecas mediante la herencia múltiple. Este contexto es el que el detractor considera un mal diseño, pero que el partidario considera una situación real y frecuente.

La otra situación sucede cuando tenemos un par de clases, una de ellas con métodos abstractos y la otra completamente concreta. Mediante la segunda clase queremos dar una implementación a los métodos abstractos de la primera. De la primera clase, heredamos realmente los métodos de la "interfaz", mientras que la segunda aporta el código. Nadie puede negar que, correctamente planteado, éste es un enfoque elegante al diseño de clases, y Borland hace uso del mismo en su biblioteca de clases contenedoras para C++. Ahora bien, existe una alternativa a la herencia múltiple que, evitando los problemas asociados a esta técnica, permite mezclar definiciones abstractas de clases con implementaciones concretas: las *interfaces*. Antes de Delphi, el lenguaje Java incluía este recurso. Las interfaces no se estudiarán en este capítulo.

Redefinición de métodos

Antes habíamos dicho que con la herencia lográbamos dos objetivos: añadir nuevas características a una clase y modificar características existentes. Este segundo objetivo es posible, fundamentalmente, gracias a la *redefinición de métodos virtuales*, un mecanismo que permite adaptar la implementación de un método de acuerdo al tipo de objeto.

Para que un método pueda ser redefinido de manera efectiva, debe haber sido declarado en la clase base como método *virtual*, añadiendo la palabra reservada **virtual** después de su declaración:

```
type
   TAnimal = class
      // …
      procedure Chillar(Duracion: Word); virtual;
   end;
```

Cuando derivemos una clase directa o indirectamente de *TAnimal*, tendremos la posibilidad de redefinir la implementación del método *Chillar*. Primero hay que anunciar la redefinición en la declaración de la nueva clase:

```
type
   TVaca = class(TAnimal)
      // …
      procedure Chillar(Duracion: Word); override;
   end;
```

La directiva **virtual** ha sido sustituida por **override**. Si utilizamos **virtual** nuevamente, o no especificamos directiva alguna, el compilador no protesta, pero el efecto logrado es completamente diferente y no tiene sentido o aplicación práctica. Tome nota de que los parámetros en la redefinición deben ser exactamente iguales que los del original. La implementación del método redefinido transcurre igual que para cualquier otro tipo de método:

```
procedure TVaca.Chillar(Duracion: Word);
begin
   if Duracion <= 1 then
      ShowMessage('Mu')
   else
      ShowMessage('Muuuu');
end;
```

La parte más interesante de este mecanismo de redefinición está en relación con la posibilidad de que una variable contenga un objeto derivado del tipo con que ha sido declarada. Como consecuencia de lo anterior, hay dos tipos relacionados con una variable de clase:

- El tipo "estático", que es el utilizado en la declaración de la variable.
- El tipo "real" o "dinámico", que es la clase a la cual pertenece el objeto que tiene asociado.

Cuando aplicamos un método a un objeto, utilizando la sintaxis *objeto.método*, tenemos dos alternativas para decidir qué método utilizar:

- Utilizar el tipo estático, el de la declaración de la variable.
- Utilizar el tipo dinámico, el asociado al objeto real.

Cuando un método ha sido declarado virtual, Delphi utiliza siempre la segunda alternativa; si el método no es virtual, se aplica la primera. El uso del tipo estático puede

ser peligroso cuando hay redefiniciones, pues no se llama a la versión más actualizada del método. Por eso, aunque Delphi permite crear una nueva implementación de un método no virtual, no es recomendable. La regla de oro dice:

"Si de un método puede afirmarse con certeza que nunca será redefinido, no lo declare virtual"

Supongamos que en la clase *TAnimal* definimos un método con el propósito de que la criatura pida auxilio en situaciones de peligro:

```
type
   TAnimal = class
      // …
      procedure SOS;
      procedure Chillar(Duracion: Word); virtual;
   end;
```

Todos los animales pedirán ayuda mediante el alfabeto Morse: tres chillidos cortos, una S, tres chillidos largos, una O. Como este comportamiento no va a cambiar para las clases derivadas sucesivamente de *TAnimal*, no ha sido declarado virtual. La implementación de *SOS*, que será compartida por todo el mundo animal, es la siguiente:

```
procedure TAnimal.SOS;
begin
   Chillar(1); Chillar(1); Chillar(1);
   Chillar(2); Chillar(2); Chillar(2);
   Chillar(1); Chillar(1); Chillar(1);
end;
```

Aunque *SOS* no es virtual, cuando un animal ejecuta este método se produce el sonido adecuado:

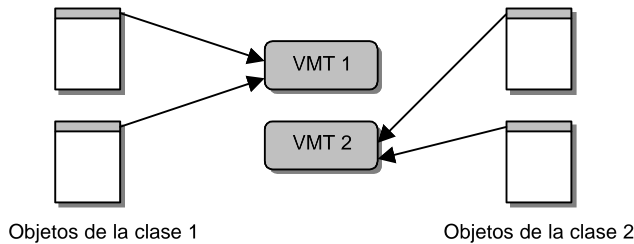
```
var
   A: TAnimal;
begin
   if Random(2) = 0 then
      A := TVaca.Create('Gertrude')
   else
      A := TCaballo.Create('Fidel');
   A.SOS;
   A.Free;
end;
```

En la llamada a *SOS* se pasa el puntero al objeto en el parámetro implícito *Self*. Dentro de la implementación del método, *Self* pertenece al tipo *TAnimal*; si *Chillar* fuese no virtual, se llamaría al método definido en esta clase, lo cual sería incorrecto. Pero como *Chillar* es virtual, Delphi verifica el tipo del objeto asociado y busca la implementación adecuada: un angustioso mugido o un desesperado relincho.

A la combinación de la regla de compatibilidad entre clases relacionadas por la herencia, y la determinación del método a aplicar mediante el tipo asociado al objeto, y no a la variable, se le denomina *polimorfismo*.

La Tabla de Métodos Virtuales

¿Cómo funciona el mecanismo de llamada a métodos virtuales? La idea es que cada objeto tiene, en su espacio reservado en memoria, un campo oculto que es un puntero a una estructura denominada la *Tabla de Métodos Virtuales* (VMT). Todos los objetos pertenecientes a una misma clase apuntan a la misma VMT. El principio básico es: una clase, una VMT. La figura que se muestra a continuación representa dos clases, cada una de la cual tiene dos objetos creados:



La Tabla de Métodos Virtuales contiene datos acerca de la clase. Aunque esta estructura es necesaria para cualquier lenguaje de programación orientado a objetos, el formato de estos datos depende de las necesidades del lenguaje. Un lenguaje como C++, que permite la herencia múltiple, necesita una estructura bastante compleja para sus VMTs. En cambio, con la herencia simple el formato es mucho más sencillo. La parte principal de las VMTs de Delphi es un vector de punteros al segmento de código de la aplicación. Por cada método virtual hay una entrada en esta tabla. Si una clase hereda cinco métodos virtuales, y declara tres nuevos métodos en ese estilo, tendrá ocho punteros en su VMT. Las redefiniciones de métodos virtuales no cuentan para esta aritmética, en cambio.

El compilador asocia a cada método virtual de una clase un número entero que representa su posición en la VMT. Se cumple entonces la siguiente regla:

> *"Si al procedimiento* Chillar *se le asigna la decimoquinta posición en la VMT de la clase* TAnimal*, esta posición se mantiene en cualquier clase derivada"*

Supongamos que la clase *TAnimal* ha sido definida de esta manera:

```
type
   TAnimal = class
      // …
      procedure Chillar(Duracion: Word); virtual;
      procedure Rascarse; virtual;
   end;

procedure TAnimal.Chillar(Duracion: Word);
begin
   // Este código comienza en la dirección hexadecimal $00001234
   // …
end;

procedure TAnimal.Rascarse;
begin
   // Este código comienza en la dirección hexadecimal $00002345
   // …
end;
```

La tabla de punteros de la VMT de la clase *TAnimal* tiene el siguiente formato:

$00001234	← La dirección del método *Chillar* (15)
$00002345	← La dirección del método *Rascarse*(16)

He asignado la decimoquinta posición a *Chillar* como ejemplo; recuerde que todas las clases heredan de *TObject*, y que esta clase base tiene métodos virtuales definidos. Ahora creamos la nueva clase *TVaca*, derivada de *TAnimal*:

```
type
   TVaca = class(TAnimal)
      // …
      procedure Chillar(Duracion: Word); override;
      procedure Rumiar; virtual;
   end;

procedure TVaca.Chillar(Duracion: Word);
begin
   // Este nuevo código comienza en la dirección $00003456
   // …
end;

procedure TVaca.Rumiar;
begin
   // Este código comienza en la dirección $00004567
   // …
end;
```

La vaca chilla de forma diferente al chillido por omisión, e inventa aquello de rumiar la comida y los pensamientos. Esta es la VMT de una vaca:

$00003456	← La dirección del método *Chillar* (15)
$00002345	← La dirección del método *Rascarse* (16)
$00004567	← La dirección del método *Rumiar* (17)

Por último, supongamos que tenemos una variable *Animal*, de tipo *TAnimal*, que puede apuntar igualmente a un objeto *TVaca* o a un *TAnimal*. La llamada a método *Animal.Chillar* se compila de la siguiente manera:

> *"Tomar el puntero al objeto y quedarnos con el puntero a la VMT, que siempre está al principio del objeto. Buscar en la VMT el puntero de la decimoquinta posición y ejecutar esa rutina"*

Una llamada a un método virtual ocupa, por lo tanto, más código que una llamada a un método estático, para el cual se conoce, en tiempo de compilación, a qué dirección saltar directamente. También estas llamadas consumen más tiempo. Es por esto que no se sigue el criterio de declarar automáticamente virtuales a todos los métodos (lo cual sería más fácil de explicar, sin embargo). En consecuencia, muchas veces se efectúa la llamada al método virtual desde un método estático, con el propósito fundamental de ahorrar espacio de código. Anteriormente vimos la existencia de *Free* y *Destroy* como métodos alternativos de destruir un objeto. Una llamada a *Destroy* se compila como una llamada a un método virtual, ocupando más código. La llamada a *Free* ocupa menos espacio, y también ejecuta de forma indirecta a *Destroy*. En cambio, debido al doble salto (primero a *Free* y luego a *Destroy*) esta última forma de llamada requiere mas tiempo total.

Utilizando el comportamiento heredado

En la mayor parte de los casos en que se redefine un método es para realizar pequeñas mejoras y adiciones a un comportamiento ya definido. Es conveniente, por lo tanto, poder utilizar la implementación anterior del método en el nuevo código. Para llamar a la implementación anterior de un método se utiliza la palabra **inherited** seguida del nombre del método y sus parámetros.

Esta técnica es particularmente necesaria en la redefinición de constructores y destructores. Cuando creamos un constructor en una clase nueva queremos inicializar en primer lugar los campos que hemos heredado, ¿quién mejor que el constructor de la clase ancestro para realizar esta tarea?

```
type
   THumano = class(TPrimate)
   public
      CreditCard: string;
      constructor Create(const AName, ACreditCard: string);
   end;

constructor THumano.Create(const AName, ACreditCard: string);
begin
   // Inicializar el nombre, cerebro, pelo y las malas pulgas
   inherited Create(AName);
```

```
    // Inicializar nuestras características exclusivamente humanas
    CreditCard := ACreditCard;
end;
```

Ahora bien, la técnica puede utilizarse en cualquier otro tipo de método. Supongamos que hemos definido una clase que representa un círculo, con un método virtual *Dibujar* al cual se le suministra como parámetro un objeto de tipo *TCanvas*: las superficies de dibujo de Delphi.

```
type
   TCirculo = class
      Centro: TPoint;
      Radio: Integer;
      // …
      procedure Dibujar(ACanvas: TCanvas); virtual;
   end;

procedure TCirculo.Dibujar(ACanvas: TCanvas);
begin
   ACanvas.Ellipse(Centro.X - Radio, Centro.Y - Radio,
      Centro.X + Radio, Centro.Y + Radio);
end;
```

Podemos entonces definir una rosquilla y redefinir su forma de dibujarse del siguiente modo:

```
type
   TRosquilla = class(TCirculo)
      Interior: TCirculo;
      // …
      procedure Dibujar(ACanvas: TCanvas); override;
   end;

procedure TRosquilla.Dibujar(ACanvas: TCanvas);
begin
   inherited Dibujar(ACanvas);
   Interior.Dibujar(ACanvas);
end;
```

Sobrecarga: hay muy pocos nombre buenos

En muchas ocasiones, tenemos que efectuar operaciones que producen el mismo resultado a partir de parámetros diferentes. El caso clásico es la creación de objetos, pues muchas veces existe más de una forma de crear un objeto. Por ejemplo, una fracción puede crearse especificando su numerador y su denominador, o copiarse desde otra fracción o a partir de un número real. Hasta Delphi 3, tal clase debía definirse de este modo:

```
type
   TFraccion = class
   protected
      Num: LongInt;
      Den: LongWord;
   public
      constructor CreateFrac(Otra: TFraccion);
      constructor CreateReal(AReal: Real);
      constructor CreateInt(ANum: LongInt; ADen: LongWord);
   end;
```

Sin embargo, Delphi 4 permite utilizar un mismo nombre para todas estas operaciones, con la condición primaria de que todas ellas se declaren con la nueva directiva **overload**:

```
type
   TFraccion = class
   protected
      Num: LongInt;
      Den: LongWord;
   public
      constructor Create(Otra: TFraccion); overload;
      constructor Create(AReal: Real); overload;
      constructor Create(ANum: LongInt; ADen: LongWord); overload;
   end;
```

La clave está en que el compilador podrá posteriormente distinguir entre ellas basándose en los argumentos que le suministremos:

```
var
   F1, F2: TFraccion;
begin
   F1 := TFraccion.Create(1);         // La antigua CreateReal
   F2 := TFraccion.Create(F1);        // La antigua CreateFrac
end;
```

Podemos ir más allá, y mezclar el nuevo recurso del lenguaje con los parámetros por omisión, definiendo el tercer constructor de este modo:

```
type
   TFraccion = class
      // …
      constructor Create(ANum: LongInt; ADen: LongWord = 1);
         overload;
   end;
```

Ahora tenemos que tener un poco más de cuidado, al adivinar cuál constructor es el que va a emplear el compilador:

```
var
   F1, F2: TFraccion;
begin
   F1 := TFraccion.Create(1);         // Ahora es CreateInt
```

```
    F2 := TFraccion.Create(1.0);      // La antigua CreateReal
end;
```

A una clase que ya tiene un método con la directiva **overload**, se le puede añadir en una clase derivada otro método por sobrecarga, siempre que el compilador pueda distinguir entre el viejo y el nuevo método. Además, la sobrecarga de nombres no está limitada a los métodos, sino que puede utilizarse también en procedimientos y funciones "normales".

SEMEJANZAS Y DIFERENCIAS

El mecanismo de métodos virtuales tiene rasgos en común con el de sobrecarga de métodos. En ambos casos, el programador que utiliza el método especifica un nombre que corresponde a varias implementaciones posibles, y el lenguaje debe decidir cuál de ellas es la más apropiada. La diferencia: en el caso de los métodos virtuales, la decisión se toma en tiempo de ejecución, basándose en el tipo del objeto, mientras que en la sobrecarga, la decisión se toma en tiempo de compilación, tomando como base los tipos de los parámetros.

Público, protegido, privado...

En el capítulo anterior vimos que un objeto podía clasificar sus métodos y atributos en secciones, para controlar el acceso a los mismos. Las dos secciones que conocemos son **public** y **private**; la primera otorga acceso sin restricciones al contenido desde cualquier unidad, mientras que la segunda restringe el acceso al interior de la unidad donde se declara el objeto. Ahora, con la introducción de la herencia, aparece una nueva sección: **protected**.

Para explicar mejor el significado de esta sección, introduciré tres *roles* que puede asumir un programador con respecto a una clase:

- Como *implementador*: Es cuando se define la clase y se implementan sus métodos.
- Como *cliente* o *usuario*: Cuando, fuera de la implementación de los métodos de la clase, se crean objetos pertenecientes a la misma y se trabaja con sus métodos y atributos.
- Como *heredero*: Cuando un programador declara una nueva clase derivada de la clase original e implementa los nuevos métodos o redefine métodos ya existentes.

Hasta el momento, hemos considerado solamente el primer y el segundo rol. Como implementadores, somos los autores de la clase, y tenemos acceso a cualquier sección

definida dentro de ella. Como usuarios o clientes, el lenguaje nos protege para que no tengamos que conocer y dominar todos los recursos de la misma. Por lo tanto, se nos limita la visibilidad a los recursos declarados en las secciones **public**. En el tercer papel, necesitamos un poco más de información acerca de la implementación de la clase, por lo cual Delphi introduce la sección **protected**: los recursos declarados en esta sección son accesibles tanto para el implementador como para los herederos, pero no para los clientes.

Métodos abstractos

¿Cómo chillan los animales? Depende, según el animal que sea. ¿Qué implementación le damos entonces al método *Chillar* de la clase base *TAnimal*? Antes hemos definido para este método una implementación vacía. Pero Delphi nos ofrece una alternativa: declarar el método *abstracto*, mediante la palabra reservada **abstract**:

```
type
   TAnimal = class
      // …
      procedure Chillar(Duracion: Word); virtual; abstract;
   end;
```

Un método puede declararse abstracto solamente si es también un método virtual. Cuando intentamos ejecutar un método abstracto, se produce una excepción. Por lo tanto, los descendientes de una clase con un método abstracto deben redefinir este método y darle una implementación.

Los animales no existen en realidad: existen perros, gatos, vacas y, posiblemente, marcianos; realmente, un animal es un concepto *abstracto*. En una jerarquía de clases puede suceder algo similar, cuando algunas de las clases del árbol de herencia se utilizan solamente como base para la derivación de clase más concretas: *TAnimal* se utiliza para derivar las clases *TPerro*, *TGato* y *TVaca*, y no tiene sentido crear un objeto de tipo *TAnimal*:

```
var
   Animal: TAnimal;
begin
   Animal := TAnimal.Create(Nombre); // ¡Esto no tiene sentido!
   // …
end;
```

Delphi no ofrece mecanismos para "marcar" una clase como abstracta, y prohibir la creación de instancias de la misma. C++ sí permite especificar estas restricciones, pero esto se debe a que no permite *referencias de clases* ni *constructores virtuales*, recursos que estudiaremos a continuación.

Referencias de clase

Una noche tormentosa, mientras estudiaba un viejo libro sobre artes innominables, alguien tocó tres veces en la puerta entreabierta de mi estudio y entró sin esperar mi permiso. Era un personaje extraño, con la cara cubierta por la sombra de una gran gorra de béisbol y un fuerte olor a azufre en la bufanda de cierto club de fútbol que envolvía su cuello. Se presentó con un nombre que he olvidado, y dijo ser un "Programador No Tradicional". Abrió un ordenador portátil, con una copia al parecer sin registrar de Delphi, y cargó una rara aplicación. El proyecto contenía un único formulario con un botón cuyo título sonaba a invitación: "¡Púlsame!". El código asociado era algo así:

```
procedure TForm1.Button1Click(Sender: TObject);
var
   Ed: TEdit;
   Cb: TComboBox;
begin
   if Random(2) = 0 then
   begin
      Ed := TEdit.Create(Self);
      Ed.Left := Random(ClientWidth - Ed.Width);
      Ed.Top := Random(ClientHeight - Ed.Height);
      Ed.Parent := Self;
   end
   else
   begin
      Cb := TComboBox.Create(Self);
      Cb.Left := Random(ClientWidth - Cb.Width);
      Cb.Top := Random(ClientHeight - Cb.Height);
      Cb.Parent := Self;
   end;
end;
```

El método, aparentemente, creaba aleatoriamente cuadros de edición y combos, y los repartía desordenadamente por la superficie de la ficha. La última asignación de cada rama, tenía el propósito de hacer visible el objeto recién creado. "Mal código" - le dije al intruso, cuando desvié la mirada del monitor. Una mueca con pretensiones de sonrisa desfiguró lo poco que se veía de su rostro: "Ya lo sé, enano". Acto seguido comenzó a teclear pausadamente y cambió el código anterior por este otro:

```
procedure TForm1.CrearControl(ACtrl: TWinControl);
begin
   ACtrl.Left := Random(ClientWidth - ACtrl.Width);
   ACtrl.Top := Random(ClientHeight - ACtrl.Height);
   ACtrl.Parent := Self;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
   if Random(2) = 0 then
      CrearControl(TEdit.Create(Self))
```

```
    else
      CrearControl(TComboBox.Create(Self));
  end;
```

Tuve que admitir que ahora el método tenía mejor pinta. El personaje había reconocido que tanto el editor como el combo tenían un antepasado común, *TWinControl*, y que las propiedades que se asignaban por separado en las dos ramas de la instrucción **if**, estaban definidas desde esa clase. Por lo tanto, había extraído el "factor común" y lo había encapsulado en el método *CrearControl*. Pero mi visitante no había terminado: "Ingenioso, pero nada extraordinario. Hasta aquí nos deja llegar *tu* polimorfismo. Más allá de esto, ¿hasta dónde *puedes* llegar hoy?" Y el vaho a azufre aumentó su intensidad.

No soporto que me llamen gallina. Lo que este individuo quería era una versión de *CrearControl* que trabajase con el *tipo* del nuevo control, no con el objeto ya creado. El problema se reducía entonces a suministrar un parámetro al método que pudiera contener la clase, no un objeto de la clase. Cuando un parámetro o variable se declara de tipo *TWinControl*, los valores que toma son objetos de esta clase o de clases derivadas. Y lo que queremos es un parámetro cuyos valores puedan ser *TButton*, *TEdit* y *TCheckBox*, aunque no *TPrinter*; los tres primeros valores de clases descienden de *TWinControl*, mientras que el último no.

Delphi añade al repertorio de recursos de la Programación Orientada a Objetos los tipos llamados *referencias de clases*. El tipo necesario en el ejemplo anterior ya está declarado en la VCL del siguiente modo:

```
type
  TWinControlClass = class of TWinControl;
```

Otras referencias de clase definidas en la VCL son las siguientes:

```
type
  TClass = class of TObject; // La referencia de clase más general
  TComponentClass = class of TComponent;
  TControlClass = class of TControl;
  TFormClass = class of TForm;
```

Físicamente, las variables de este tipo almacenan punteros a Tablas de Métodos Virtuales; como hemos visto, estas tablas identifican unívocamente a las clases. A una variable cuyo tipo es una referencia de clase, se le puede asignar tipos de clases derivadas de la clase inicial, incluyendo a esta misma:

```
var
  WC: TWinControlClass;
begin
  WC := TButton;      // Bien: TButton desciende de TWinControl
  WC := TEdit;        // También correcto
  WC := TWinControl; // Correcto, evidentemente
```

```
   WC := TGato;       // Incorrecto: el compilador protesta
   // …
end;
```

Cuando tenemos una variable o un parámetro que es una referencia de clase, podemos utilizarlo en cualquier instrucción donde sea correcto utilizar la clase base de la definición. El único contexto en que hemos utilizado el tipo de la clase, hasta el momento, es durante la llamada a los constructores:

```
var
   WC: TWinControlClass;
   W: TWinControl;
begin
   WC := TButton;
   W := WC.Create(Self);
   // Equivalente a:
   // W := TButton.Create(Self);
   // …
end;
```

Por lo tanto, gané mi apuesta con el desconocido escribiendo el siguiente método:

```
procedure TForm1.CrearControl(ACtrlClass: TWinControlClass);
begin
   with ACtrlClass.Create(Self) do
   begin
      Left := Random(Self.ClientWidth - Width);
      Top := Random(Self.ClientHeight - Height);
      Parent := Self;
   end;
end;

procedure TForm1.Button1Click(Sender: TObject);
const
   MAX = 4;
   OfertaDelMes: array [0..MAX-1] of TWinControlClass
      = (TEdit, TComboBox, TCheckBox, TRadioButton);
begin
   CrearControl(OfertaDelMes[Random(MAX)]);
end;
```

Constructores virtuales

Para que el ejemplo de la sección anterior funcionase, me hizo falta aprovechar un detalle acerca de los constructores de componentes que no mencioné: para los componentes de Delphi, derivados de la clase *TComponent*, el constructor *Create* es un *constructor virtual*. Delphi permite tanto los constructores virtuales como los constructores estáticos, que es el tipo de constructor frecuentemente encontrado en otros lenguajes de programación (léase C++).

Realmente, los constructores virtuales solamente tienen sentido cuando el lenguaje permite referencias de clases. La razón de la existencia de constructores virtuales es la misma que la de la existencia de métodos virtuales: una referencia de clase puede contener un puntero a la VMT de la clase declarada estáticamente, o un puntero a una VMT de una clase derivada. Por lo tanto, se produce la misma dualidad. Si *WC* pertenece al tipo *TWinControlClass*, pero contiene una clase diferente, derivada de *TWinControl*. Cuando se ejecute la instrucción *WC.Create(Self)*, ¿qué constructor se invoca, el de *TWinControl* o el perteneciente a la clase real asociada? Si el constructor *Create* de *TWinControl* es virtual, se llama al constructor correcto, pero si no es virtual se llama al constructor original.

La clase *TComponent* introduce un constructor *Create* virtual. Su declaración es la siguiente:

```
constructor TComponent.Create(AOwner: TComponent); virtual;
```

Del mismo modo que con los métodos virtuales, al redefinir un constructor virtual debemos utilizar los mismos parámetros que en el ancestro. De aquí que el constructor de todos los componentes de Delphi tenga, precisamente, esos parámetros.

La VMT de una clase almacena el puntero a la implementación de sus constructores virtuales, igual que si fueran métodos virtuales "normales".

Información de tipos en tiempo de ejecución

Una vez que las referencias de clase se convierten en ciudadanos normales en nuestro lenguaje de programación, es fácil incluir mecanismos para obtener información acerca de las clases de los objetos en tiempo de ejecución. El operador **is**, por ejemplo, permite conocer si un objeto pertenece a cierta clase, o si su clase se deriva por herencia de la misma:

```
if ActiveControl is TButton then
   // …
```

El operador **as** permite realizar una conversión de tipo en tiempo de ejecución con verificación previa. Este operador, en contraste con **is**, no devuelve un valor lógico, sino un puntero a un objeto del tipo especificado en el segundo parámetro. Se puede utilizar de cualquiera de las siguientes formas:

```
(ActiveControl as TButton).Click;
// …
```

```
with Sender as TEdit do
begin
   Text := IntToStr(Date);
   if Visible then SetFocus;
end;
```

El operador **as** se puede descomponer en instrucciones más simples. La primera instrucción del ejemplo anterior es equivalente a la siguiente secuencia de instrucciones:

```
if ActiveControl is TButton then
   TButton(ActiveControl).Click
else
   raise EInvalidCast.Create('Invalid type cast');
```

La última instrucción lanza una *excepción*; las excepciones se tratan más adelante. Esta instrucción aborta la ejecución de las instrucciones que deberían seguir, en orden lógico, a la actual.

Si alguien me preguntase por las cosas que me molestan, incluiría sin vacilar a los programadores que escriben cosas como la siguiente:

```
if Objeto is TYoQueSe then
   with Objeto as TYoQueSe do
   begin
      // …
   end;
```

Los programas escritos en este estilo aumentan las ventas de hardware. Mi estimado programador, ¿qué necesidad hay de utilizar **as** si ya sabemos con toda certeza que *Objeto* pertenece a la clase *TYoQueSe*? Cambie a las siguientes instrucciones, y sus usuarios se lo agradecerán:

```
if Objeto is TYoQueSe then
   with TYoQueSe(Objeto) do
```

Para obtener la referencia de la clase a la cual pertenece un objeto, se utiliza la función *ClassType*, definida por la clase *TObject*:

```
function TObject.ClassType: TClass;
```

La siguiente función devuelve un componente del mismo tipo que el pasado como parámetro:

```
function Clonar(Component: TComponent): TComponent;
begin
   Result := TComponentClass(Component.ClassType).Create(
      Component.AOwner);
end;
```

Hay que tener cuidado de no confundir el resultado de estas instrucciones:

```
if ActiveControl.ClassType = TButton then …
// Pregunta si el control activo es un botón
if ActiveControl is TButton then …
// También es cierto si su clase desciende de TButton
```

Métodos de clase

Otro recurso del lenguaje que utiliza las referencias a clase son los *métodos de clase*. Un método de clase se programa como un método "normal", pero con la palabra reservada **class** al principio de la declaración.

```
type
   TBurocrata = class(TPrimate)
      // …
      constructor Create(const ANombre, ACargo: string);
      destructor Destroy; override;
      class function Cantidad: Integer;
   end;
```

La función *Cantidad* debe contar la cantidad de burócratas existentes en el sistema. Esta función no debe diseñarse como un método más de la clase, porque para conocer cuantos burócratas contaminan nuestro entorno tendríamos que tener uno de estos objetos a mano. Una alternativa es definir una función global, de las de toda la vida. Pero queremos contar, además de los burócratas, los objetos de tipo *TDiaDePrimavera*. Y no quiero volver a los días en que mis funciones se llamaban así:

```
function CantidadBurocratas: Integer;
function CantidadDiasDePrimavera: Integer;
```

Los métodos de clase se aplican a una referencia de clase, en vez de a un objeto de la clase:

```
if TBurocrata.Cantidad > 0 then
   ShowMessage('¡Demasiados!');
```

Si tenemos un objeto también podemos utilizarlo, pues Delphi extrae primeramente su referencia de clase y luego aplica el método de clase indicado:

```
ShowMessage(IntToStr(PedroPerez.Cantidad));
// Equivalente a la instrucción:
ShowMessage(IntToStr(PedroPerez.ClassType.Cantidad));
```

Para implementar la función de clase utilizaré una variable global declarada en la implementación de la unidad donde se define la clase. Esta variable se debe incrementar en cada ejecución del constructor y decrementar cuando se llame al destructor:

```
var
   CantidadInterna: Integer = 0;

constructor TBurocrata.Create;
begin
   inherited Create(ANombre);
   Cargo := ACargo;
   Inc(CantidadInterna);
end;

destructor TBurocrata.Destroy;
begin
   Dec(CantidadInterna);
   inherited Destroy;
end;

class function TBurocrata.Cantidad;
begin
   Result := CantidadInterna;
end;
```

A diferencia de lo que ocurre con los métodos normales, el parámetro predefinido *Self* no se refiere a un objeto de la clase, sino a la referencia de clase misma. Esto quiere decir que dentro de la función *Cantidad* de la clase *TBurocrata*, el parámetro *Self* no es de tipo *TBurocrata*, sino de tipo **class of** *TBurocrata*. Este parámetro no se ha aprovechado en el ejemplo anterior.

Hay toda una lista de métodos de clase disponibles desde la clase *TObject*. Estos son:

```
class function ClassInfo: Pointer;
class function ClassName: ShortString;
class function ClassNameIs(const Nombre: string): Boolean;
class function ClassParent: TClass;
class function InheritsFrom(AClass: TClass): Boolean;
class function InitInstance(Objeto: Pointer): TObject;
class function InstanceSize: LongInt;
class function MethodAddress(const Nombre: ShortString): Pointer;
class function MethodName(Direccion: Pointer): ShortString;
```

Sin embargo, *ClassType* no es método de clase pues solamente puede aplicarse a instancias de objetos, no a referencias de clase.

Capítulo

8

Elementos de programación con Windows

UNO DE LOS PRINCIPALES PROBLEMAS que afrontan los nuevos programadores de Delphi es el comprensible desconocimiento de la arquitectura básica del sistema operativo en que programan. Y digo "comprensible" pues, además de las dificultades técnicas inherentes al sistema, la documentación asociada es voluminosa, oscura y de difícil acceso. Hasta los tratados de alquimia son más legibles.

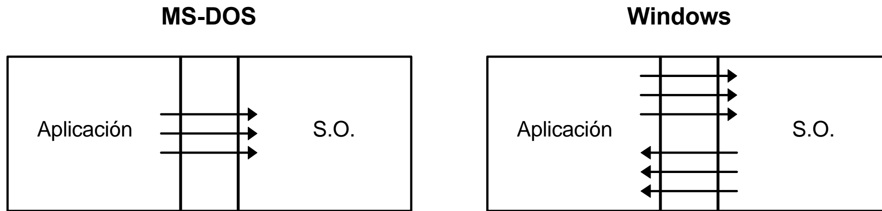
Es imposible cubrir incluso los rasgos generales de la interfaz de programación de aplicaciones y sus conceptos fundamentales en un solo capítulo. No obstante, intentaré mencionar las ideas principales y la terminología básica necesaria para un programador de Delphi. Recomiendo encarecidamente al lector que profundice en estos temas, si es que quiere llegar a dominar verdaderamente la programación para esta familia de sistemas operativos.

Si me necesitas, ¡llámame!

En los buenos viejos tiempos de MS-DOS, cuando cuatro millones y pico de ciclos por segundo era una velocidad vertiginosa y un disco duro de 20 megas un lujo oriental, bastaba con estudiarse la interfaz de la interrupción 21 (que era en realidad la 33 decimal) y un par de trucos con la BIOS para alcanzar el honorable título de *hacker*. Esta interfaz ofrecía servicios para el sistema de archivos, entrada y salida básica, y poco más. ¿Ratón, para qué, si sólo algunos afortunados tienen uno? ¿Gráficos?, ¡bah, por favor, esas son chorradas! ¿Multitarea, animación y sonido, comunicaciones, redes, Internet...? No son horas de andar bebiendo, Ian.

Así que ahora tenemos miles de funciones para aprender, con sus respectivas subfunciones, en vez de los pocos cientos que teníamos en MS-DOS. Pero lo que realmente complica la programación en Windows es el carácter *bidireccional* de la comunicación entre las aplicaciones y el sistema operativo. Antes, la relación de la aplicación (y en consecuencia del programador) con la interfaz del sistema operativo se

reducía a que solamente la primera podía "dictar órdenes" al último: abre tal fichero, muévete al vigésimo segundo byte, lee cinco bloques, cierra el fichero... Estas "órdenes" se producían al ejecutar las funciones o interrupciones de la interfaz de programación. Si el sistema operativo tenía algo que decir, tenía que aprovechar el retorno de las funciones, para protestar por los malos tratos: la protesta tenía lugar en forma síncrona.



Ahora, sin embargo, puedo pedirle a Windows: "oye, dibuja esta ventana", y darme la vuelta, pero lo más probable es que Windows me toque en el hombro y me pregunte insolentemente: "¿me puedes decir cómo?". Con esto quiero decir que el sistema operativo también puede llamar a funciones definidas dentro de la aplicación. Piense, por ejemplo, en cómo Windows nos permite dibujar el interior de una ventana, poniéndose en el papel de los desarrolladores del sistema operativo. Usted deja que una aplicación dibuje en el área interior de la ventana. Estupendo. Ahora el sistema operativo presenta un mensaje al usuario en un cuadro de dialogo que aparece sobre el área de dibujo. Al cerrar el cuadro de diálogo, ¿cómo restauramos el contenido original del rectángulo? Si almacenamos el mapa de bits como regla general, pronto nos veríamos con problemas de memoria; si no me cree, saque cuentas. La solución de Windows es dejar que sea la propia aplicación quien se encargue de estos detalles, y para eso necesita que la función que dibuja, que está dentro de la aplicación, pueda ser ejecutada cuando Windows lo necesite, fuera de la secuencia de ejecución original prevista por el programador.

Las funciones y procedimientos definidas por el programador para ser ejecutadas directamente por el sistema operativo reciben el nombre de *funciones de respuesta* (*callback functions*).

¿Qué es una ventana?

Evidentemente, para un arquitecto y un ladrón una ventana trae asociaciones mentales diferentes. Un usuario de Windows, posiblemente pensará que una ventana es "esa cosa con barra de título, tres iconos en una esquina, que se puede mover, ampliar, reducir o ignorar". Bueno, eso es cierto, nuestro usuario está pensando en una *ventana solapada* (*overlapped window*), pero para Windows muchos otros objetos son también ventanas. Por ejemplo, un simple botón. O una casilla de verificación aislada.

En realidad, una ventana es un objeto interno del sistema operativo, que se responsabiliza del dibujo de una zona rectangular del monitor[11], y que puede recibir mensajes (más adelante veremos qué son) de Windows. Estos objetos pueden ser manejados internamente por el sistema, pero también por el programador. Este último identifica siempre a una ventana mediante un *handle*, que es un valor numérico. El motivo fundamental es que, al residir el objeto dentro del espacio de memoria del sistema operativo, no es conveniente, y en ocasiones es imposible, suministrarle al programador el puntero directo al objeto. El *handle* representa verdaderamente una entrada dentro de una tabla; dado un *handle*, Windows busca en la tabla y encuentra en la entrada asociada el puntero real al objeto. Los *handles* se utilizan, además, como identificación de muchos otros objetos de Windows que no son precisamente ventanas: ficheros, semáforos, procesos, hilos, etc. En Delphi pueden almacenarse genéricamente en variables de tipo *THandle*.

Clases de ventana

¿Por qué entonces el aspecto visual de las ventanas es tan diferente, y por qué tienen comportamientos tan disímiles? La clave es que cada ventana pertenece a una *clase de ventana*, y que esta clase determina globalmente cómo se dibuja y cómo responde a las acciones del usuario y del sistema operativo. Las clases se identifican por medio de cadenas de caracteres. Por ejemplo, la clase predefinida de botones se denomina *BUTTON*. Este nombre de clase es el que hay que suministrar a la función de Windows que crea ventanas.

Las clases de ventanas deben *registrarse* antes de crear objetos pertenecientes a las mismas. Antes, en Windows 16, se utilizaba el método *RegisterClass*; ahora hay un nuevo método, *RegisterClassEx*, que es el que debemos llamar.

```
procedure RegisterClassEx(const WndClass: TWndClassEx);
```

El tipo *TWndClassEx* tiene la declaración que se muestra a continuación:

```
type
   TWndClassEx = packed record
      cbSize: UINT;
      style: UINT;
      lpfnWndProc: TFNWndProc;
      cbClsExtra: Integer;
      cbWndExtra: Integer;
      hInstance: HINST;
      hIcon: HICON;
      hCursor: HCURSOR;
```

[11] No es estrictamente necesario que una ventana sea rectangular. La función *SetWindowRgn*, de Windows 95 y NT, permite que la forma de una ventana sea una elipse, un polígono o cualquier otra figura que nos dicte nuestra imaginación.

```
    hbrBackground: HBRUSH;
    lpszMenuName: PAnsiChar;
    lpszClassName: PAnsiChar;
    hIconSm: HICON;
  end;
```

El nombre de la clase que queremos registrar se asigna al campo *lpszClassName*. He mostrado la declaración de *TWndClassEx* para que el lector pueda hacerse una idea de las características que definen a una clase de ventana: un conjunto de indicadores de estilo (*style*), iconos asociados (*hIcon* y *hIconSm*), el cursor (*hCursor*), un menú por omisión (*lpszMenuName*), etcétera. El parámetro *lpfnWndProc* será explicado más adelante.

Una vez que tenemos una clase de ventana predefinida o registrada por nosotros mismos, utilizamos su nombre para crear ventanas de esa clase con las funciones *CreateWindow* ó *CreateWindowEx*:

```
function CreateWindowEx(dwExStyle: DWORD; lpClassName: PChar;
  lpWindowName: PChar; dwStyle: DWORD;
  X, Y, nWidth, nHeight: Integer;
  hWndParent: HWND; hMenu: HMENU; hInstance: HINST;
  lpParam: Pointer): HWND;
```

En esta función se indica el estilo de ventana (*dwExStyle* y *dwStyle*), el nombre de la clase, el título, posición y tamaño, de qué ventana depende (*hWndParent*), y la función devuelve el *handle* de la ventana creada.

Recuerde, sin embargo, que en contadas ocasiones tendrá que llamar a alguna de estas funciones desde Delphi, pues la VCL se encarga automáticamente del registro de clases y de la creación de ventanas.

Eventos o, más bien, mensajes

El usuario se comunica con el sistema operativo mediante los periféricos de entrada de datos, teclado y ratón fundamentalmente; en un futuro inmediato, el dispositivo de reconocimiento de voz. Y antes habíamos dicho que el sistema operativo también se comunicaba con la aplicación para pedirle acciones o notificarle acerca de situaciones: "¿cómo dibujo el interior de esta ventana?", "van a apagar el sistema, ¿qué te parece?"... Uno de los puntos de partida de los sistemas controlados por eventos es el reconocer las semejanzas entre ambos tipos de comunicación, para unificarlos en un mismo mecanismo. En una comunicación, sea del tipo usuario/sistema ó sistema/aplicación, siempre se mueve cierta información acerca de cierto suceso; a estos sucesos se les conoce teóricamente con el nombre de *eventos*, aunque en el caso concreto de Windows se le aplica el nombre de *mensajes* (*messages*).

Los mensajes pueden clasificarse de varias formas. La principal es dividirlos en mensajes *internos* y *externos*; en la primera categoría entrarían los que son provocados por el usuario, dejando para la segunda los que surgen de la comunicación del sistema operativo con la aplicación. Hay que tener en cuenta, sin embargo, que casi todos los mensajes se producen, en forma más o menos indirecta, como resultado de una acción del usuario. Por ejemplo, la notificación del cierre del sistema tuvo su causa original en la pulsación de la combinación de teclas ALT+F4 por parte del individuo sentado frente al ordenador. Uno de los pocos casos de mensajes en los que no interviene un operador humano son los mensajes del temporizador. Por otra parte, ningún mensaje pasa directamente del usuario a una aplicación, pues tienen primero que ser procesados y transformados por el núcleo de Windows. Otra clasificación útil de los mensajes se basa en su modo de propagación, es decir, en el algoritmo que utiliza el sistema para decidir qué objeto da respuesta a cierto mensaje. Por ejemplo, los eventos de ratón se consideran *posicionales*, pues el algoritmo de distribución se fija principalmente en las coordenadas del evento y de los objetos existentes. Los eventos de teclado se clasifican como eventos de *foco* (*focused events*), pues su distribución depende de la cadena de objetos activos en el escritorio.

La información más importante de un mensaje es su código, que es un valor entero. Windows define constantes simbólicas que representan estos códigos numéricos: *WM_PAINT*, *WM_DESTROY*, *EM_UNDO*… La mayor parte de los identificadores de mensajes comienzan con las iniciales *WM*, *window message*, pero algunos mensajes específicos de ciertos controles comienzan con otros prefijos. Por ejemplo, *EM* se utiliza en los mensajes que se envían para controlar los cuadros de edición.

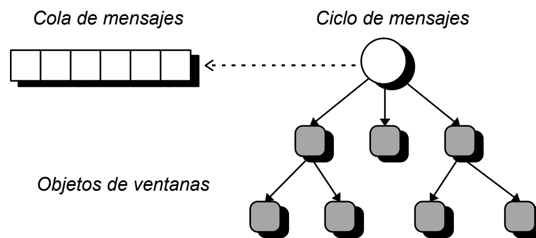
Pero un único valor numérico contiene muy poca información. Cada mensaje lleva además otros dos valores numéricos, conocidos por razones históricas como los parámetros *wParam* y *lParam*. En Windows 16, el primero era un entero de 16 bits, mientras que el segundo tenía 32 bits; en Windows 32 ambos son enteros largos. Estos parámetros pueden contener punteros a estructuras disfrazados de valores enteros, y también *handles* de objetos del sistema.

El ciclo de mensajes y la cola de mensajes

Para los que gustan de resumir un concepto en una frase, una de las claves de los SCE es la *colección centralizada de eventos*. En este tipo de sistemas, los eventos se leen en un único procedimiento; si el sistema está basado en objetos, es un solo objeto el encargado de la lectura de eventos. El proceso mediante el cual se *recogen* y se *distribuyen* los mensajes para darles tratamiento se conoce como *ciclo o bucle de mensajes* (*message loop*).

¿De dónde se leen los mensajes? Resulta que Windows mantiene para cada aplicación una estructura de datos llamada *cola de mensajes* (*message queue*), donde se van alma-

cenando los eventos internos y los eventos de usuario que corresponden a la aplicación. El sistema operativo es el responsable de insertar mensajes en esta cola, y es también quien decide a qué aplicación va a parar cada mensaje. Estrictamente hablando, la cola de mensajes no se ajusta a la definición matemática de cola: una estructura lineal en la que se inserta por un extremo y se extrae por el otro. Algunos mensajes de Windows entran en la desdichada categoría de *mensajes de baja prioridad*, como es el caso de *WM_PAINT* (redibujar una ventana) y *WM_TIMER* (notificaciones del temporizador). La explicación es que es más eficiente, por ejemplo, postergar las operaciones de dibujo mientras existan cosas más importantes que hacer. Cualquier mensaje que exista en la cola puede arrebatarle el turno a *WM_PAINT*. Otra característica es que no tiene sentido acumular mensajes de baja prioridad: si quedan dos mensajes de dibujo en la cola, se mezclan en una sola operación.



En versiones anteriores de Windows y de Windows NT, la cola de mensajes de una aplicación tenía un tamaño fijo, y en ocasiones debía ampliarse manualmente para admitir un número mayor de mensajes pendientes; uno de los casos en que esto era necesario era cuando la aplicación debía trabajar con OLE. En las versiones actuales de estos sistemas operativos, las colas de mensajes se implementan mediante listas enlazadas y crecen por demanda, quedando obsoletas las rutinas de reasignación de tamaño de las colas.

Pregunta: ¿Por qué es tan importante que la recolección de mensajes se centralice?
Respuesta: Porque es lo que permite que los distintos objetos dentro de una aplicación trabajen de forma concurrente.

Todos los programadores, en algún oscuro momento de nuestras vidas, hemos intentado programar un editor de texto para un sistema operativo orientado a caracteres, estilo MS-DOS ó UNIX. Posiblemente, la forma en que nos planteamos el editor fue mediante un procedimiento:

```
procedure Editar( … parámetros y más parámetros … );
```

El código de *Editar* sería parecido al siguiente:

```
Tecla := LeerTeclado;
while Tecla <> #27 do
begin
```

```
    case Tecla of
      #8: BorrarCaracter;
      ' '..#255: InsertarCaracter(Tecla);
      // … otros comandos …
    end;
    Tecla := LeerTeclado;
  end;
```

El programa principal ejecuta este procedimiento y todo funciona estupendamente bien. ¿Pero qué pasa si queremos editar simultáneamente en dos ventanas? ¿Cómo interrumpimos "temporalmente" el ciclo de lectura y procesamiento de la primera ventana para ejecutar el de la segunda? Evidentemente, la culpa es del bucle **while**, y la solución consiste en realizar un único ciclo de mensajes por aplicación.

Un ciclo de mensajes sencillo en Windows puede ser como el que mostramos a continuación:

```
var
  Msg: TMsg;
begin
  // …
  while GetMessage(Msg, 0, 0, 0) do
  begin
    TranslateMessage(Msg);
    DispatchMessage(Msg);
  end;
  // …
end;
```

GetMessage espera a que haya un mensaje en la cola de mensajes, y devuelve la información del mismo en el registro que se pasa en el primer parámetro; los otros dos parámetros sirven para pedir que se lean mensajes correspondientes a determinada ventana, o cuyo valor numérico esté dentro de cierto rango. *GetMessage* devuelve el valor *False* solamente cuando encuentra el mensaje especial cuyo código es *WM_QUIT*; el resultado de éste es terminar la aplicación. Cuando ya tenemos un mensaje, *TranslateMessage* es la primera función en procesarlo, y se encarga de traducir secuencias de pulsaciones de teclas en caracteres ANSI. Por ejemplo: pulsar la tecla de mayúsculas y pulsar, a continuación, la tecla del 4 produce el signo del dólar. Finalmente, *DispatchMessage* hace que Windows envíe el mensaje a la ventana que debe procesarlo.

En realidad, un ciclo de mensajes verdadero puede ser más complicado, si se tienen en cuenta las aplicaciones MDI, los aceleradores de teclado y la posibilidad de efectuar acciones durante los intervalos de tiempo en que el usuario no interactúa con la aplicación. Windows pudiera ofrecer una función que realizara un ciclo de mensajes estándar, pero deja en nuestras manos las piezas para que montemos esta parte tan delicada de la estructura de nuestro programa.

El ciclo de mensajes de una aplicación en Delphi se ejecuta dentro del método *Run* de la clase *TApplication*, que es llamado desde el fichero de proyecto de la aplicación.

Ejecución modal y no modal

Cualquier usuario de Windows nota inmediatamente las diferentes formas de interacción con una ventana. Habrá observado que, dentro de una aplicación, algunas ventanas permiten seleccionar a sus compañeras, mientras siguen activas en segundo plano. Otras ventanas, casi siempre cuadros de diálogo, son tan prepotentes que no aceptan pasar al segundo plano; hay que darles una respuesta, usualmente pulsando un botón, para cerrarlas y continuar trabajando con el resto de la aplicación.

Las instrucciones para mostrar una ventana modal son completamente distintas de la secuencia de pasos necesarios para añadir una ventana no modal a una aplicación. Si lo que quiere mostrarse modalmente es un cuadro de diálogo, usualmente definido a partir de un recurso, la función *DialogBox* de Windows combina en un solo paso la creación, ejecución y destrucción de la ventana. Si, en cambio, la ventana que pretendemos mostrar de forma modal no es un cuadro de diálogo, tenemos que, después de crear la ventana, mostrarla normalmente (la función *ShowWindow* del API) y ejecutar un ciclo de mensajes que propague los mensajes *a partir de la ventana activa*. Este es, en el fondo, el quid de la ejecución modal: mover el *ápex*, que es el nodo del árbol de objetos que ejecuta el ciclo de mensajes, reduciendo la propagación de mensajes al subárbol que depende del nuevo nodo.

Delphi simplifica estas tareas para beneficio nuestro. Para mostrar una ventana en forma no modal, se puede utilizar el método *Show* de la clase *TForm*, o asignar *True* a la propiedad *Visible* de esta clase. Para ocultarla y destruir el objeto correspondiente de Windows se utiliza *Hide*, o se asigna *False* a *Visible*. La ejecución modal se alcanza mediante el método *ShowModal*; como condición previa a la aplicación del método, la ventana tiene que tener *Visible* igual a *False*. Es interesante saber que Delphi no utiliza los cuadros de diálogos de Windows, aquellos que se crean y ejecutan con *DialogBox*, sino que los simula mediante ventanas "normales". Esta simulación tiene que ver, sobre todo, con el manejo del teclado por los controles de edición.

El procedimiento de ventana

De todas las características posibles de una clase ventana, la más importante es el *procedimiento de ventana* (*window procedure*) asociado; este no es más que un caso importante de función de respuesta. El procedimiento de ventana se indica cuando registramos una clase de ventana, y corresponde al parámetro *lpfnWndProc* del tipo de registro *TWndClass*, que vimos anteriormente.

Cuando Windows ha decidido que un mensaje tiene que ser manejado por determinada ventana, ejecuta el procedimiento de ventana que corresponde a la misma, pasando como parámetros el identificador de ventana, el código de mensaje y los parámetros del mensaje. El prototipo de un procedimiento de ventana, en notación Pascal y para 32 bits, debe ser como el siguiente:

```
function ProcVentana (Window: HWND;
   Message, WParam, LParam: LongInt): LongInt; stdcall;
```

El valor de retorno de la función sirve para que la aplicación pueda comunicarse con el sistema operativo. Ciertos mensajes requieren que el objeto que los recibe devuelva cierta información, o sencillamente que exprese su opinión sobre la acción que se realizará a continuación por omisión.

Un procedimiento de ventana típico consiste en una instrucción de selección, para discriminar los tipos de mensajes por su código, e interceptar aquellos que nos interesan. Por ejemplo:

```
function MiProcVentana(Window: HWND;
   Message, WParam, LParam: LongInt): LongInt; stdcall;
begin
   case Message of
      WM_DESTROY:
         begin
            PostQuitMessage(0);
            Result := 0;
         end;
   else
      Result := DefWindowProc(Window, Message, WParam, LParam);
   end;
end;
```

El único mensaje al que prestamos atención en el procedimiento anterior es a *WM_DESTROY*, que es el aviso que se envía a una ventana cuando termina su existencia. Nuestra ventana es egoísta, y en respuesta a esta notificación ejecuta la llamada a *PostQuitMessage*, que coloca una bomba en la cola de mensajes de la aplicación: el mensaje fatal *WM_QUIT*. El resto de los mensajes nos trae sin cuidado, y dejamos que sea la función de Windows *DefWindowProc* quien se ocupe de ellos. Este es el comportamiento habitual de las *ventanas principales* (*main window*): si las cerramos, termina automáticamente la ejecución del programa.

Interceptando mensajes en Delphi

La programación en Windows utilizando directamente las funciones del API es demasiado complicada y laboriosa para aplicaciones reales, sobre todo si existen alternativas. Todas las bibliotecas de desarrollo para Windows, como OWL, MFC o la VCL de Delphi, ofrecen técnicas más sencillas para automatizar el tratamiento de

mensajes. En particular, para interceptar mensajes de Windows en una ventana de Delphi, se utilizan los procedimientos declarados mediante la directiva **message**:

```
type
  TForm1 = class(TForm)
  private
    procedure WMEraseBkgnd(var Msg: TWMEraseBkgnd);
      message WM_ERASEBKGND;
    procedure WMSize(var Msg: TWMSize); message WM_SIZE;
    procedure WMPaint(var Msg: TMessage); message WM_PAINT;
  public
  end;
```

La directiva **message** indica qué código de mensaje se asocia a cada procedimiento. El formulario del ejemplo anterior intercepta los mensajes de Windows *WM_PAINT*, que dibuja el interior de una ventana, *WM_ERASEBKGND*, que sirve para evitar que Windows llene automáticamente el fondo de la ventana, y *WM_SIZE*, que avisa cuando la ventana cambia de tamaño. Todo los procedimientos de mensajes necesitan un solo parámetro por referencia, cuyo tipo no se verifica. Esto permite declarar para cada mensaje el tipo de parámetro más adecuado a la información que contiene. Por ejemplo, esta es la declaración de *TWMSize*:

```
type
  TWMSize = record
    Msg: Cardinal;
    SizeType: Longint;
    Width: Word;
    Height: Word;
    Result: Longint;
  end;
```

Pero, con el mismo éxito, hubiéramos podido definir todos los procedimientos con el tipo genérico de mensaje *TMessage*, cuya declaración es:

```
type
  TMessage = record
    Msg: Cardinal;
    case Integer of
      0: (
        WParam: Longint;
        LParam: Longint;
        Result: Longint);
      1: (
        WParamLo: Word;
        WParamHi: Word;
        LParamLo: Word;
        LParamHi: Word;
        ResultLo: Word;
        ResultHi: Word);
  end;
```

El objetivo de los mensajes interceptados en el formulario anterior es dibujar un gradiente de colores como fondo de la ventana. De los procedimientos creados, el más sencillo corresponde al dibujo del fondo:

```
procedure TForm1.WMEraseBkgnd(var Msg: TWMEraseBkgnd);
begin
   Msg.Result := 1;
end;
```

Lo único que se realiza durante este mensaje es decirle a Windows que no se preocupe por dibujar el fondo, que ya nos encargaremos nosotros. El paso siguiente es interceptar la notificación de cambio de tamaño:

```
procedure TForm1.WMSize(var Msg: TWMSize);
begin
   inherited;
   InvalidateRect(Handle, nil, False);
end;
```

Como puede ser que la clase *TForm* o cualquier otro ancestro de *TForm1* tenga un manejador para este mensaje (y de hecho, lo tiene), ejecutamos este posible manejador mediante la primera instrucción. Observe que, a diferencia del uso de **inherited** en la redefinición de métodos virtuales, no se especifica el nombre del método heredado ni se le pasa el parámetro del mensaje. El quid consiste en que no sabemos a ciencia cierta si tal método existe, cómo se llama y de qué tipo se ha definido su parámetro; todo lo anterior se deja a la sabiduría de Delphi. La segunda instrucción simplemente obliga a Windows a redibujar el contenido de la ventana cuando tenga un momento libre.

Por último, este es el tratamiento que se da al mensaje *WM_PAINT*:

```
procedure TForm1.WMPaint(var Msg: TWMPaint);
var
   DC, Brocha: THandle;
   PS: TPaintStruct;
   R: TRect;
   I: Integer;
begin
   R.Left := 0;
   R.Right := ClientWidth;
   DC := BeginPaint(Handle, PS);
   for I := 0 to 31 do
   begin
      R.Top := I * ClientHeight div 32;
      R.Bottom := (I + 1) * ClientHeight div 32;
      Brocha := CreateSolidBrush(RGB(0, 0,
         (I + 1) * 128 div 32 + 127));
      Windows.FillRect(DC, R, Brocha);
      DeleteObject(Brocha);
   end;
   EndPaint(Handle, PS);
end;
```

Intencionalmente, he realizado todo el dibujo ejecutando funciones de bajo nivel del API; así el lector tendrá una idea del "sabor" de la programación con el API de Windows. Además, los mensajes anteriores, con excepción de *WM_ERASE-BKGND*, podían haberse tenido en cuenta más fácilmente utilizando eventos de alto nivel de Delphi: *OnPaint* y *OnResize*.

También podemos enviar mensajes

Los implementadores de Windows decidieron, al parecer, complicar deliberadamente la programación en este sistema. De acuerdo a lo que hemos explicado hasta el momento, el lector puede realizar la siguiente clasificación mental: si quiero realizar llamadas a Windows, utilizo funciones; si Windows quiere llamar a mi programa utilizará mensajes. Y es un cuadro correcto, hasta cierto punto, pero falla en un pequeño detalle: algunas llamadas a Windows se efectúan mediante mensajes.

Por ejemplo, tenemos un editor de textos, un *Memo* de Delphi, y queremos deshacer los últimos cambios realizados en el mismo. El programador echa mano de la referencia de Windows y busca frenéticamente en la lista de funciones alguna de las siguientes combinaciones *MemoUndo*, *UndoMemo*, *EditUndoRedo*, ... ¡nada! La forma de hacerlo es la siguiente:

```
SendMessage(HandleDelMemo, EM_UNDO, 0, 0);
// Memo1.Perform(EM_UNDO, 0, 0), si utilizamos la VCL
```

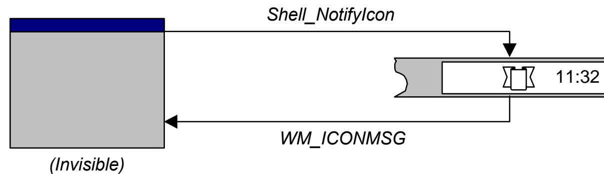
Para enviar mensajes contamos con dos funciones del API: *SendMessage* y *PostMessage*, y todo depende de la prisa que tengamos. *SendMessage* llama directamente al procedimiento de ventana de la ventana sobre la cual se aplica, con los parámetros adecuados. En cambio, *PostMessage* deposita el mensaje en la cola de la aplicación, respetando el orden de los mensajes pendientes. El método *PostQuitMessage*, que vimos en la sección anterior, equivale precisamente a una llamada a *PostMessage* con el mensaje *WM_QUIT*.

Aplicaciones en la bandeja de iconos

Quedamos entonces en que nuestras aplicaciones se comunican con Windows ejecutando funciones del API y, en algunos casos, enviando mensajes a los objetos internos de Windows. Y que Windows se comunica con nosotros, en la mayoría de los casos, enviándonos mensajes que deben ser manejados por alguna de nuestras ventanas. ¿Correcto?

Bien, entonces vamos a mostrar una pequeña aplicación de estos conceptos. Vamos a crear un programa que no tenga ventana principal visible, sino que la interacción con

el mismo se produzca a través de un icono en la barra de tareas, en la bandeja de iconos que aparece a la derecha de la misma. Nos comunicaremos con la bandeja de iconos mediante una función del API, llamada *Shell_NotifyIcon*, que se encuentra declarada en la unidad *ShellApi* de Delphi. Estaremos, además, pendientes de lo que le suceda a nuestro icono mediante un mensaje que este nos enviará cuando sea necesario. ¿Qué mensaje será éste? Tenemos que decidirlo nosotros mismos. Y será nuestra ventana principal la encargada de interceptar y dar tratamiento a este mensaje ... un momento, ¿no habíamos quedado en que no tendríamos ventana principal visible? Sí, pero que no esté visible no quiere decir que no exista.



Creamos un nueva aplicación, vamos al código fuente de la unidad principal y, antes de la declaración del tipo *TForm1*, justo debajo de la cláusula **uses** de la interfaz, declaramos la siguiente constante que corresponderá al código del mensaje que debe enviarnos el icono:

```
const
   WM_ICONMSG = WM_USER + 2000;
```

A la constante predefinida *WM_USER*, que marca el inicio del rango de los mensajes de usuarios, le he añadido una constante arbitraria. ¿Por qué 2000? Porque si logro llegar vivo a ese año, me voy a desternillar de risa viendo como cascan las aplicaciones escritas en COBOL que todavía queden por ahí.

Traiga un componente *TImage* al formulario, y cargue en el mismo el icono que desea que aparezca en la bandeja. Ahora debemos simplificar un poco la llamada a la función *Shell_NotifyIcon*. Esta función, como corresponde a una función del API, tiene una interfaz complicada y desagradable, así que declaramos el siguiente método en la parte privada de la clase *TForm1* y lo implementamos más adelante:

```
procedure TForm1.NotifyIcon(Value: Word; const Tip: string);
var
   Data: TNotifyIconData;
begin
   Data.cbSize := SizeOf(Data);
   Data.uId := 0;
   Data.uFlags := NIF_MESSAGE or NIF_ICON or NIF_TIP;
   Data.Wnd := Self.Handle;
   Data.uCallbackMessage := WM_ICONMSG;
   Data.hIcon := Image1.Picture.Icon.Handle;
   StrPLCopy(Data.szTip, Tip, SizeOf(Data.szTip));
```

```
    Shell_NotifyIcon(Value, @Data);
end;
```

Al parecer, la idea es empaquetar unos cuantos datos en un registro de tipo *TNotify-IconData*, pasarlo a la función del API y decirle qué queremos hacer con estos datos. El primer parámetro de *Shell_NotifyIcon*, el que hemos denominado *Value* puede ser uno de los siguientes:

- *NIM_ADD*: Crear un icono
- *NIM_DELETE*: Eliminar un icono
- *NIM_MODIFY*: Modificar cualquiera de sus datos (icono, ayuda)

En cuanto a los campos de la estructura, los más importantes ahora son *Wnd* y *uCall-backMessage*. El segundo es el código del mensaje del icono, mientras que el primero es el *handle* de la ventana a la que debe enviarse el mensaje: a nuestra ventana principal, por descontado.

Ya estamos listos para crear el icono y para destruirlo. Escribimos los siguientes manejadores para los eventos *OnCreate* y *OnClose* del formulario:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
   Application.ShowMainForm := False;
   ShowWindow(Application.Handle, SW_HIDE);
   NotifyIcon(NIM_ADD, Application.Title);
end;

procedure TForm1.FormClose(Sender: TObject);
begin
   NotifyIcon(NIM_DELETE, '');
end;
```

Durante la creación de la ventana principal hemos instruido a la aplicación para que no la muestre, más adelante. La segunda instrucción puede parecernos algo extraña. El objeto *Application* de Delphi es realmente una ventana, escondida para el programador; desde el punto de vista del usuario es la ventana que aparece solamente en la barra de tareas. Tenemos que esconder también a esta ventana, y nos vemos obligados a utilizar la función del API *ShowWindow*, aplicada sobre el *handle* de dicha ventana. Por último, creamos el icono, utilizando como ayuda o indicación el título de la aplicación. La destrucción del icono va en paralelo con el cierre de la aplicación, y es igual de sencilla ... ¿he dicho el cierre de la aplicación? ¿Cómo podemos cerrar una aplicación que no tiene ventana principal?

Ha llegado el momento de traer un componente *TPopupMenu* al formulario. Este menú debe aparecer cuando se pulse el botón derecho del ratón sobre el icono de la bandeja. Por el momento, solamente configuraremos un comando de menú dentro de este *TPopupMenu*, y lo llamaremos *Cerrar*. He aquí la respuesta a su selección:

```
procedure TForm1.Cerrar1Click(Sender: TObject);
begin
   Close;
end;
```

Antes he dicho que estaríamos pendientes de cuando el usuario pulsara el botón derecho del ratón sobre el icono. Este tipo de sucesos es el que el icono nos comunicará a través de su mensaje especial. Por lo tanto, vamos a la sección **private** de la declaración de *TForm1* y declaramos un manejador para dicho mensaje:

```
type
   TForm1 = class(TForm)
      // …
   private
      procedure WMIconMessage(var Msg: TMessage);
         message WM_ICONMSG;
      // …
   end;
```

Cuando recibimos el mensaje *WM_ICONMSG*, el *lParam* del mensaje nos informa acerca de qué le ha sucedido al icono; el valor almacenado en este parámetro corresponde al mensaje de Windows que se hubiera enviado de no ser éste un icono, sino una ventana. Nuestra implementación del manejador del mensaje debe pues analizar el *lParam* para decidir qué rumbo tomarán los acontecimientos:

```
procedure TForm1.WMIconMessage(var Msg: TMessage);
var
   Pt: TPoint;
begin
   if Msg.LParam = WM_RBUTTONUP then
   begin
      GetCursorPos(Pt);
      SetForegroundWindow(Application.Handle);
      Application.ProcessMessages;
      PopupMenu1.Popup(Pt.X, Pt.Y);
   end;
end;
```

Y con esto ya tenemos un esquema básico de aplicación para la bandeja de iconos, que usted puede ampliar y mejorar todo lo que desee.

Ciclo de mensajes y concurrencia

El concepto del ciclo de mensajes desempeña un papel fundamental en la arquitectura de las versiones de Windows de 16 bits. Gracias a este algoritmo, Windows 16 puede implementar la *multitarea cooperativa*, en la cual los procesos que se ejecutan concurrentemente en un ordenador ceden gustosamente el control del procesador a otro proceso cuando no tienen nada que hacer. Recordemos la estructura de un ciclo de mensajes:

```
while GetMessage(Msg, 0, 0, 0) do
   // … etcétera …
```

Antes hemos dicho que *GetMessage* espera pacientemente a que haya un mensaje en la cola de la aplicación para retornar. ¿Qué sucede mientras tanto? Como la aplicación está bloqueada en espera del mensaje y no puede realizar ninguna acción, Windows arrebata el control al proceso activo y lo transfiere a otro proceso que tenga mensajes en su cola. Hasta Windows 95, no existía la llamada *multitarea apropiativa*, en la que las aplicaciones tienen que ceder el control a su pesar transcurrido un intervalo determinado de tiempo.

La consecuencia fundamental de esta estructura del sistema operativo de 16 bits es que una aplicación mal desarrollada, o que durante largos intervalos de tiempo ejecute procesos en lote sin revisar su cola, puede dejar colgado al sistema y al resto de las aplicaciones que suspiran lánguidamente por una oportunidad en la vida.

Cuando no hay nada mejor que hacer…

Windows ofrece una alternativa a *GetMessage*: la función *PeekMessage*. Esta función, en vez de esperar y bloquear la aplicación hasta que haya un mensaje en la cola, retorna inmediatamente siempre. Si el valor de retorno es *True*, la función encontró un mensaje. Mediante *PeekMessage* las aplicaciones de Windows 16 podían implementar tareas en segundo plano; basta con ejecutar una pequeña porción de la tarea cada vez que *PeekMessage* devuelva *False* en el ciclo de mensajes. Esto quiere decir que la tarea en cuestión debe poder fragmentarse en instrucciones que no tarden demasiado en ejecutarse. Por ejemplo, si queremos imprimir un fichero mientras un programa desarrolla su hilo principal, podemos imprimir un carácter del fichero cada vez que la aplicación se encuentre ociosa. Este mecanismo, en Delphi, se puede aprovechar interceptando el evento *OnIdle* que es disparado por el objeto *Application*.

Un problema relacionado con la concurrencia cooperativa surge cuando se ejecuta un algoritmo "cerrado", que no necesita leer eventos, y que tarda en devolver el control al ciclo de mensajes. Un ejemplo sería la copia de un fichero. En Windows 16 estos algoritmos dejan bloqueado al sistema operativo entero, pues acaparan egoístamente el tiempo del procesador. En Windows 32 el problema no es tan grave, pues las restantes aplicaciones recuperan el procesador periódicamente gracias a la multitarea apropiativa. No obstante, si la aplicación está programada sobre un solo hilo, el usuario pierde control sobre la interfaz gráfica; no puede, por ejemplo, cancelar la operación que se está desarrollando. Por ello, es necesario intercalar dentro de estos algoritmos instrucciones que revisen la cola de mensajes; Delphi ofrece para este propósito el método *ProcessMessages* de la clase *TApplication*. El siguiente procedimiento implementa un ciclo de espera en Delphi:

```
procedure Esperar(Milisegundos: Integer);
var
   T: Integer;
begin
   T := GetTickCount + Milisegundos;
   // Mientras T sea un instante del futuro …
   while T > GetTickCount do
      Application.ProcessMessages;
end;
```

Ejecutar y esperar

Utilizaremos los conocimientos adquiridos para mostrar cómo lanzar una aplicación desde un programa en Delphi y esperar a que ésta acabe antes de seguir ejecutando. Primero desarrollaremos una función que ejecute una aplicación y nos devuelva el identificador de proceso de la misma. Este es el código necesario:

```
function Ejecutar(const AppName, Params: string): THandle;
var
   ProcInfo: TProcessInformation;
   Info: TStartupInfo;
begin
   FillChar(Info, SizeOf(Info), 0);
   Info.cb := SizeOf(Info);
   Info.dwFlags := STARTF_USESHOWWINDOW;
   Info.wShowWindow := SW_SHOWNORMAL;
   if not CreateProcess(nil, PChar(AppName + ' ' + Params), nil,
      nil, False, 0, nil, nil, Info, ProcInfo) then
      raise Exception.Create('No se puede iniciar la aplicación');
   Result := ProcInfo.hProcess;
end;
```

La función anterior utiliza *CreateProcess*, que es la forma recomendada en Windows 32 para lanzar procesos. Cuando no se puede ejecutar la aplicación indicada, *Ejecutar* lanza una excepción; en el capítulo 11 veremos qué son las excepciones.

A continuación, necesitamos una rutina que espere mientras el proceso está todavía activo:

```
procedure TwndMain.Esperar(H: THandle);
begin
   while WaitForSingleObject(H, 100) = WAIT_TIMEOUT do
   begin
      Application.ProcessMessages;
      if Application.Terminated then Break;
   end;
end;
```

WaitForSingleObject espera a que el proceso finalice, o a que transcurra una décima de segundo. En el segundo caso, permite que la aplicación ejecute un paso del ciclo de

mensajes, y comprueba si la aplicación ha recibido el mensaje *WM_QUIT* para interrumpir el bucle.

Finalmente, creamos una aplicación. En su formulario principal situamos un botón y le asociamos el siguiente manejador de eventos:

```
procedure TwndMain.Button1Click(Sender: TObject);
begin
   Button1.Enabled := False;
   try
      Esperar(Ejecutar('NotePad.Exe', ''));
   finally
      Button1.Enabled := True;
   end;
end;
```

Cuando pulsamos el botón, éste se inhabilita; así evitamos lanzar dos veces la aplicación hija. En este caso, estamos ejecutando el Bloc de Notas de Windows. La instrucción **try...finally** también será estudiada en el capítulo 11.

Capítulo

9

Propiedades

LA TEORÍA "CLÁSICA" de la Programación Orientada a Objetos trata únicamente con atributos y métodos para los objetos. Sin embargo, la programación para Interfaces Gráficas de Usuarios (GUI) plantea necesidades que son cubiertas a duras penas por este modelo de objetos. En este capítulo estudiaremos el modelo PME (*Propiedades/Métodos/Eventos*), y veremos cómo con este modelo nuestros objetos inanimados cobran vida y se reencarnan en *componentes*. En particular, estudiaremos las propiedades, dejando para el siguiente capítulo los eventos.

Los problemas de la POO clásica

En los primeros tiempos heroicos de la programación para Windows, el único lenguaje con que contaba el programador era C. Intentaré contener mis emociones negativas hacia este lenguaje, pero no puedo dejar de mencionar, como hecho curioso, que la instrucción peor diseñada de C, la instrucción **switch**[12], era la instrucción más usada en aquellos primeros programas. Como estudiamos en el capítulo anterior, el núcleo de un programa en Windows es el bucle de mensajes y los *procedimientos de ventanas*; estos últimos consisten en una gigantesca instrucción de selección.

La Primera Revolución en la programación para Windows se produjo cuando empezaron a aparecer bibliotecas de objetos para este sistema; en inglés las denominaban *Application Frameworks*. El primer paso importante lo dio (¡quién podía ser si no!) Borland, con Turbo Pascal para Windows 1.0. Este producto, el primer compilador de Pascal para Windows, incluía un conjunto de clases, bautizado como Object Windows. Existe una anécdota acerca de cierto escritor famoso de libros técnicos, vinculado a la programación en C y a Microsoft, quien "predijo" que la programación para Windows en Pascal era poco menos que imposible.

El éxito de esta biblioteca de clases provocó su migración a C++. En la siguiente versión de Borland C++ para Windows, se incluía Object Windows para C++, versión 1.0. Y aquí comenzaron los problemas. La cuestión es que Borland había dise-

[12] Equivalente al **case** de Pascal, que tampoco es una maravilla.

ñado una novedosa técnica para eliminar las instrucciones de selección de mensajes: las tablas de métodos dinámicos, que aprovechan los métodos declarados con la directiva **message**. Para trasladar el código de Pascal a C++, fue necesario ampliar la sintaxis de la definición de métodos virtuales de este último lenguaje. Y la comunidad de programadores de C++, obsesionada por la "portabilidad", reaccionó virulentamente a la innovación. La actitud del programador medio hacia los entornos gráficos por aquel entonces era:

> *"Las interfaces de usuario son la nata encima del pastel. Como esto puede ser una moda pasajera, no voy a aceptar cambios en la forma en que pienso y programo."*

Pobre programador, las GUI llegaron para quedarse. Como resultado de las críticas recibidas, algunas justas pero la mayoría injustas, Borland tuvo que dar marcha atrás. Para aquella época, Microsoft había desarrollado ya su primera biblioteca de clases para Windows, la famosa *Microsoft Foundation Classes* (MFC). Esta biblioteca utilizaba macros de C para resolver el problema de despachar mensajes. La versión 2.0 de ObjectWindows para C++ adoptó este mecanismo, rechazando los métodos dinámicos. El autor de este libro programó un tiempo con estas bibliotecas. Son muy potentes, pero el mecanismo de macros obliga al desarrollador a memorizar gran número de identificadores y estilos de llamadas sin relación entre sí, además de que hay que teclear más y que aumenta la posibilidad de introducir errores. Fueron épocas de locura e insania; si alguien me preguntaba por mi nombre y dirección mientras estaba frente al ordenador, tenía que esperar a que mi mecanismo de paginación trajera a la memoria la página con los datos descartados.

De cualquier manera, utilícense métodos dinámicos o macros, el enfoque clásico de la Programación Orientada a Objetos, que solamente considera objetos y métodos, plantea las siguientes dificultades a la hora de programar para interfaces gráficas:

- Una GUI, debido a su inevitable evolución histórica, contiene por lo general demasiadas funciones como para ser recordadas por el programador. Una parte de esta complejidad se produce por la necesidad de tener métodos de acceso y de transformación por separado.
- La mayor parte de los objetos que ofrece una biblioteca de objetos para controlar una interfaz gráfica son, precisamente, objetos del estilo "mando a distancia". Muchos métodos de este tipo de clases se llaman de dos en dos: el primero cambia el estado interno del objeto, y el segundo "transmite" la acción al elemento de interfaz controlado.
- Las GUI necesitan comunicación bidireccional. La solución ofrecida por la POO clásica para "escuchar" a los objetos es redefinir clases para sobrescribir métodos virtuales. Esta metodología es, por razones incluso tipográficas, larga y tediosa.

El tercer problema se analizará en el siguiente capítulo, pues nos servirá para introducir los eventos. Consideremos, entonces, los dos primeros problemas. En primer

lugar, tenemos la existencia separada de métodos de acceso y de transformación. Por ejemplo, para leer y modificar el título de una ventana, Windows ofrece el siguiente par de funciones:

```
function  GetWindowText(Wnd: HWnd; Str: PChar;
   MaxLen: Integer): Integer;
procedure SetWindowText(Wnd: HWnd; Str: PChar);
```

Utilizando la POO clásica, estos procedimientos se transformarían en métodos de una hipotética clase *TVentana*:

```
function TVentana.GetText: string;
procedure TVentana.SetText(Str: string);
```

Sin embargo, como el lector sabe o puede fácilmente comprobar, Delphi utiliza una sola "propiedad", *Caption* para leer o modificar el título de una ventana. La respuesta a "cómo lo hace" vendrá después:

```
if Ventana1.Caption = 'Yin' then
   Ventana1.Caption := 'Yang';
```

Analicemos ahora el segundo argumento, el de la necesidad de pares de métodos: uno para cambiar el estado interno y el otro para su transmisión. Supongamos que queremos programar una etiqueta de texto tridimensional con posibilidad de cambiar la profundidad del texto. Declaramos un atributo *Profundidad* en el objeto *TLabel3d*, de tipo entero positivo. Si utilizamos POO clásica, nos vemos en un dilema con respecto a este atributo. El problema es que la modificación del mismo, si se realiza directamente, actúa solamente sobre el estado interno del objeto de etiqueta, que solamente es un "transmisor". Estamos obligados a utilizar una solución similar a la siguiente, en la que se tiene que encapsular el método de modificación y, en consecuencia, el método de acceso:

```
type
   TLabel3d = class(…)
   private
      FProfundidad: Word;
   public
      function Profundidad: Word;
      procedure CambiarProfundidad(P: Word);
   end;
```

CambiarProfundidad debe asignar un valor al atributo privado *FProfundidad*, y redibujar el objeto. La función *Profundidad* devuelve directamente el valor del atributo interno. El programador puede utilizar estos métodos así:

```
if MiEtiqueta.Profundidad < 3 then
   MiEtiqueta.CambiarProfundidad(3);
```

El principal problema con este código es que lo que podía ser una sencilla y eficiente pregunta por el valor de un atributo, se convierte en una costosa llamada a función.

– ¡Pero para eso C++ tiene funciones en línea! - alardea un fanático de este lenguaje.

Es cierto que las funciones en línea pueden resolver este problema de eficiencia, pero por favor, que no mencionen a C++ en mi presencia; nuestro avispado programador de C++ tiene que programar de este modo:

```
if (MiEtiqueta->Profundidad() < 3)
   MiEtiqueta->CambiarProfundidad(3);
```

En este par de instrucciones queda claro, demasiado claro, que *Profundidad* es una función. Si posteriormente cambiamos el diseño de la clase y se revierte la función en atributo, ¡busque todas las llamadas a la función y quítele los paréntesis! El principio de diseño de lenguajes que C y C++ violan en este caso se conoce como *uniformidad referencial*.

Propiedades

Las propiedades son una extensión al concepto de campo o atributo de una clase. Una propiedad se define mediante la siguiente sintaxis básica:

```
property Nombre: Tipo read ModoLectura write ModoEscritura;
```

En el caso más general, tanto la forma de leer como la de escribir son métodos. El método de lectura debe ser una función sin parámetros que devuelva un valor del mismo tipo que la propiedad. El método de escritura, por su parte, debe ser un procedimiento monoparamétrico, cuyo parámetro debe tener el mismo tipo que la propiedad. Por ejemplo:

```
type
   TAtomo = class
   private
      function GetCarga: Integer;
      procedure SetCarga(Value: Integer);
   public
      property Carga: Integer read GetCarga write SetCarga;
   end;
```

Cuando el programador "hace algo" con una propiedad, el compilador utiliza la especificación de los métodos de acceso para "traducir" el código. Muestro a continuación una instrucción con su correspondiente traducción:

```
// Esta es la instrucción original:
Nitrogeno.Carga := Oxigeno.Carga - 1;
// Esta es la traducción:
Nitrogeno.SetCarga(Oxigeno.GetCarga - 1);
// La carga del oxígeno es 8, si a alguien le interesa.
```

Naturalmente, no todas las cosas que pueden hacerse con un campo o atributo pueden realizarse con las propiedades. La limitación básica es que no podemos obtener la dirección de una propiedad, pues una propiedad no es un atributo. Esto implica que tampoco podemos utilizar una propiedad en un parámetro por referencia de un procedimiento, pues el traspaso por referencia implica el uso de punteros:

```
Inc(Mercurio.Carga);  // ¡¡¡No se puede (desgraciadamente)!!!
```

Siempre, o casi siempre, los métodos de acceso de una propiedad se declaran en la parte privada de la clase. ¿Para qué definir un método de acceso alternativo a un recurso si ya existe otro?

Implementación de propiedades mediante atributos

La implementación de los métodos de acceso mediante procedimientos y funciones es apropiada si el objeto en que se encuentran pertenece al modelo de "mando a distancia", esto es, si el valor de la carga se almacena en una variable perteneciente a otro objeto, o está "enterrada" en el sistema operativo, o si hace falta un cálculo complicado para deducirla a partir de otras variables. Pero lo más común, como sabemos, es que el propio objeto almacene el estado de la propiedad. Si estamos implementando átomos, lo natural es almacenar el valor de la carga internamente en una variable del objeto:

```
type
   TAtomo = class
   private
      FCarga: Integer;
      // …
   end;
```

La letra *F* inicial es un convenio de Borland para atributos privados de una clase, y quiere decir *field*, es decir, *campo*. Partiendo de esta implementación, ¿para qué utilizar propiedades? ¿No es mejor dejar en la sección **public** de la clase el atributo *Carga*, sin la *F*, y evitarnos complicaciones? No, no es lo mismo. La carga de un átomo es un valor mayor o igual que 1, y si damos acceso directo a la variable, un programador puede asignar un valor negativo a la carga. Por lo tanto, necesitamos que el acceso en modo escritura a la carga transcurra mediante un procedimiento.

En este tipo de situación, podemos declarar la propiedad *Carga* del siguiente modo:

```
type
   TAtomo = class
   private
      FCarga: Integer;
      procedure SetCarga(Value: Integer);
   public
      property Carga: Integer read FCarga write SetCarga;
   end;
```

De este modo, nos evitamos las llamadas a una función *GetCarga*, que solamente haría más lento el acceso para lectura a la propiedad. La instrucción que mostramos anteriormente, la que asigna una carga al átomo de nitrógeno, se traduce ahora de la siguiente forma:

```
// Nueva traducción:
Nitrogeno.SetCarga(Oxigeno.FCarga - 1);
```

Posiblemente, el procedimiento *SetCarga* se programa de este modo:

```
procedure TAtomo.SetCarga(Value: Integer);
begin
   if Value <> FCarga then
      if FCarga < 1 then
         raise Exception.Create('Carga menor que la unidad')
      else
         Value := FCarga;
end;
```

En el código anterior he utilizado una instrucción de lanzamiento de excepciones; la instrucción **raise** interrumpe el proceso que se está ejecutando, comunicando al usuario el error descrito en la cadena de caracteres pasada al constructor *Create* de la excepción.

La primera línea de la implementación del método *SetCarga* muestra también algo común a casi todas las propiedades: si a una propiedad se le asigna un valor igual al que tenía anteriormente, no debe suceder nada en absoluto. Este es un convenio que deben seguir todos los desarrolladores de componentes.

Por último, es posible utilizar también un atributo en sustitución del método de escritura de una propiedad:

```
type
   TAtomo = class
   private
      FCarga: Integer;
   published
      property Carga: Integer read FCarga write FCarga;
   end;
```

Si el lector presta atención, verá que ahora la propiedad ha sido declarada en la sección **published**, y en tiempo de ejecución existirá información sobre ella. Si además

la clase *TAtomo* descendiese de *TComponent* y la situáramos en la Paleta de Componentes, la propiedad *Carga* podría aparecer en el Inspector de Objetos en tiempo de diseño Este es el sentido de utilizar atributos para ambos métodos de acceso; si la propiedad no se sitúa en la parte **published**, no se gana nada con definir una propiedad en vez de utilizar directamente el atributo.

Si no creemos en la alquimia…

Los alquimistas buscaban el camino para convertir el plomo en oro. Si no creemos en la alquimia y no nos interesa la física nuclear, no debemos permitir la posibilidad de cambiar la carga de un átomo una vez que éste ha sido creado. Podemos lograr que la propiedad *Carga* pueda solamente leerse, pero no modificarse, si omitimos la cláusula **write** en la definición de la propiedad:

```
type
   TAtomo = class
   private
      FCarga: Integer;
   public
      property Carga: Integer read FCarga;
   end;
```

Antes de la introducción de las propiedades en el lenguaje, se podía lograr un efecto similar definiendo *Carga* como una función sin parámetros:

```
function Carga: Integer;
```

De este modo, se podía llamar a la función cada vez que necesitáramos el valor, sin poder realizar asignaciones a la misma. La uniformidad referencial se logra en Pascal gracias a que una llamada a una función sin parámetros no necesita los paréntesis para que el compilador "se entere" de que estamos tratando con una función. Sin embargo, utilizar una función es menos eficiente que utilizar una propiedad de sólo lectura basada en un atributo.

¿Y qué hay con las propiedades de sólo escritura? También son posibles en Object Pascal, pues basta con omitir la cláusula **read** de la propiedad. Ahora bien, son menos frecuentes. La única propiedad de este tipo que conozco es la propiedad *Output*, de tipo **string**, de la clase *TApdComPort*, de la librería de comunicaciones Async Professional, de TurboPower Software. Para enviar una cadena de caracteres por un puerto serial con este componente, se realiza la siguiente asignación:

```
ApdComPort1.Output := '1234567890';
```

Esta asignación se traduce en una llamada a un procedimiento con un solo parámetro:

```
ApdComPort1.SetOutput('1234567890');
```

Anteriormente hemos visto como con la programación orientada a objetos se reduce la cantidad necesaria de parámetros de los procedimientos y funciones. La mayor parte de los métodos resultantes se definen con uno o con ningún parámetro. Y ahora nos encontramos con que la mayor parte de los métodos de transformación de un solo parámetro se pueden rediseñar como propiedades. El truco consiste en que este parámetro casi siempre se utiliza para modificar una variable del estado interno del objeto, de la cual posiblemente también queramos saber su valor. El procedimiento monoparamétrico puede utilizarse como método de escritura en la definición de la propiedad, y el método de lectura será la variable interna o una función equivalente.

En el caso de la propiedad *Output*, los desarrolladores de Async Professional decidieron, correcta o incorrectamente, que no era importante para el programador saber cuál es el último valor que se ha asignado a la propiedad en cuestión, por lo cual omitieron añadir un método de lectura para la misma.

La semántica de la asignación a propiedades

Como ya sabe el lector, las variables declaradas de tipo clase son en realidad punteros a objetos. Una asignación entre dos variables compatibles de estos tipos se implementa como una asignación entre punteros, con todas las consecuencias predecibles:

```
var
   Font1, Font2: TFont;
begin
   Font1 := TFont.Create;
   Font2 := Font1;
   // Ambas variables apuntan ahora al mismo objeto
   Font1.Free;
   // Se ha destruido el objeto
   // ¡La dirección almacenada en Font2 ya no es válida!
end;
```

Muchos componentes de Delphi tienen una propiedad llamada *Font*, de tipo *TFont*. ¿Qué sucede con asignaciones como la siguiente?

```
Memo1.Font := FontDialog1.Font;
```

Ambas propiedades *Font* devuelven punteros a objetos de esta clase. ¿Quiere decir que estamos frente a una asignación de punteros? Resulta que no. Si *Font* fuera una variable en vez de una propiedad, claro que sería una asignación de punteros. Pero al ser *Font* una propiedad, esta asignación se traduce en una aplicación del método de escritura:

```
Memo1.SetFont(FontDialog1.GetFont);
```

La implementación de *SetFont* es, posiblemente, la siguiente:

```
procedure TMemo.SetFont(Value: TFont);
begin
   FFont.Assign(Value);
end;
```

Como recordará, el método *Assign* realiza la copia de los campos de un objeto en otro. Este comportamiento es el recomendado para propiedades de tipo objeto, pero un desarrollador de componentes puede decidir que las asignaciones a cierta propiedad deben implementarse como asignaciones a punteros:

```
Label1.FocusControl := Edit1;
```

En la instrucción anterior, no se crea un nuevo objeto de tipo cuadro de edición, ni se copian las propiedades de *Edit1* en algún objeto construido dentro de *Label1*. El implementador de *TLabel* decidió que *SetFocusControl* asignara punteros en vez de llamar a *Assign*. Por supuesto, en este caso hay peligro de que la destrucción de *Edit1* deje un valor incorrecto en la propiedad *FocusControl* de *Label1*. El desarrollador de *TLabel* previene estos problemas redefiniendo en esta clase el método virtual *Notification*:

```
procedure TLabel.Notification(AComponent: TComponent;
   Operation: TOperation);
begin
   inherited Notification(AComponent, Operation);
   if (Operation = opRemove) and (AComponent = FFocusControl) then
      FFocusControl := nil;
end;
```

Notification es un método virtual declarado en la clase *TComponent*, que se llama cuando un componente es creado o destruido. En el método anterior, el objeto *TLabel* detecta la destrucción de un componente (la operación es *opRemove*). Entonces, con el corazón en un puño, pregunta si el fallecido es su viejo amigo, *FFocusControl*, para no enviarle más tarjetas de felicitación por Navidad.

El método *FreeNotification* está estrechamente relacionado con *Notification*, y fue introducido por Delphi 2, debido a la existencia de referencias en tiempo de diseño entre componentes ubicados en distintos formularios. *Notification* solamente se llama de forma automática cuando el objeto borrado o insertado está en el mismo formulario que el objeto notificado. Para garantizar la recepción de estas notificaciones incluso en los casos en que los punteros se saltan las fronteras de fichas, hay que indicarlo con una llamada a *FreeNotification*. El siguiente código muestra un fragmento del método de escritura asociado a la propiedad *FocusControl* de la clase *TLabel*:

```
procedure TLabel.SetFocusControl(Value: TWinControl);
begin
   FFocusControl := Value;
   if Value <> nil then Value.FreeNotification(Self);
end;
```

Propiedades vectoriales

Como hemos explicado, las propiedades permiten consultar y modificar el estado de un objeto mediante simples asignaciones. Sin embargo, en ocasiones este estado está estructurado de forma compleja. La forma más frecuente de "estructuración" se presenta en propiedades a las cuales hay que acceder mediante un índice: las propiedades vectoriales. Por ejemplo, para obtener y modificar el texto de la primera línea de un cuadro de lista, se utilizan las siguientes instrucciones:

```
PrimeraLinea := ListBox1.Items[0];
if PrimeraLinea = 'Richard Bachman' then
   ListBox1.Items[0] := 'Stephen King';
```

Una propiedad vectorial se declara mediante la siguiente sintaxis:

```
property Propiedad[Indice: TipoIndice]: Tipo
   read MétodoLectura write MétodoEscritura;
```

Esta vez no se pueden utilizar variables privadas para el acceso a la propiedad, siendo necesarios tanto la función lectora como el procedimiento escritor. Como ejemplo, mostraremos parte de la definición de la clase *TList*, que estudiaremos en un capítulo posterior. Esta clase almacena una lista de punteros, a los cuales se accede por medio de la propiedad vectorial *Items*:

```
type
   TList = class
   private
      // ...
      function GetItems(I: Integer): Pointer;
      procedure SetItems(I: Integer; Value: Pointer);
   public
      // ...
      property Items[I: Integer]: Pointer
         read GetItems write SetItems; default;
   end;
```

Más adelante explicaré el uso de la cláusula **default**. Con las declaraciones anteriores, se pueden efectuar las siguientes operaciones:

```
var
   Elemento: Pointer;
   Lista: TList;
begin
   // ...
```

```
    Elemento := Lista.Items[0];
    Lista.Items[0] := Lista.Items[1];
    Lista.Items[1] := Elemento;
    // ...
end;
```

El código anterior se traduce internamente al siguiente:

```
    Elemento := Lista.GetItems(0);
    Lista.SetItems(0, Lista.GetItems(1));
    Lista.SetItems(1, Elemento);
```

¿Cuáles son las cotas mínima y máxima de una propiedad vectorial? Resulta que estas cotas no existen realmente, y se definen a discreción por la propiedad. Casi todas las propiedades vectoriales para las que el tipo de su índice es un entero, utilizan el 0 como cota mínima, mientras que la cota máxima varía dinámicamente; en el caso de las listas de punteros, la cota máxima es *Lista.Count - 1*, donde *Count* es la cantidad de elementos almacenados en la lista. Si el programador utiliza un valor fuera de rango para el índice, el método que implementa el acceso a la propiedad lanza una excepción para indicar el error.

Como se deduce de la sintaxis de la definición de propiedades, el tipo del índice de la propiedad no tiene por qué ser un valor entero, u ordinal, en general. Esto nos permite implementar en Delphi un tipo de datos abstracto conocido como *vectores asociativos*. Por ejemplo, en Delphi tenemos la clase *TStrings* para representar listas de cadenas. Si estas cadenas tienen todas el formato *Parámetro=Valor*, como el de los ficheros de configuración, es posible utilizar la propiedad *Values* de esta clase para acceder a un valor conociendo el nombre de un parámetro:

```
if LoginParams.Values['USERNAME'] = '' then
begin
    LoginParams.Values['USERNAME'] := 'Tutankhamon';
    LoginParams.Values['PASSWORD'] := 'Osiris';
end;
```

La propiedad *Values* ha sido definida del siguiente modo:

```
type
    TStrings = class
    private
        function GetValue(const Name: string): string;
        procedure SetValue(const Name, Value: string);
    public
        property Values[const Name: string]: string
            read GetValue write SetValue;
    end;
```

Nuevamente, es responsabilidad de *GetValue* y de *SetValue* el definir qué se hace cuando se pasa un índice que no tiene valor asociado, como puede ser un nombre de parámetro inexistente.

En contraste con el carácter vectorial de estas propiedades, a las propiedades "normales" se les llama con frecuencia *propiedades escalares*.

Propiedades vectoriales por omisión

Al tipo de recurso que explicaré ahora se le conoce en inglés como *syntactic sugar*: una facilidad de tipo puramente sintáctico, pero que permite teclear menos y hacer más expresivos los listados de programas. Se trata de la posibilidad de definir propiedades vectoriales por omisión. Recordemos la definición de la propiedad *Items* de la clase *TList*:

```
property Items[I: Integer]: Pointer
   read GetItems write SetItems; default;
```

La cláusula **default** indica que podemos omitir el nombre de la propiedad y aplicar directamente los corchetes a la variable de la clase. Aprovechando esto, podemos escribir cosas como:

```
var
   Lista: TList;
   Elemento: Pointer;
begin
   // …
   Elemento := Lista[0];      // En vez de Lista.Items[0]
   Lista[0] := Lista[1];
   Lista[1] := Elemento;
   // …
end;
```

A primera vista, da la impresión de que la variable *Lista* es una variable vectorial, aunque realmente es una variable de clase. Por lo demás, la traducción de estas instrucciones es exactamente la misma que antes. Esta técnica se parece en ciertos aspectos a la sobrecarga de operadores en C++.

Solamente puede haber una propiedad vectorial por omisión en cada clase. Una propiedad por omisión mantiene tal carácter incluso en las clases que heredan de la clase en que se define. Por supuesto, no existe nada parecido a una propiedad escalar por omisión.

Especificaciones de almacenamiento

Aunque las propiedades nos permiten una mejor abstracción del concepto de estado interno de un objeto, y nos ofrecen facilidades sintácticas y mnemotécnicas para actuar sobre el objeto, su propósito primordial es permitir la especificación del es-

tado inicial del objeto durante el tiempo de diseño. Es esta la diferencia básica entre un sistema RAD y los entornos tradicionales de programación.

Para ilustrar mejor el tema necesito contar un poco de historia. En el Neolítico de los PCs, antes de que se generalizara el uso de Windows y aparecieran las primeras librerías de desarrollo orientadas a objetos para este sistema operativo, Borland comercializaba junto con sus versiones de Pascal para DOS una librería denominada TurboVision, para el manejo de ventanas, diálogos, menús y todo eso (no confundir con ObjectVision, otro antiguo producto de Borland). TurboVision era orientado a objetos y su arquitectura estaba basada en eventos. El gran problema de TurboVision era la creación de los objetos visuales. Si necesitábamos un cuadro de diálogo con veinte cuadros de edición y cinco botones, había que programar instrucciones para la creación e inicialización de cada uno de estos objetos. Ciertos estudios realizados sobre programas reales llegaron a la conclusión de que el 15% del código programado se dedicaba a la creación e inicialización de objetos.

En un sistema operativo con interfaz gráfica el problema adquiere mayores connotaciones. Windows reconoce este problema y ofrece los denominados *ficheros de recursos* para hacer frente al asunto. Un fichero de recursos contiene la descripción binaria de uno o más objetos, algunos de ellos gráficos. Por medio de funciones simples del API, es posible crear un objeto complejo, como puede ser un cuadro de diálogo, especificando solamente dónde encontrar el recurso asociado. Los ficheros de recursos se editan comúnmente mediante herramientas visuales, y se incorporan al ejecutable de la aplicación en la fase final del proceso de enlace; a este fase se le denomina en ocasiones *binding*.

Pero las especificaciones de recursos de Windows se limitan a cierta cantidad limitada de objetos: diálogos, menús, mapas de bits, etc. Delphi, y los demás sistemas RAD, aumentan este mecanismo para poder hacer frente a la inicialización de clases arbitrarias definidas por el programador; en el caso de Delphi, mediante el mecanismo de los ficheros *dfm*. La VCL de Delphi puede inicializar cualquier clase basada en la clase predefinida *TPersistent* a partir de información guardada en uno de estos ficheros. Y esta técnica se basa en la generación de información en tiempo de compilación para las propiedades definidas en las secciones **published** de las clases de componentes.

Normalmente, los valores de las propiedades de un componente se graban en el fichero *dfm* cuando el programador crea el componente en tiempo de diseño. Posteriormente, en tiempo de ejecución, durante la creación de los formularios, cada componente debe ser capaz de extraer estos valores del fichero *dfm*. La información acerca de cómo leer y escribir los valores de una propiedad en un fichero *dfm* se genera de forma automática, sin necesidad de que el programador intervenga. No obstante, con el propósito de optimizar el tamaño de la información almacenada, el programador puede utilizar dos cláusulas, **store** y **default**, para indicarle a la VCL cuándo puede omitir el almacenamiento del valor de una propiedad. Estas cláusulas

se aplican solamente a propiedades escalares. La más sencilla y fácil de comprender es **default**:

```
type
   TAtomo = class(TPersistent)
   published
      property Carga: Integer read FCarga write SetCarga default 1;
   end;
```

Al indicar que el valor por omisión de la carga es 1, no estamos realizando la inicialización del mismo. Por el contrario, en el constructor de la clase seguimos asignando un valor para esta propiedad:

```
constructor TAtomo.Create;
begin
   inherited Create;
   FCarga := 1;
   // …
end;
```

Lo único que hemos dicho, al especificar la cláusula **default**, es que si al guardar un átomo en un *dfm* encontramos que su carga es unitaria, no hace falta guardar el valor para esta propiedad. Esto permite quitar bastante carga en los ficheros *dfm*. Un componente típico tiene unas cuantas propiedades que casi nunca son modificadas por el programador. Si siempre se guardaran todos los valores de todas las propiedades, los *dfm*s crecerían desmesuradamente. No obstante, las cláusulas **default**, además de estar limitadas a propiedades escalares, solamente pueden aplicarse a aquellas propiedades de tipo ordinal (enteros, enumerativos), o a tipos de conjuntos con un máximo de elementos menor o igual que 32.

El mayor control sobre el algoritmo de almacenamiento se logra con la cláusula **store**. A través de esta cláusula indicamos si queremos almacenar el valor de una propiedad, mediante una constante lógica o una función sin parámetros que retorne verdadero o falso. Por ejemplo:

```
type
   TAtomo = class(TPersistent)
   private
      FNombre: string;
      // …
      function AlmacenarCarga: Boolean;
      function AlmacenarNombre: Boolean;
   published
      property Carga: Integer read FCarga write SetCarga
         store AlmacenarCarga;
      property Nombre: string read FNombre write SetNombre
         store AlmacenarNombre;
   end;
```

```
function TAtomo.AlmacenarCarga: Boolean;
begin
   Result := FCarga <> 1;
end;

function TAtomo.AlmacenarNombre: Boolean;
begin
   Result := FNombre <> '';
end;
```

En este caso, el algoritmo de *AlmacenarCarga* es equivalente a la cláusula **default** del ejemplo anterior, pero pueden utilizarse procedimientos de decisión arbitrarios. *AlmacenarNombre*, por ejemplo, evita que se guarden en el fichero *dfm* nombres vacíos, algo que no podemos lograr con una cláusula **default** por ser la propiedad de tipo **string**.

Acceso a propiedades por su nombre

¿Es posible saber si determinado objeto, cuya clase exacta desconocemos, posee determinada propiedad? ¿Se puede cambiar entonces el valor de dicha propiedad? Sí, Delphi lo permite cuando la propiedad ha sido declarada en una sección **published**. ¿Qué sentido tiene todo esto? Recordemos que Delphi no soporta la herencia múltiple. En el capítulo 17 veremos, por ejemplo, que todos los controles asociables a bases de datos tienen una propiedad *DataSource*, para especificar indirectamente de qué tabla extraen sus datos. Desgraciadamente, esta propiedad no ha sido declarada en una clase base común a todos estos controles, sino que ha sido adquirida independientemente a lo largo del arduo proceso evolutivo en la jerarquía de herencia. De modo que, si tenemos una colección de objetos de tipo *TControl*, no podemos trabajar polimórficamente con la propiedad *DataSource* de aquellos objetos que la implementen, si nos limitamos a la programación orientada a objetos "clásica".

La unidad *TypInfo* ofrece funciones y procedimientos para poder manejar propiedades en tiempo de ejecución cuando solamente tenemos el nombre de la misma. Como no hemos presentado todavía los componentes de bases de datos, realizaremos la demostración con un ejemplo más sencillo: tenemos una variada colección de controles en un formulario, y queremos desactivarlos en una sola operación genérica. Para esto, asignaremos *False* a las propiedades *Enabled* de los controles, y la cadena vacía a la propiedad *Text* de aquellos controles que la implementen, como los cuadros de edición. Coloque en un formulario vacío varios controles heterogéneos, entre ellos al menos un botón, y asigne la siguiente respuesta al evento *OnClick* del botón:

```
procedure TwndMain.Button1Click(Sender: TObject);
var
   P: PPropInfo;
   I: Integer;
   Desactivar: Boolean;
```

```
begin
   Desactivar := TButton(Sender).Caption = 'Desactivar';
   for I := 0 to ComponentCount - 1 do
      if Components[I] <> Sender then
      begin
         P := GetPropInfo(Components[I].ClassInfo, 'Enabled');
         if P <> nil then
            SetOrdProp(Components[I], P, Ord(not Desactivar));
         P := GetPropInfo(Components[I].ClassInfo, 'Text');
         if (P <> nil) and Desactivar then
            SetStrProp(Components[I], P, '');
      end;
   if Desactivar then
      TButton(Sender).Caption := 'Activar'
   else
      TButton(Sender).Caption := 'Desactivar';
end;
```

Para que este código compile, debemos incluir la unidad *TypInfo* en la cláusula **uses** del fichero. La función *GetPropInfo* nos permite obtener un puntero a un descriptor de propiedades; recuerde que esto funcionará sólo con propiedades publicadas. Luego, los procedimientos *SetOrdProp* y *SetStrProp* nos permiten asignar valores a las propiedades. También existen funciones *GetOrdProp* y *GetStrProp* para recuperar los valores de una propiedad dado su descriptor.

Capítulo

10

Eventos

PRIMER PLANO: las manecillas de un reloj antiguo se van acercando a las 8 de la mañana. Cuando el horario alcanza esa posición, su borde afilado rompe una fina cuerda que sostiene, a través de una polea, una tostadora. La tostadora cae sobre una tijera que corta un alambre, que deja en libertad a un ratón, tras el cual comienza a correr un gato sobre una cinta infinita que va subiendo una jarra de agua hasta el techo. Cuando la jarra no puede subir más, se derrama sobre el hombre que duerme plácidamente en la vieja cama y...

La historia anterior está inspirada en una película muda, pero describe adecuadamente el funcionamiento de un programa escrito en Delphi. En este lenguaje, la activación de un método puede realizarse automáticamente en respuesta a acciones del usuario o a cambios del estado interno de ciertos objetos. En el presente capítulo estudiaremos los *eventos*, que constituyen el mecanismo ofrecido por Delphi para este propósito.

Punteros a funciones

Tras toda la filosofía de los eventos se esconde el mecanismo de punteros a funciones o procedimientos, por lo cual comenzaremos analizando esta técnica. En sus principios, Pascal no tenía un tipo de datos para almacenar punteros a funciones y procedimientos. Curiosamente, la primera extensión de este tipo la propuso el propio autor del lenguaje, Niklaus Wirth, pero solamente contemplaba la posibilidad de utilizar estos tipos para el traspaso de parámetros. El Pascal de Borland incluyó los punteros a procedimientos en su versión 4.

¿Qué es un puntero a función? Cuando ejecutamos un programa, cada función o procedimiento incluido en éste tiene una posición bien determinada en el segmento de código de la memoria de la máquina. Esta dirección, tanto en programas de 16 como de 32 bits sobre procesadores Intel, se puede expresar en 32 bits. Esto es un puntero a función.

Cuando ejecutamos una rutina del modo tradicional, estamos instruyendo a la máquina que "salte" a una dirección fija en memoria:

```
function SenoHiperbolico(Parametro: Double): Double;
// Supongamos que estas instrucciones se cargan
// en la dirección 0010CAFE.
// El seno hiperbólico es la función (e^x - e^-x)/2, no es pornopoesía.
begin
   Result := Exp(Parametro);
   Result := (Result - 1 / Result) / 2;
end;

var
   X: Double;
begin
   X := SenoHiperbolico(Pi / 2);
   // Quiere decir:
   // Ejecuta la rutina 0010CAFE con el parámetro 1.57 y ...
   X := CosenoHiperbolico(Pi / 2);
   // Ejecuta la rutina 0307DEB0 con el parámetro 1.57 y ...
end;
```

Si utilizamos variables para almacenar las direcciones de rutinas, podemos hacer cosas como la siguiente:

```
type
   TFuncionMatematica = function (Arg: Double): Double;

var
   X: Double;
   F: TFuncionMatematica;
begin
   X := SenoHiperbolico;
   // Se almacena 0010CAFE en la variable F
   X := F(Pi / 2);
   // Quiere decir:
   // Ejecuta la rutina cuya dirección está almacenada en F y ...
end;
```

Los punteros a procedimientos no permiten el trabajo con las funciones y procedimientos anidados de Pascal: aquellos definidos localmente dentro de otros procedimientos.

Punteros a métodos

Los punteros a funciones existen en la programación desde hace mucho tiempo. Pero con la llegada de la Programación Orientada a Objetos las cosas se complicaron. ¿Qué sentido debe tener un puntero a un método? Bjarne Stroustrup, el autor de C++, escogió el siguiente camino: un puntero a método sigue almacenando una dirección a una rutina. Esto quiere decir que para ejecutar la rutina necesitaremos, además de los parámetros, el objeto sobre el cual se aplica el método.

El significado de los punteros a métodos de Delphi, sin embargo, es diferente. En Object Pascal, este tipo de datos almacena la dirección de la rutina asociada al método y, además, el objeto sobre el cual se debe aplicar éste. Un puntero a método se declara del mismo modo que un puntero a función, pero al final de la declaración hay que añadir la frase **of object**. Por ejemplo:

```
type
   TNotifyEvent = procedure (Emisor: TObject) of object;
```

Desde el punto de vista de la implementación, un puntero a método se representa en 8 bytes mediante un par de punteros "normales" de 4 bytes: la dirección del objeto receptor y la dirección del método del mismo que debe ejecutarse en respuesta al evento.

La sintaxis de la asignación a variables de este nuevo tipo es similar a la de los punteros a procedimientos, con dos excepciones:

- Esta vez necesitamos un método, no un procedimiento.
- Además del método necesitamos un objeto. Este puede ser *Self*, y es asumido implícitamente.

Por ejemplo:

```
var
   PuntMet1, PuntMet2: TNotifyEvent;

procedure TForm1.Metodo1(Sender: TObject);
begin
   // …
end;

procedure TForm1.Metodo2;
begin
   PuntMet1 := Metodo1;
   // Realmente: PuntMet1 := Self.Metodo1;
   PuntMet2 := Form2.Metodo3;
   // El objeto receptor es Form2
   PuntMet1 := nil;
end;
```

En la última instrucción estamos asignando un puntero vacío a la variable *PuntMet1*. Posteriormente, podemos utilizar la función *Assigned* para saber si un puntero a método contiene el puntero vacío. El ejemplo siguiente comprueba el valor almacenado en variables de punteros a métodos antes de ejecutar el método asociado:

```
procedure TForm1.Metodo4;
begin
   if Assigned(PuntMet2) then PuntMet1(Self);
   // Llama a: Form2.Metodo3(Self);
end;
```

Objetos activos: ¡abajo la vigilancia!

Nada es eterno, sólo la tierra y el cielo[13]. Incluso los átomos, indivisibles en apariencia, se desintegran al cabo de un tiempo aleatorio, disminuyendo su masa nuclear. Supongamos que nuestra aplicación tiene en sus manos un átomo radioactivo de Carbono 14, y quiere avisar al usuario cuando el átomo sufra su metamorfosis. Es probable que ésta sea nuestra primera idea:

```
var
   Carbono: TAtomo;
begin
   Carbono := TAtomo.Create(6, 14);  // Carga y masa nuclear
   while Carbono.Masa = 14 do
      { Esperar pacientemente };
   ShowMessage('¡Atomo desintegrado!');
   Carbono.Free;
end;
```

Haga esto en una aplicación de Windows 3.x si quiere que el sistema operativo quede completamente colgado hasta que al átomo le apetezca desintegrarse. Incluso en Windows 95 y NT esta es un forma tonta de desperdiciar tiempo de procesamiento. Mientras espera por el átomo, nuestra aplicación no puede hacer nada más: ni escuchar lo que desesperadamente trata de decirle el usuario, ni extrapolar las combinaciones acertantes de la lotería para ayudarnos a hacernos ricos.

Dependemos de un átomo, y sólo el átomo sabe cuando cambiará su estado, ¿por qué no dejar que sea el átomo quien nos avise cuando se desintegre? Este tipo de distribución de tareas es precisamente la que propone la Programación Orientada a Objetos. He sugerido que el átomo nos avise, pero ¿qué es un aviso en programación? Bueno, cuando a uno lo avisan es para que hagamos algo, aunque ese "algo" sea no hacer nada. Y hacemos algo en programación cuando ejecutamos una rutina. Entonces, cuando el átomo se desintegre debe llamar a una rutina. ¿Cuál? Si queremos que nuestros átomos puedan utilizarse con flexibilidad, el átomo no tiene por qué saber de antemano qué rutina debe ejecutar. En consecuencia, el átomo necesita un puntero a función o a método para que le digamos qué tiene que hacer cuando se desintegre. A continuación propongo una solución:

```
type
   TAtomo = class;

   TEventoAtomo = procedure (Emisor: TAtomo) of object;

   TAtomo = class
      EventoDesintegracion: TEventoAtomo;
      // …
   end;
```

---

[13] Dust in the wind (Point of Know Return, Kansas, 1977)

Como estamos trabajando en un entorno orientado a objetos, he preferido utilizar un puntero a método. Se supone que cuando el átomo se desintegre, debe llamar al método asociado a su puntero *EventoDesintegracion*. Esta es la forma en que tenemos que programar el aviso de desintegración:

```
procedure TForm1.AvisoDesintegracion(Emisor: TAtomo);
begin
   ShowMessage('¡Atomo desintegrado!');
   Emisor.Free;
end;

procedure TForm1.CrearAtomoClick(Sender: TObject);
begin
   with TAtomo.Create(6, 14) do
      EventoDesintegracion := AvisoDesintegracion;
end;
```

Nos queda explicar algo: hemos dicho que, misteriosamente, el átomo pierde parte de su masa al cabo de cierto tiempo. Para la explicación anterior, no nos interesa qué mecanismo utiliza el átomo para esto. Pero como no me gusta dejar cabos sueltos, voy a mostrar una posible implementación ¡que también utiliza eventos!

```
type
   TAtomo = class
   private
      FCarga: Integer;
      FMasa: Integer;
      Temporizador: TTimer;
      procedure AvisoTemporizador(Sender: TObject);
   public
      EventoDesintegracion: TEventoAtomo;
      constructor Create(ACarga, AMasa: Integer);
      destructor Destroy; override;
      property Carga: Integer read FCarga;
      property Masa: Integer read FMasa;
   end;

constructor TAtomo.Create(ACarga, AMasa: Integer);
begin
   FCarga := ACarga;
   FMasa := AMasa;
   if (FCarga = 6) and (FMasa = 14) then
   begin
      Temporizador := TTimer.Create(nil);
      Temporizador.Enabled := False;
      Temporizador.Interval := 1000 + Random(1000);
      Temporizador.OnTimer := AvisoTemporizador;
      Temporizador.Enabled := True;
   end;
end;

destructor TAtomo.Destroy;
begin
   Temporizador.Free;
   inherited Destroy;
end;
```

```
procedure TAtomo.AvisoTemporizador(Sender: TObject);
begin
   Temporizador.Enabled := False;
   FCarga := FCarga - 2;
   if Assigned(EventoDesintegracion) then
      EventoDesintegracion(Self);
end;
```

Aunque no es frecuente que una aplicación real tenga que vigilar el comportamiento de un átomo, las consideraciones anteriores se aplican a cualquier programa que ofrezca una Interfaz Gráfica de Usuario (GUI). En estos sistemas, el objeto vigilado ¡es el usuario! Bueno, más bien la cola de mensajes (teclas, movimientos de ratón, etc.) que va generando el usuario con sus acciones.

Eventos vs redefinición de métodos virtuales

Un programador "orientado a objetos", pero que no esté acostumbrado a trabajar con componentes, habría "resuelto" el problema del aviso de desintegración creando, en la clase *TAtomo* un método virtual vacío:

```
type
   TAtomo = class
   protected
      procedure Desintegrado; virtual;
      // …
   end;

// …

procedure TAtomo.Desintegrado;
begin
end;

procedure TAtomo.AvisoTemporizador(Sender: TObject);
begin
   Temporizador.Enabled := False;
   FCarga := FCarga - 2;
   Desintegrado;
end;
```

En teoría, todo lo tenemos previsto para el átomo. Si necesitamos un átomo que avise cuando se desintegre, podemos *derivar una clase* a partir de la clase anterior y *redefinir* el método *Desintegrado*:

```
type
   TAtomoConAviso = class(TAtomo)
   protected
      procedure Desintegrado; override;
   end;
```

```
procedure TAtomoConAviso.Desintegrado;
begin
   ShowMessage('¡Atomo desintegrado!');
   Free;
end;
```

Este enfoque funciona si todos los átomos de nuestra aplicación lanzan un desesperado mensaje al desintegrarse. Pero si uno de nuestros átomos debe activar una bombilla, el otro bailar una jiga y el tercero formatear el disco duro de nuestro vecino, ¡tenemos que definir tres nuevas clases de átomos! Y estos son los inconvenientes de tener tantas clases:

- Sumando la declaración de la clase más la implementación del método, hay que teclear más.
- A nivel de implementación, redefinir una clase para modificar un solo método para crear solamente un objeto no es rentable: esto implica duplicar una Tabla de Métodos Virtuales en el segmento de datos estático. Una VMT real tiende a ser bastante grande, aún con las mejoras de los métodos dinámicos de Borland.
- Un *TAtomoConAviso* nace siendo un "átomo con aviso", crece como "átomo con aviso" y muere siendo lo mismo. No puede convertirse dinámicamente en un "átomo destrozador de discos".

Realmente, vale la pena crear la clase *TAtomoConAviso* si vamos a utilizar esta clase en varias aplicaciones, o en diferentes contextos de una misma aplicación. Delphi siempre nos deja esta puerta abierta, permitiéndonos redefinir la clase, o incluso crear un nuevo componente.

UNA FABULA

Había una vez un rey que se preocupaba tanto por el bienestar de sus súbditos, que prohibió el uso de cuchillos, tijeras y cualquier objeto afilado, con tal de que nadie en su reino pudiera hacerse pupa, siquiera accidentalmente. Como resultado, la plebe no podía sacrificar ganado, cortar los filetes, arreglarse el pelo, y en pocos meses el que no había muerto de hambre, se había vuelto loco.

Uno de los muchos motivos por los que detesto a Java (sí, estaba pensando en Java) es que no ofrece punteros, mucho menos punteros a métodos. ¿Resultado? Que para soportar un simple evento (Java Beans) hacen falta objetos intermedios y un mecanismo complicado hasta más no poder. Todo esto significa lentitud en ejecución y dificultad para el mantenimiento. Súmele a esto el que la máquina virtual es más lenta que los tribunales de justicia, y los retrasos que causa la recogida de basura en aplicaciones serias. Si no le basta, le propongo un acertijo orientado a objetos: meto un perro en una colección de animales, y cuando lo saco es solamente un animal abstracto (no puede menear la cola).

El emisor y el receptor del evento

En la comunicación entre objetos mediante eventos podemos siempre identificar dos participantes: el emisor del evento y el receptor. El emisor del evento es el objeto que produce el evento, o en términos de implementación, el que llama al método asociado con el puntero del evento, si es que este puntero es distinto de **nil**. El receptor, en cambio, es un método dentro de cierto objeto; no es sencillamente un procedimiento aislado, ni un objeto en general.

Cuando Delphi crea automáticamente una respuesta a un evento perteneciente a un componente, sitúa al receptor en el formulario donde se encuentra el componente; si el evento pertenece al propio formulario, el objeto receptor es el mismo emisor. Con la interfaz visual de Delphi no se pueden crear receptores de eventos que violen este convenio; para esto tenemos que inicializar la conexión emisor/receptor por medio de código.

El primer ejemplo de esta conexión manual es el tratamiento de eventos del objeto *Application*. Este objeto, global a la aplicación, tiene eventos útiles como *OnException*, para el tratamiento centralizado de excepciones, *OnIdle*, para aprovechar los intervalos en que el usuario de la aplicación está ocioso, y *OnHint*, que se dispara cuando el puntero del ratón pasa por encima de un componente que tiene una indicación asociada. El último evento se puede aprovechar para mostrar las indicaciones en una barra de estado. Como no podemos seleccionar al objeto *Application* en el Inspector de Objetos, ni buscar la página de Eventos y hacer un doble clic en el evento *OnHint*, tenemos que realizar nosotros mismos el equivalente de este proceso en el código fuente.

```
type
   TFormularioPrincipal = class(TForm)
      // …
      StatusBar1: TStatusBar;
      procedure FormCreate(Sender: TObject);
      // Declarado por Delphi
   private
      procedure MostrarAyuda(Emisor: TObject);
      // Creado por nosotros
   end;

procedure TFormularioPrincipal.MostrarAyuda(Sender: TObject);
begin
   StatusBar1.Panels[StatusBar1.Panels.Count - 1].Text :=
      Application.Hint;
end;

procedure TFormularioPrincipal.FormCreate(Sender: TObject);
begin
   // Conectar el evento al receptor
   Application.OnHint := MostrarAyuda;
   // En realidad: Self.MostrarAyuda
end;
```

Una situación diferente se presenta en las aplicaciones MDI. El desarrollo de este tipo de aplicaciones comienza por definir una ventana principal, en la cual se incluye un menú mínimo, con las opciones que son aplicables cuando no hay documentos abiertos. Por ejemplo, el menú *Fichero* puede tener solamente las opciones *Nuevo*, *Abrir* y *Salir*. El comando de menú *Abrir* corresponde a un componente *TMenuItem* nombrado *Abrir1*, con la siguiente respuesta asociada a su evento *OnClick*:

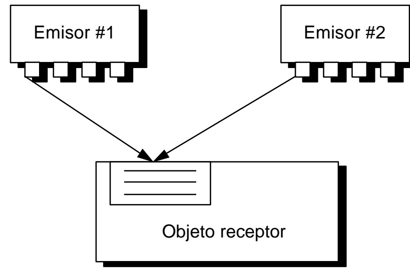
```
procedure TPrincipalMDI.Abrir1Click(Sender: TObject);
begin
   if OpenDialog1.Execute then
      // … abrir documento MDI …
end;
```

El próximo paso es crear la clase de las ventanas de documentos. Estas ventanas tienen un menú más detallado, con los comandos específicos para manipular documentos. El menú local se mezcla con el menú global de la ventana principal cuando la nueva ventana está activa. En el menú local necesitamos también un menú *Fichero* ampliado, que sustituye al *Fichero* global, y en este nuevo menú también tenemos un comando *Abrir*, cuya respuesta debe ser exactamente la misma que antes. ¿Qué hacemos, duplicamos el código de *Abrir1Click*? Existe una mejor solución, que es asociar al evento del menú local la respuesta existente en el formulario principal. Esto no se puede hacer en Delphi con el Inspector de Objetos, por lo que nuevamente realizamos el enlace emisor/receptor mediante instrucciones:

```
procedure TDocumentoMDI.FormCreate(Sender: TObject);
begin
   Abrir1.OnClick := PrincipalMDI.Abrir1.OnClick;
   // O, lo que es lo mismo:
   // Abrir1.OnClick := PrincipalMDI.Abrir1OnClick;
end;
```

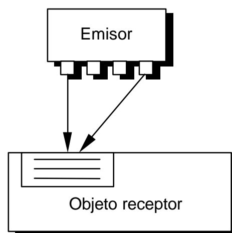
Receptores compartidos

El último ejemplo de la sección anterior muestra dos objetos emisores que dirigen sus eventos a un mismo receptor. Aunque en el ejemplo los dos objetos se encuentran en diferentes formularios, es mucho más común que los objetos se encuentren en la misma ficha, estando el receptor definido en la clase del formulario. La situación se puede describir mediante el siguiente diagrama:



Cuando se da este tipo de receptores compartidos, puede ser importante distinguir qué emisor ha producido el evento. Es por esto que casi todos los eventos de Delphi han sido diseñados con un parámetro *Sender*, para informar acerca del emisor. Este *Sender* se declara casi siempre de tipo *TObject*. De este modo, el receptor del evento *OnClick* de un comando de menú puede ser compartido con el evento *OnClick* de un *TSpeedButton* ó *TToolButton* (Delphi 3/4) colocado en una barra de herramientas. Las reglas de compatibilidad de tipos de clase exigen que, para que estos receptores puedan compartirse, el parámetro del emisor debe pertenecer a una clase común a todos los tipos de emisores. Los candidatos obvios son *TComponent* y *TObject*. Delphi se queda con este último.

Otro caso de receptor compartido se produce cuando dos eventos de un mismo componente apuntan al mismo procedimiento receptor. Por ejemplo, una tabla tiene los eventos *AfterPost*, que se dispara después de una inserción o modificación, y *AfterDelete*, que se produce después de una operación de borrado. Si queremos alguna acción que se produzca después de cualquier actualización sobre esta tabla, podemos compartir el receptor entre estos dos eventos. El siguiente esquema ilustra la situación:



En este caso, sin embargo, no tenemos forma de distinguir cuál ha sido el evento que ha provocado la ejecución del receptor. El parámetro *Sender*, en cualquiera de los dos casos, es el mismo objeto emisor.

Tipos de eventos

La mayoría de los eventos de Delphi se disparan por una de las siguientes razones:

- El emisor ha sufrido un cambio de su estado interno y desea avisarnos.
- El usuario ha realizado alguna acción. Windows notifica a la VCL con un mensaje. La VCL, a su vez, nos avisa con un evento.
- Dentro de un algoritmo interno del emisor hace falta tomar una decisión. El emisor pregunta a cualquier receptor interesado su opinión sobre el asunto antes de continuar.

El primer tipo de evento es típico de los componentes no visuales. Por ejemplo, las tablas y consultas (*TTable* y *TQuery*) utilizan cambios de estado para permitir las actualizaciones. Si queremos modificar los campos de una tabla debemos aplicar el método *Edit* sobre el objeto de tabla. La implementación de *Edit* es, a grandes rasgos, parecida a esto:

```
if Assigned(FBeforeEdit) then FBeforeEdit(Self);
// …
// Aquí la tabla se pone en el modo de edición
// …
if Assigned(FAfterEdit) then FAfterEdit(Self);
```

De este modo, un usuario interesado en controlar las veces en que una tabla es modificada puede interceptar estos eventos. En el evento *BeforeEdit*, por ejemplo, puede comprobar que se cumplan ciertas condiciones necesarias; de no ser así, puede utilizar el mecanismo de excepciones para abortar toda la operación (la instrucción **raise**).

Al segundo tipo de evento pertenecen los eventos de teclado y ratón: *OnKeyDown*, *OnMouseDown*, etc. Generalmente estos eventos tienen varios parámetros para indicar la tecla pulsada, el estado de las teclas de control, las coordenadas del ratón e información similar. Esta es, por ejemplo, la declaración del tipo de los eventos de ratón:

```
type
   TMouseEvent = procedure (Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer) of object;
```

Indirectamente, estos objetos disparan otra generación de eventos: *OnClick*, *OnDblClick*, *OnChange*, etc. Sin embargo, estos mismos eventos pueden producirse por cambios internos de estado. El evento *OnChange* de un cuadro de edición *TEdit* se dispara también cuando se le asigna algo a la propiedad *Text* del componente.

También pertenecen a este segundo grupo los eventos relacionados con los mensajes de ventana de Windows: *OnClose*, *OnActivate*, *OnResize*, que corresponden a los mensajes *WM_CLOSE*, *WM_ACTIVATE* y *WM_SIZE*.

Nos piden nuestra opinión...

La característica principal de la tercera categoría de eventos de la sección anterior es la presencia, en la declaración del tipo de evento, de parámetros pasados por referencia, con la palabra reservada **var**. El propósito de esta técnica es que el receptor pueda pasar información directamente al emisor. En este tipo de evento, el objeto que lo produce debe inicializar el parámetro por referencia con un valor determinado. De este modo, si no realizamos modificaciones en el parámetro, el objeto sigue su comportamiento habitual.

Un ejemplo típico es el evento *OnCloseQuery*, de los formularios:

```
type
   TCloseQueryEvent = procedure (Sender: TObject;
      var CanClose: Boolean);
```

La documentación de Delphi nos dice que *CanClose* trae *True* como valor predeterminado. Este evento se lanza mediante un código similar a grandes rasgos al siguiente:

```
var
   CanClose: Boolean;
begin
   // …
   CanClose := True;
   if Assigned(FOnCloseQuery) then FOnCloseQuery(Self, CanClose);
   if CanClose then
   // …
end;
```

Otro evento similar es *OnClose*. El parámetro *Action* de este evento permite personalizar la acción que se realiza cuando el usuario cierra una ventana:

```
procedure TDocumentoMDI.FormClose(Sender: TObject;
   var Action: TCloseAction);
begin
   // Inicialmente, Action vale caMinimize
   Action := caFree;
   // Se debe liberar la memoria de la ventana
end;
```

Los eventos expresan contratos sin obligaciones

¿Qué sucede si un usuario crea una respuesta a *OnCloseQuery* que no toca el parámetro *CanClose*, o que sencillamente no hace nada? Pues no pasa nada: si observamos el código de lanzamiento del evento, veremos que si el manejador no cambia el valor del parámetro, éste mantiene su valor inicial, *True*, y el algoritmo del emisor puede

continuar. Es más, si no hay receptor asociado al evento tampoco pasa nada, pues *CanClose* sigue siendo *True*.

Este comportamiento es generalizable como filosofía de diseño de los eventos. Cuando un objeto le ofrece a un programador un conjunto de eventos, está estableciendo un contrato bajo los siguientes términos:

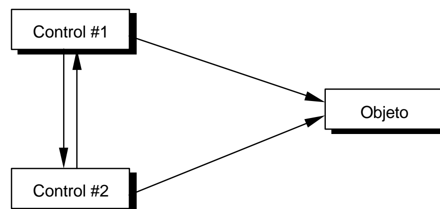
"Aquí te ofrezco estos eventos. Si los usas, bien; y si no, no pasa nada"

Esto contrasta con la filosofía de muchas bibliotecas de clases tradicionales, donde la personalización de un objeto se realiza redefiniendo un método virtual, y se exige que el programador llame obligatoriamente, en algún momento, al método heredado (**inherited**). Si esta llamada se omite, las consecuencias son impredecibles.

Activación recursiva

El gran problema de la programación con eventos es la posibilidad de disparar, durante el tratamiento de un evento, otros eventos secundarios inesperados. En particular, esto puede causar la activación recursiva de un manejador, con el riesgo de bucles infinitos y de desbordamientos de pila. En estas situaciones hay que controlar que los eventos se produzcan de modo exclusivo, de modo que si estamos dando respuesta a un evento, ignoremos el otro evento.

Este tipo de problemas se presenta con frecuencia cuando dos controles actúan sobre el estado de un mismo objeto, y tienen que estar sincronizados entre sí. El siguiente diagrama ilustra la situación:



Cuando el primer control cambie su valor, además de cambiar el estado del objeto, debe asignar un nuevo valor al segundo control. Esto provoca, para la mayoría de los controles, un segundo evento emitido esta vez por el segundo control. En esta respuesta se cambia el estado del objeto y se reajusta el valor del primer control ... y así *per secula seculorum*.

En un formulario vacío coloque los siguientes objetos; los componentes *TTrackBar* y *TUpDown* pueden encontrarse en la página *Win95* de la Paleta de Componentes:

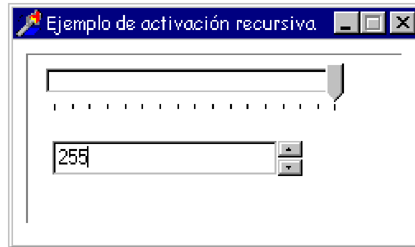
Objeto	Propiedad	Valor
TrackBar1: TTrackBar	*Min*	*0*
	Max	*255*
Edit1: TEdit		
UpDown1: TUpDown	*Min*	*0*
	Max	*255*
	Associate	*Edit1*

El objetivo de este programa es cambiar el color del fondo de la ventana a un gris cuya intensidad esté dada por el cuadro de edición o por la barra deslizante, que estarán sincronizados en todo momento. El usuario debe poder cambiar esta intensidad desde cualquiera de estos controles. El código necesario se muestra a continuación:

```
type
   TForm1 = class(TForm)
      // …
   private
      FCambiando: Boolean;
   end;

procedure TForm1.Edit1Change(Sender: TObject);
var
   Valor: Integer;
begin
   if not FCambiando then
   begin
      FCambiando := True;
      Valor := UpDown1.Position;
      Color := RGB(Valor, Valor, Valor);
      TrackBar1.Position := Valor;
      FCambiando := False;
   end;
end;

procedure TForm1.TrackBar1Change(Sender: TObject);
var
   Valor: Integer;
begin
   if not FCambiando then
   begin
      FCambiando := True;
      Valor := TrackBar1.Position;
      Color := RGB(Valor, Valor, Valor);
      UpDown1.Position := Valor;
      FCambiando := False;
   end;
end;
```

La variable *FCambiando* se utiliza como un semáforo de exclusión mutua (*mutex*). Cada respuesta a evento comprueba que no existe alguna otra respuesta pendiente en la pila de ejecución examinando esta variable. Cuando el método obtiene la luz verde,

asigna *True* a *FCambiando* para señalar su presencia. Al terminar, está obligado a restablecer la luz verde, asignando *False* al semáforo.



La función *RGB*, utilizada antes, pertenece al API de Windows y calcula un color a partir de las intensidades de sus componentes rojo, verde y azul. Si los colores básicos se mezclan en igual proporción, se obtienen los distintos tonos de gris.

Capítulo

11

Excepciones

SI CON LA PROGRAMACIÓN ORIENTADA A OBJETOS aprendimos a organizar nuestros datos y algoritmos, si con los Sistemas Controlados por Eventos nos liberamos de la programación "posesiva", con las Excepciones daremos el último paso necesario: aprenderemos a ver a un programa como un proceso que se desarrolla en *dos* dimensiones. Y nos liberaremos de los últimos vestigios de la neurosis "yo lo controlo todo en mi programa", un paso importante hacia la Programación Zen.

Sistemas de control de errores

Errar es de humanos, pero para meter de verdad la pata hace falta un ordenador. O, como dice la famosa ley de Murphy: todo lo que puede fallar, falla. Acabamos de comprar una biblioteca de funciones para trabajar, digamos, con la nueva NDAPI (*Nuclear Devices Application Programming Interface*). Por motivos de seguridad, en adelante utilizaremos nombres falsos y cortos para las funciones de la biblioteca; también omitiremos los parámetros. Supongamos que cierto proceso que está programando requiere una secuencia de llamadas a rutinas:

```
A; B; C; D; E
```

Ahora bien, cada una de estas rutinas puede fallar por ciertos y determinados motivos. Para controlar los errores cada rutina es una función que devuelve un código entero representando el estado de error. Supondremos para simplificar que 0 significa ejecución exitosa. Este mecanismo es equivalente a toda una amplia gama de técnicas de notificación de errores: variables de estado global, parámetros de estado, etcétera, por lo cual nuestra suposición no quitará generalidad a nuestras conclusiones. Partiendo de estas premisas, ¿cómo hay que modificar el código anterior para controlar los posibles errores? La respuesta es la siguiente:

```
Result := -1;
if A = 0 then
   if B = 0 then
      if C = 0 then
         if D = 0 then
```

```
                    if E = 0 then
                       Result := 0;
```

Observe que este código, a su vez, debe formar parte de una función o procedimiento, este último creado por nosotros. Y esta rutina hereda el carácter "falible" de las funciones que invoca, por lo que hay que devolver también un valor para indicar si nuestra función terminó bien o mal. En este caso, hemos simplificado el algoritmo, devolviendo únicamente 0 ó -1 para indicar éxito o fracaso.

Un programador con un mínimo de picardía puede pensar en la siguiente alternativa al código anterior:

```
if (A = 0) and (B = 0) and (C = 0) and (D = 0) and (E = 0) then
   Result := 0
else
   Result := -1;
```

No está mal, se parece un poco al estilo de programación de Prolog[14]. Pero la función *A* levanta la tapa de un contenedor de sustancias radiactivas y, pase lo que pase, hay que llamar a la función *E*, que cierra el depósito, si *A* terminó satisfactoriamente, ¡y sólo en ese caso, o aténgase a las consecuencias! Está claro que por este camino vamos al holocausto nuclear.

En la práctica, cuando el tema de programación no es tan dramático, lo que sucede es que el código de error no se verifica en todos los momentos necesarios. Es frecuente ver a un programa herido de muerte tambalearse por todo el escenario derribando muebles y tropezando con el resto de los actores. A todo esto nos ha conducido la filosofía "clásica" de verificación de errores.

Contratos incumplidos

La falta de precisión es un pecado, ¿qué queremos decir exactamente con aquello de que "una rutina puede fallar"? Para entendernos vamos a adoptar una metáfora de la programación conocida como *programación por contrato*. Según este punto de vista, una llamada a una rutina representa un contrato que firmamos con la misma. En este contrato hay dos partes: el contratador, que es la rutina donde se produce el llamado, y el contratado, que es la rutina a la cual se llama. El contratador debe proveer ciertas condiciones para la ejecución del contrato; estas condiciones pueden consistir en el estado de ciertas variables globales a la rutina llamada o los parámetros pasados a la misma. Por su parte, el contratado se compromete a cumplir ciertos y determinados objetivos. Es el fallo en el cumplimiento de estos objetivos, o la detección de incumplimientos por parte del contratador, lo que puede causar un error.

---

[14] ¿Hay alguien que aún recuerde el *boom* de la Programación Lógica y de los Lenguajes Funcionales?

El problema de los sistemas tradicionales de control de errores es que el incumplimiento de un contrato por parte de una rutina no utiliza ningún mecanismo especial de señalamiento. Para detectar un contrato incumplido hay que utilizar las mismas instrucciones de control condicionales que cuando se cumplen los contratos, mezclándose las situaciones en que van bien las cosas y las situaciones en que van mal. Por este mismo motivo, un contrato incumplido puede pasar desapercibido. Podemos resumir las condiciones mínimas que hay que exigir a una política correcta de control de errores en estos puntos:

- Si quiero llamar sucesivamente a tres funciones, debo poder hacerlo sin mezclar condicionales que contaminen la lógica de mi algoritmo. Debo poder mantener la linealidad del algoritmo original.
- Un error no puede pasar desapercibido.
- Deben respetarse las *condiciones de clausura*: toda puerta abierta debe cerrarse, aún en caso de fallo.
- Los errores deben poder corregirse: un contrato incumplido no debe causar inevitablemente la caída del sistema.

Cómo se indica un error

Entonces vienen las excepciones al rescate. Con nuestro nuevo sistema, cada vez que una función contratada detecta que no puede cumplir con sus obligaciones, produce una excepción. Para producir la excepción se utiliza la instrucción **raise**, que significa "elevar" o "levantar". Esta instrucción interrumpe la ejecución del programa y provoca la terminación de todas las funciones pendientes, a menos que se tomen medidas especiales. Al ejecutar la excepción hay que asociarle un objeto que porte la información acerca de la situación que la ha provocado. Un ejemplo típico de instrucción para elevar una excepción es el siguiente:

```
raise Exception.Create('¡Algo va mal!');
```

Exception es una clase definida por Delphi, aunque su nombre no comience con la letra *T* (es precisamente la excepción del convenio). En la instrucción anterior se crea "al vuelo" un nuevo objeto de esta clase utilizando el constructor *Create*. Este constructor tiene un único parámetro, el mensaje explicativo de las causas que motivaron la excepción. El mensaje se almacena en la propiedad *Message* del objeto de excepción construido.

Supongamos que necesitamos una función que divida dos números entre sí. Contratamos la función pasándole un par de enteros, y nos comprometemos a que el divisor que suministramos es distinto de cero. Utilizando el nuevo mecanismo de señalización de fallos, podemos realizar esta implementación:

```
function Dividir(Dividendo, Divisor: Integer): Integer;
begin
   if Divisor = 0 then
   // Se ha detectado una violación de las cláusulas del contrato
      raise Exception.Create('División por cero');
   Result := Dividendo div Divisor;
end;
```

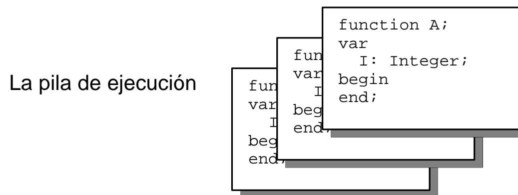
Si en algún momento se llama a esta función con un divisor igual a cero, se ejecutará la instrucción **raise**. Esta instrucción interrumpe el flujo normal de la ejecución del programa: las instrucciones siguientes no se ejecutarán y el programa, si no se toman medidas especiales, debe terminar.

> He utilizado como ejemplo la división de dos enteros por simplicidad. En realidad, no hay que verificar explícitamente que el divisor sea distinto de cero, pues Delphi se encarga de la verificación y lanza una excepción. Eso sí, el objeto de excepción que se construye pertenece a la clase *EDivByZero*, que desciende de nuestra *Exception*.

La ejecución del programa fluye en dos dimensiones

Voy a recordarle al lector un hecho que, aunque conocido, se olvida con facilidad. Y es que la ejecución de un programa se puede visualizar como un flujo bidimensional. Para poner un ejemplo inmediato, considere cualquiera de los fragmentos de código del epígrafe anterior. Dentro de cada secuencia, descontando los saltos hacia adelante de las condicionales y los saltos hacia atrás de los bucles, las instrucciones siguen una línea de ejecución. ¿Qué sucede cuando se ejecuta la última línea de la secuencia? ¿Caemos al vacío, como imaginaban los medrosos marineros medievales que les sucedía a las embarcaciones que llegaban al fin del mundo?

Pues no: a la función o procedimiento actual la llamó, en su momento, alguna otra rutina, a la cual se regresa entonces. Y a esta rutina, a su vez, la debe haber llamado alguna otra, y así sucesivamente hasta llegar al punto de inicio del programa. La estructura de datos de la aplicación que gestiona esta lista de rutinas pendientes de terminación es la *pila de ejecución* del programa.



Toda cadena es tan fuerte como su eslabón más débil. Si una rutina llama a rutinas propensas a fallar, esta propia rutina es una potencial incumplidora de contratos. Si una de las subrutinas llamadas por esta rutina eleva una excepción, la rutina debe también interrumpir su ejecución.

El estado de pánico

Uno de los conceptos principales para comprender las excepciones es que en todo momento el programa está en uno de dos estados posibles: el estado que podemos catalogar de "normal" y el estado de excepción, al cual también podemos llamar "estado de pánico". En este último estado se produce la interrupción en cadena de las rutinas pendientes en la pila del programa. No existe nada parecido a una excepción sobre otra excepción: o el programa se está ejecutando normalmente o está interrumpiendo rutinas.

Una de las consecuencias de la existencia de sólo dos estados de la ejecución para un programa es la posibilidad de utilizar una variable global para almacenar la información sobre la posible excepción activa. El objeto de excepción se puede obtener en cualquier momento por medio de la función *ExceptObject*. De todos modos, un poco más adelante veremos técnicas más adecuadas para trabajar con este objeto.

Pagamos nuestras deudas

Todo parece indicar que nuestro "revolucionario" sistema para tratamiento de errores se reduce a interrumpir simplemente el funcionamiento del programa. Según esto, el uso de la instrucción **raise** produce casi el mismo efecto que el procedimiento *Halt* de Turbo Pascal. Si piensa así, se equivoca.

En la descripción de cómo una instrucción **raise** interrumpe la ejecución de un programa hay un detalle importante: la ejecución va interrumpiendo una a una cada función pendiente en la pila de la aplicación. Y el sentido de esta conducta es la búsqueda de posibles instrucciones de manejo de excepciones. Tenemos, en primer lugar, la instrucción **try**…**finally**, cuya sintaxis es la siguiente:

```
try
   // Instrucciones …
finally
   // Más instrucciones …
end;
```

Si todo va bien durante la ejecución de la instrucción, se ejecutan tanto las instrucciones de la parte **try** como las de la parte **finally**, esto es, se puede leer la instrucción ignorando las tres palabras reservadas. En cambio, si se produce una excepción

durante la ejecución de la parte **try** (que quiere decir "intentar" en inglés) se interrumpe la ejecución del resto de esta parte, pero se ejecuta entonces la parte **finally**. En este último caso, al terminar las instrucciones **finally** la excepción sigue propagándose. Por lo tanto, la instrucción **try**…**finally** no nos permite resolver el problema ocasionado por el incumplimiento de un contrato; nos deja, a lo sumo, pagar a los trabajadores por el trabajo realizado, pero el programa sigue en el modo de pánico.

El concepto de "recurso de programación"

A la instrucción **try**…**finally** también se le conoce como el "bloque de protección de recursos". Por recurso de programación se entiende cualquier entidad cuya petición deba ir acompañada de una devolución. Un fichero abierto es un recurso, pues su apertura debe ir emparejada con su cierre, más tarde o más temprano. Un objeto temporal en Delphi es otro ejemplo, pues su construcción está balanceada por su destrucción. También son recursos, desde este punto de vista, un bloqueo (*lock*), una tabla abierta, una transacción de bases de datos iniciada, un semáforo para el control de concurrencia…

¿En qué sentido **try**…**finally** protege recursos? El código típico de trabajo con un fichero de texto en Delphi se debe escribir, aprovechando el tratamiento de excepciones, de la siguiente forma:

```
var
   Txt: TextFile;
begin
   AssignFile(Txt, 'NombreDeFichero');
   // Esta instrucción abre el fichero
   Reset(Txt);
   try
      // Instrucciones que trabajan con el fichero abierto
      // y que pueden provocar una excepción
      // …
   finally
      // Esta instrucción cierra el fichero siempre
      CloseFile(Txt);
   end;
end;
```

La pregunta más frecuente que surge, a la vista del código anterior, es: ¿por qué está la llamada al procedimiento *Reset* fuera de **try**, sin protección? La respuesta es simple: si colocáramos la llamada a *Reset* dentro de la instrucción **try**, un fallo en la apertura del fichero provocaría la ejecución de la parte **finally** de la instrucción, que intentaría cerrar un fichero que no hemos logrado abrir. Este ejemplo de programación se puede generalizar al uso de recursos arbitrarios:

```
// Instrucción que pide un recurso
try
   // Instrucciones que trabajan con el recurso
finally
   // Instrucción que devuelve el recurso
end;
```

Incluso el cambio de cursor durante una operación larga puede considerarse como un caso especial de gestión de recursos. Una vez cambiado el cursor, tenemos que garantizar el retorno a su imagen normal, pase lo que pase en el camino. El siguiente fragmento de código muestra cómo cambiar temporalmente el cursor de la aplicación durante una operación de este tipo:

```
Screen.Cursor := crHourGlass;
try
   // Una operación larga que puede fallar
finally
   Screen.Cursor := crDefault;
end;
```

Cómo tranquilizar a un programa asustado

Si solamente tuviéramos las instrucciones **raise** y **try**...**finally** para el manejo de excepciones, ya hubiéramos logrado algo: la posibilidad de crear programas que fallaran elegantemente. Cuando ocurriese un error en estos programas, el usuario se encontraría de nuevo en el Escritorio de Windows, pero con la seguridad de que todos sus ficheros abiertos se han cerrado y que no se han quedado recursos asignados en el servidor de la red. ¡Menudo consuelo!

Hace falta algo más, hace falta poder corregir el error. Y este es el cometido de la instrucción **try**…**except**. En su forma más simple, la sintaxis de esta instrucción es la siguiente:

```
try
   // Instrucciones …
except
   // Más instrucciones …
end;
```

Analizaremos la instrucción considerando los dos casos posibles: falla alguna de las instrucciones protegidas dentro de la parte **try** o todo va bien. Si todo va bien, se ejecutan solamente las instrucciones de la parte **try**. En este caso, el programa salta por encima de la cláusula **except**. Pero si se produce una excepción durante la ejecución del **try**, el programa salta directamente a la parte **except**, a semejanza de lo que ocurre en **try**...**finally**. En contraste, una vez finalizada la parte **except** de la instrucción, la ejecución continúa en la instrucción siguiente a la instrucción

try...**except**: hemos logrado que el programa retorne a su modo de ejecución normal.

Ejemplos de captura de excepciones

Si consideramos las excepciones como modo de señalar el incumplimiento de un contrato, podemos ver a la instrucción **try**…**except** como una forma de reintentar el contrato bajo nuevas condiciones. Tomemos por caso el código para la apertura de un fichero. Muchas veces, si necesitamos el fichero para escribir sobre él, desearemos crear el fichero si no existe. Para ello, podemos modificar el algoritmo de la siguiente forma:

```
AssignFile(Txt, 'NombreDeFichero');
try
   // Intentar abrir el fichero
   Reset(Txt);
except
   // Crear el fichero y abrirlo
   Rewrite(Txt);
end;
try
   // Instrucciones que trabajan con el fichero abierto
finally
   CloseFile(Txt);
end;
```

Observe que la creación del fichero con *Rewrite* sigue estando fuera de protección. Más adelante explicaremos el motivo de este "descuido".

Otra posibilidad es reintentar la apertura cíclicamente, pues el fichero puede estar bloqueado por otra aplicación, puede residir en un disquete que hay que insertar o estar ubicado en un servidor al cual no nos hemos conectado. En este caso, hay que programar manualmente un ciclo de reintentos; no hay instrucción en Delphi que lo haga de forma automática:

```
Ok := False;
while not Ok do
   try
      Reset(Txt);
      Ok := True;
   except
      if MessageDlg('No puedo abrir fichero'#13 +
         '¿Reintentar apertura',
         mtError, [mbYes, mbNo], 0) <> mrYes then raise;
   end;
```

He utilizado una nueva variante de la instrucción **raise** dentro de la cláusula **except**. Dentro de las cláusulas **except** podemos escribir esta instrucción sin especificar el objeto de excepción. En este caso, si el usuario desiste de la apertura del fichero de-

jamos la excepción sin tratar repitiéndola. ¿Por qué repetimos la excepción, en vez de dar un mensaje aquí mismo y dar por zanjado el tema? Dentro de muy poco daremos respuesta a esto.

Capturando el objeto de excepción

Antes hemos visto que el objeto lanzado con la instrucción **raise** se podía recuperar mediante la función *ExceptObject*. Aunque *ExceptObject* retorna un valor de tipo *TObject*, y la instrucción **raise** no requiere que los objetos lanzados mediante ella pertenezcan a alguna clase en especial, la práctica recomendada es que los objetos de excepción pertenezcan a la clase *Exception* o a clases derivadas de la misma.

Si necesitamos el objeto de excepción dentro de una cláusula **except** la mejor forma de obtenerlo es utilizando la siguiente sintaxis:

```
try
   // Instrucciones
except
   on E: Exception do
      ShowMessage(E.Message);
end;
```

La variable local *E* puede ser utilizada en la instrucción asociada a la cláusula **on**. Si necesitamos dos instrucciones en esa cláusula, tenemos que utilizar una instrucción compuesta, como es típico en Pascal:

```
try
   // Instrucciones
except
   on E: Exception do
   begin
      // Lista de instrucciones
   end;
end;
```

Distinguir el tipo de excepción

La sintaxis completa de la instrucción **try**…**except** es un poco más complicada, pues también podemos distinguir directamente la clase a la que pertenece la excepción. Para esto hay que utilizar varias cláusulas **on** dentro de la sección **except** de la instrucción de captura de excepciones. Por ejemplo:

```
try
   // Instrucciones
except
   on E: EZeroDivide do
      ShowMessage('División por cero');
```

```
      on EOutOfMemory do
         ShowMessage('Memoria agotada');
   else
      ShowMessage('Fallo de excepción general');
   end;
```

En este ejemplo he mostrado varios estilos de discriminación entre excepciones. La primera cláusula **on** permite trabajar en su interior con el objeto de excepción, nombrado *E*, aunque en este caso no se ha aprovechado el objeto para nada. En la siguiente cláusula **on**, en donde se capturan las excepciones de tipo *EOutOfMemory*, no se declara una variable para recibir el objeto de excepción; si no nos interesa trabajar con dicho objeto, es perfectamente posible. Por último, he ilustrado el uso de una sección **else** dentro de **try**…**except**, para capturar las restantes instrucciones. Si usamos una cláusula **else** perdemos la posibilidad de trabajar con el objeto de excepciones directamente. La mejor alternativa es utilizar una cláusula **on** con el tipo general *Exception* al final de la sección **except**:

```
while True do
   try
      Reset(Txt);
      Break;
   except
      on E: EInOutError do
         if MessageDlg('¿Reintentar?', mtError, [mbYes, mbNo], 0)
            <> mrYes then raise;
      on E: Exception do
         begin ShowMessage(E.Message); Break; end;
   end;
```

Hay que tener cuidado con el orden de las cláusulas **on**. Si en el algoritmo anterior invertimos el orden y capturamos primero las excepciones de tipo *Exception* nunca recibiremos excepciones en la segunda cláusula, ¡pues todas las excepciones son de tipo *Exception*!

Una última regla relacionada con **try**…**except**: si no escribimos una sección **else**, ni una cláusula **on** para el tipo *Exception* y además se produce una excepción no contemplada en la instrucción, esta excepción no se captura, por lo cual sigue propagándose por la pila de ejecución de la aplicación. La forma correcta de programar el ejemplo anterior sería la siguiente:

```
while True do
   try
      Reset(Txt);
      Break;
   except
      on E: EInOutError do
         if MessageDlg('¿Reintentar?', mtError, [mbYes, mbNo], 0)
            <> mrYes then raise;
   end;
```

Las tres reglas de Marteens

He dejado a oscuras hasta el momento, intencionalmente, un par de puntos importantes. En primer lugar, ¿por qué, en el algoritmo de "apertura o creación de ficheros" dejamos sin proteger la llamada a la rutina de creación? ¿O es que no nos preocupa que ésta falle? ¿Por qué, en segundo lugar, cuando el usuario decide no reintentar la apertura del fichero volvemos a lanzar la excepción original?

Cuando imparto cursos de Delphi a personas que nunca han trabajado con excepciones, me gusta exponer un conjunto de tres reglas simples para la correcta aplicación de esta técnica. Son reglas pensadas fundamentalmente para el programador novato; un programador avanzado descubrirá numerosas excepciones, sobre todo a la tercera regla. Hay una regla por cada tipo de instrucción, y cada regla se refiere al uso y frecuencia que se le debe dar a la misma. Estas son las reglas:

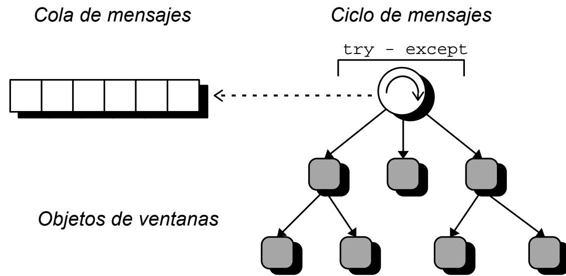
- ➊ (La regla de **raise**) Utilice **raise** con toda naturalidad, cada vez que quiera evitar que el algoritmo siga su ejecución lineal, cada vez que detecte una situación que no puede manejar en ese momento. En pocas palabras, cada vez que se le antoje.
- ➋ (La regla de **try**…**finally**) Utilice **try**…**finally** cada vez que pida un recurso, para asegurar su devolución. Esta es la regla de la "honestidad".
- ➌ (La regla de **try**…**except**) Utilice **try**…**except** lo menos posible; en la práctica, casi nunca.

La tercera regla, como se puede ver, es paradójica. Es fácil comprender la primera: la invocación a **raise** es el reconocimiento, por parte nuestra, de la imposibilidad de cumplir el objetivo, o contrato, de la rutina. Todo lo más, nos queda la duda de que nuestro programa pueda funcionar con un uso tan "alegre" de esta instrucción. Por otra parte, la segunda regla es la que garantiza la robustez y estabilidad del programa: todos los ficheros abiertos se cierran siempre, toda la memoria pedida para objetos temporales se libera automáticamente. Pero, ¿es que nunca debemos plantearnos, como pretende la tercera regla, reintentar un contrato incumplido? ¿No estaremos programando aplicaciones que se rinden al enfrentarse al primer problema?

Si estuviéramos escribiendo aplicaciones estilo UNIX/MS-DOS con interfaz de terminal, todas estas dudas serían correctas. Pero el hecho es que estamos programando aplicaciones para Sistemas Controlados por Eventos, y las suposiciones sobre el tipo de arquitectura de estos sistemas hacen variar radicalmente las condiciones de tratamiento de excepciones. La respuesta a los interrogantes anteriores es: no necesitamos instrucciones de captura de excepciones a cada paso porque ya tenemos una en el punto más importante del programa, en el ciclo de mensajes.

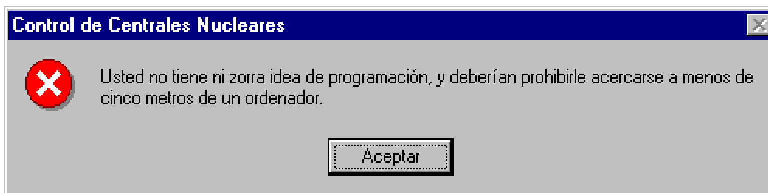
Ciclo de mensajes y manejo de excepciones

El punto más sensible de un programa en Windows es el ciclo de mensajes, y es ahí donde se ha colocado el manejador de excepciones **try**...**except** que acabamos de mencionar. La ejecución de una aplicación controlada por eventos se puede resumir en lo siguiente: mientras haya mensajes que procesar, procesémoslos.



Hay un teorema fundamental para estas aplicaciones: la respuesta a un mensaje o evento es virtualmente independiente de la respuesta al próximo mensaje. Por lo tanto, un incumplimiento en un contrato durante el procesamiento de un mensaje solamente debe causar la interrupción de ese proceso, pero el próximo mensaje no debe verse afectado por tal situación.

¿Qué hace la cláusula **except** del gestor central de excepciones? Muy sencillo: captura el objeto lanzado por la excepción y presenta al usuario el mensaje que contiene en su propiedad *Message*. El método *ShowException* del objeto global *Application* muestra un cuadro de diálogo como el de la figura que muestro a continuación. Un poco más adelante veremos que existe una importante excepción a este comportamiento, y que podemos personalizar la forma en que se presenta el mensaje de la excepción.



La explicación de la tercera regla de Marteens es ahora evidente: si, en un sistema controlado por eventos, se produce un fallo durante el procesamiento de un mensaje y no está en nuestras manos cambiar las condiciones del contrato para reintentar procesar dicho mensaje, lo más sensato es abortar completamente la rama de ejecución actual y pasar al procesamiento del siguiente mensaje. El destino de una excepción que señala el incumplimiento de un contrato debe ser su extinción en el **try**…**except** del ciclo de mensajes. Si antes de llegar a ese punto colocamos una

instrucción de este tipo y finalizamos sin cumplir el contrato, estamos defraudando al sistema, lo cual puede ser peligroso.

Para demostrarlo, vamos a suponer que, en el algoritmo de "apertura o creación" colocamos otra instrucción **try**…**except** para proteger también a la llamada a *Rewrite*:

```
try
   Reset(Txt);
except
   try
      Rewrite(Txt);
   except
      // ¿¿¿???
   end;
end;
```

He puesto a propósito signos de interrogación en el **except** interno. ¿Qué hacemos en ese punto, llamamos a *ShowMessage*? Bueno, eso ya se hace cuando se captura la excepción en el ciclo de mensajes. Casi siempre algún avispado sugiere entonces transformar nuestra rutina en una función y devolver un error si no podemos abrir ni crear el fichero … y entonces muestro mi mejor sonrisa irónica: ¿no estamos regresando a la filosofía de tratamientos de errores que tanto criticamos al comienzo del capítulo?

Lo dice la experiencia: cuando el programador novato siente deseos irresistibles de utilizar un **try**…**except** es casi seguro que está tratando de convertir un error señalado por una excepción al antiguo sistema de códigos de errores. Por lo tanto, antes de utilizar esta instrucción debe pensarlo dos veces. Este es el sentido de la tercera regla.

Excepciones a la tercera regla de Marteens

Una regla para las excepciones debe tener sus propias excepciones, como es natural. El lector ya conoce una: cuando podemos reintentar el contrato que ha fallado bajo otras condiciones. Otro caso importante sucede cuando queremos realizar alguna labor de limpieza, al estilo de **try**…**finally**, pero que se ejecute solamente en caso de excepción. Recuerde que la sección **finally** se ejecuta tanto si se producen excepciones como si no las hay. En tal situación se puede utilizar un **try**…**finally** de este modo:

```
try
   // Instrucciones
finally
   if ExceptObject <> nil then LaboresDeLimpieza;
end;
```

Pero es mucho más elegante utilizar el siguiente estilo:

```
try
   // Instrucciones
except
   LaboresDeLimpieza;
   raise;
end;
```

Este uso de **try**…**except** no viola el espíritu de las reglas de Marteens: la excepción sigue extinguiéndose en el ciclo de mensajes. La excepción original sencillamente se vuelve a repetir.

Sin embargo, puede haber un motivo más humano y cotidiano para utilizar un **try**…**except**: para transformar el mensaje de error asociado a la excepción. Puede ser que estemos trabajando con un Delphi en inglés y queramos mostrar los errores en castellano o swahili. Si este es su caso, llame a su distribuidor de software y pregúntele por los precios del Language Pack. Puede también, y esto sí es serio, que el mensaje original de la excepción sea muy general o muy técnico. Para el usuario final, hay que mostrar un mensaje más específico y explicativo. Por ejemplo, cuando en Delphi se intenta insertar un registro cuya clave está repetida, en pantalla aparece algo tan esclarecedor como *"Key violation"* o *"Violación de clave"*. Muy interesante para el usuario. En estos casos, una solución directa podría ser:

```
try
   // Instrucciones que pueden fallar
except
   raise Exception.Create('Este es mi nuevo mensaje');
end;
```

De todos modos Delphi nos ofrece, para la gestión de errores durante el procesamiento de bases de datos, técnicas mejores basadas en la intercepción de eventos, como *OnEditError* y *OnPostError*, que estudiaremos en su momento.

El evento OnException

Aunque la captura de la excepción sucede dentro de un procedimiento de la biblioteca VCL, podemos modificar este proceso gracias al evento *OnException* de la clase *TApplication*. El prototipo de este evento es el siguiente:

```
TExceptionEvent = procedure (Emisor: TObject; E: Exception)
   of object;
```

Si este evento tiene asignado un receptor, el tratamiento convencional de una excepción que llega al ciclo de mensajes no se produce. Este tratamiento consiste simplemente en mostrar el mensaje de la excepción en un cuadro de diálogo, como ya he-

mos visto. Podemos ser más imaginativos y utilizar un algoritmo alternativo para informar al usuario, quizás utilizando la tarjeta de sonido del sistema (no es broma). No obstante, una aplicación frecuente que se da a este evento es la traducción global de ciertos y determinados mensajes de excepción.

Para imitar el comportamiento normal de Delphi, la respuesta a este evento podría ser:

```
procedure TForm1.MostrarExcepcion(Sender: TObject; E: Exception);
begin
   Aplication.ShowException(E);
end;
```

Una sencilla aplicación de *OnException* es llevar en un fichero de texto el registro de todas las excepciones que llegan a ser visibles para el usuario de una aplicación. Vamos a plantearnos el ejemplo como una unidad que podamos incluir más adelante en cualquier proyecto nuestro. Inicie una aplicación y cree una unidad de código vacía, por medio del comando *File|New* y el icono *Unit*, del cuadro de diálogo que aparece. Guarde el fichero con el nombre que quiera darle a la unidad, y modifique su contenido de este modo:

```
unit ELog;
interface

procedure StartLog(const AFileName: string);
procedure StopLog;

implementation

uses SysUtils, Forms;

type
   TExceptionLog = class
   protected
      SavedHandler: TExceptionEvent;
      FileName: string;
      procedure ExceptionHandler(Sender: TObject; E: Exception);
   public
      constructor Create(const AFileName: string);
      destructor Destroy; override;
   end;

{ TExceptionLog }
// Aquí vendrá la implementación de la clase TExceptionLog

{ Interface procedures }

var
   Log: TExceptionLog = nil;
```

```
procedure StartLog(const AFileName: string);
begin
   if not Assigned(Log) then
      Log := TExceptionLog.Create(AFileName);
end;

procedure StopLog;
begin
   if Assigned(Log) then
   begin
      Log.Free;
      Log := nil;
   end;
end;

initialization
   // Nada
finalization
   StopLog;
end.
```

Esta unidad exportará los procedimientos *StartLog* y *StopLog*, para activar y desactivar la escritura de las excepciones a disco. La segunda función se llama automáticamente en la sección de finalización de la unidad. En cuanto a *StartLog*, puede activarse desde el código inicial del fichero *dpr*, o en cualquier otro momento, de esta manera:

```
   // …
   StartLog(ChangeFileExt(Application.ExeName, '.log'));
   // …
```

Cuando se activa el mecanismo, la unidad crea un objeto de tipo *TExceptionLog* y lo asigna a una variable interna. La idea es que este objeto actúe como receptor del evento *OnException* de la aplicación. Recuerde que un evento es un puntero a un método, y que por lo tanto necesita la presencia de un objeto. No se puede hacer que un evento apunte a un procedimiento global.

La implementación del constructor y del destructor de *TExceptionLog* es la siguiente:

```
constructor TExceptionLog.Create(const AFileName: string);
begin
   inherited Create;
   FileName := AFileName;
   SavedHandler := Application.OnException;
   Application.OnException := ExceptionHandler;
end;

destructor TExceptionLog.Destroy;
begin
   Application.OnException := SavedHandler;
   inherited Destroy;
end;
```

Por último, la implementación del método *ExceptionHandler* es todo un ejemplo de manejo de excepciones:

```
procedure TExceptionLog.ExceptionHandler(Sender: TObject;
   E: Exception);
var
   Txt: TextFile;
begin
   Application.ShowException(E);
   try
      AssignFile(Txt, FileName);
      try
         Append(Txt);
      except
         Rewrite(Txt);
      end;
      try
         WriteLn(Txt, FormatDateTime('hh:nn:ss', Now), ' ',
            E.ClassName, ': ', E.Message);
      finally
        CloseFile(Txt);
      end;
   except
   end;
end;
```

El método comienza mostrando la excepción al usuario. A continuación, se intentará la grabación de la excepción en el fichero. Pero como el Diablo carga los discos duros, tendremos sumo cuidado de no provocar una segunda excepción que nos lleve de forma recursiva hasta este mismo lugar, para lo cual encerramos todo el bloque de instrucciones dentro de un **try**/**except** con la cláusula de recuperación vacía (¡sí, estoy violando la 3ª Regla de Marteens!). A su vez, la apertura del fichero texto utiliza un **try**/**except** con dos alternativas: o el fichero se abre para escribir al final (*Append*) o se crea un nuevo fichero (*Rewrite*) ... o falla el algoritmo. La instrucción **try**/**finally** que sigue a continuación garantiza que el fichero abierto se cierre. Finalmente, escribimos una línea en el fichero que indique la hora actual, el nombre de la clase de excepción y el mensaje asociado. Eso es todo.

La excepción silenciosa

Delphi define un tipo especial de excepción cuya clase es *EAbort*. Desde el punto de vista del lenguaje, este es un tipo de excepción como cualquier otro. Pero la implementación de la Biblioteca de Controles Visuales le otorga un papel especial: si el ciclo de mensajes recibe esta excepción no se muestra mensaje alguno. Es tan frecuente su uso que la unidad *SysUtils* define un procedimiento, *Abort*, que la produce.

Volvamos al caso en que iniciábamos un ciclo de reintentos si fallaba la apertura de un fichero:

```
while True do
   try
      Reset(Txt);
      Break;
```

```
    except
      on E: EInOutError do
        if MessageDlg('¿Reintentar?', mtError, [mbYes, mbNo], 0)
          <> mrYes then raise;
    end;
```

La razón por la cual lanzamos un **raise** sin parámetros cuando no queremos reintentar es para no enmascarar la excepción y terminar la rama actual de ejecución dentro del árbol de llamadas. Sin embargo, póngase en el lugar del usuario, que recibe entonces la siguiente secuencia de mensajes:

```
El Ordenador: Ha fallado la apertura de manias.txt. ¿Reintentar?
El Usuario: Bueno, sí.
El Ordenador: Ha fallado nuevamente la apertura de manias.txt.
      ¿Reintentar?
El Usuario: Creo que no.
El Ordenador: Por tu culpa ha fallado la apertura de manias.txt.
```

El último mensaje sobra; lo único que queríamos era *abortar* la secuencia de funciones pendientes en la pila de llamadas, no culpabilizar al usuario de un error que ya conoce. La solución es, claro está, terminar con *Abort*, no con **raise**:

```
while True do
  try
    Reset(Txt);
    Break;
  except
    on E: EInOutError do
      if MessageDlg(E.Message + #13'¿Reintentar?',
        mtError, [mbYes, mbNo], 0) <> mrYes then Abort;
  end;
```

De este modo, cuando lanzamos una excepción después de mostrar un cuadro de mensajes, es conveniente utilizar *Abort*, para evitar la posible confusión del usuario al recibir múltiples mensajes.

Constructores y excepciones

Una característica de Object Pascal, por lo general ignorada por los programadores, es la destrucción automática cuando se produce una excepción durante la ejecución de un constructor. Supongamos que, en una aplicación MDI, tenemos ventanas hijas que muestran mapas de bits cargados desde ficheros gráficos. Para poder ser precisos, digamos que la clase de ventanas hijas se denomina *THija*, y que cada ventana de este tipo tiene un control de imágenes, *Image1*, en su interior:

```
type
  THija = class(TForm)
    // …
    Image1: TImage;
```

```
    // …
  end;
```

La ventana principal debe crear dinámicamente ventanas de este tipo, en respuesta al comando *Fichero|Abrir*; en realidad debe crearlas y además cargar el fichero dentro del control:

```
procedure TPrincipal.Abrir1Click(Sender: TObject);
var
   Hija: THija;
begin
   if OpenDialog1.Execute then
   begin
      Hija := THija.Create(Application);
      Hija.Image1.Picture.LoadFromFile(OpenDialog1.FileName);
   end;
end;
```

¿Qué sucede si no se puede cargar la imagen desde el fichero? Sucede que la ventana recién creada queda abierta, con su interior vacío. ¿Qué tal si intentamos cargar primero el fichero en una variable temporal, y luego creamos la ventana? Da lo mismo, porque la creación de la ventana también tiene razones para fallar y dejarnos con un objeto creado en memoria. Por supuesto, para solucionar estos problemas tenemos las instrucciones **try**. Pero podemos hacer algo mejor. ¿Por qué no reconocer que la creación de la ventana y la inicialización de su interior es un proceso atómico? Y digo atómico para indicar que se realizan las dos operaciones o no se realiza ninguna. Podemos crear un nuevo constructor en la clase *THija*:

```
constructor THija.CreateBmp(AOwner: TComponent;
   const AFileName: string);
begin
   inherited Create(AOwner);
   Image1.Picture.LoadFromFile(AFileName);
end;
```

y llamar a este constructor para crear la ventana:

```
procedure TPrincipal.Abrir1Click(Sender: TObject);
begin
   if OpenDialog1.Execute then
      THija.CreateBmp(Application, OpenDialog1.FileName);
end;
```

Aparentemente no hemos ganado nada, excepto desde el punto de vista filosófico. Y es que todavía no he explicado que los constructores definen en su interior una instrucción **try...except** implícita. El constructor anterior equivale aproximadamente al siguiente código:

```
constructor THija.CreateBmp(AOwner: TComponent;
   const AFileName: string);
begin
   try
      inherited Create(AOwner);
      Image1.Picture.LoadFromFile(AFileName);
   except
      Destroy;
      raise;
   end;
end;
```

Si se produce ahora un error durante la carga del fichero, se ejecuta automáticamente el destructor del objeto: un objeto se construye por completo o no se construye en absoluto. Esta característica de Delphi explica la insistencia en el uso de *Free* para liberar objetos en vez de la llamada directa a *Destroy*. Supongamos que el constructor de *THija*, en vez de cargar el fichero en un control *TImage*, crea un mapa de bits en memoria, un objeto de la clase *TBitmap*:

```
type
   THija = class(TForm)
      // …
      Bitmap: TBitmap;
      // …
   public
      constructor CreateBmp(AOwner: TComponent;
         const AFileName: string);
      destructor Destroy; override;
   end;

constructor THija.CreateBmp(AOwner: TComponent;
   const AFileName string);
begin
   inherited Create(AOwner);
   Bitmap := TBitmap.Create;
   Bitmap.LoadFromFile(AFileName);
end;
```

Bajo estas condiciones, puede surgir un nuevo problema. Si falla la lectura de fichero, se ejecuta el destructor, en el cual programaremos la destrucción de la variable *Bitmap*. ¿Y si falla la creación del propio objeto *Bitmap*? En ese caso, esta variable tendrá el valor inicial **nil**, y tenemos que cuidar que no se libere el objeto si encontramos este valor en *Bitmap*:

```
destructor THija.Destroy;
begin
   Bitmap.Free;
   // Es decir:
   // if Bitmap <> nil then Bitmap.Destroy;
   inherited Destroy;
end;
```

Existen casos en que sabemos que el objeto a destruir existe, como en secuencias como ésta:

```
with TLoQueSea.Create do
try
   // …
finally
   Destroy; // Claro que el objeto ha sido creado
end;
```

Aquí también tenemos una razón para sustituir la llamada a *Destroy* por una llamada a *Free*, aunque no tiene que ver con la seguridad del código. Como el lector sabe, una llamada a un método virtual, como *Destroy*, genera siempre más código que la llamada a un método que no lo es. Si queremos ahorrarnos un par de bytes, utilizamos entonces *Free*. Ahora bien, recuerde que, de este modo, estamos realizando una llamada adicional: los bytes que ahorramos los compensamos con un par de ciclos de reloj adicionales.

Aserciones

A lo largo de este capítulo he insistido en considerar el lanzamiento de una excepción como el reconocimiento por parte de una rutina, y de aquellas que la utilizan, de un fallo en el cumplimiento de un "contrato". En todos los ejemplos que hemos mostrado del uso de **raise**, este contrato es bien explícito: el contratado verifica recelosamente las condiciones en que se le encarga su trabajo. En ocasiones, sin embargo, existen suposiciones lógicas acerca del funcionamiento de un algoritmo que no necesitamos verificar a cada paso; tal comprobación, además, nos haría perder eficiencia. Un ejemplo trivial se produce en el recorrido de una lista enlazada:

```
type
   PListaEnlazada = ^TListaEnlazada;
   TListaEnlazada = record
      Info: Integer;
      Prox: PListaEnlazada;
   end;

procedure IncrementarContadores(L: PListaEnlazada);
begin
   while L <> nil do
   begin
      Inc(L.Info);    // Sólo en Delphi 1 es necesaria la flecha (^)
      L := L.Next;
   end;
end;
```

En realidad, la llamada al procedimiento *Inc* asume que la variable *L* es distinta del puntero vacío, y esta cláusula del contrato pudiera también hacerse explícita del siguiente modo:

```
if L = nil then
   raise Exception.Create('Puntero vacío');
Inc(L.Info);
```

Por supuesto, nadie programa así, pues en el contexto inmediato de la instrucción queda bien claro que el puntero *L* no puede estar vacío, gracias a la condición de parada del bucle. Sin embargo, en ejemplos reales puede que no estén tan claras las condiciones en que se ejecuta determinado algoritmo. Sospechamos que está bien programado, pero siempre nos queda aquella duda… ¿Por qué no establecer explícitamente nuestras "certezas", al menos mientras el programa esté en fase de desarrollo, y luego desactivarlas en la aplicación final?

La forma de realizarlo, en Delphi 3, es por medio de las *aserciones* (*assertions*). En esta versión, contamos con los siguientes procedimientos:

```
procedure Assert(Condicion: Boolean);
procedure Assert(Condicion: Boolean; const Mensaje: string);
```

El efecto de este procedimiento es aproximadamente el siguiente:

```
if not Condicion then
   raise EAssertionFailed.Create(Mensaje);
```

Digo "aproximadamente", porque este código se genera únicamente cuando la directiva de compilación *ASSERTIONS* ha sido activada:

```
{$ASSERTIONS ON}        ó        {$C+}
{$ASSERTIONS OFF}       ó        {$C-}
```

Esta directiva puede también controlarse en el diálogo *Project|Options*, en la página *Compiler*, opción *Assertions*.

El algoritmo anterior puede escribirse de esta manera:

```
while L <> nil do
begin
   Assert(L <> nil, '¡Paranoia!');
   Inc(L.Info);
   L := L.Next;
end;
```

Si utilizamos la variante de *Assert* que no especifica mensaje, la excepción se genera con un mensaje predefinido.

La teoría de la programación distingue unos cuantos tipos especiales de aserciones. Estos tipos son:

Precondiciones: Se sitúan al principio del código de la rutina, y verifican los valores de los parámetros y de las condiciones globales en que se ejecuta. Si fallan, es por responsabilidad del que llama a la rutina.

Postcondiciones: Se sitúan al final del código de la rutina, y comprueban que ésta ha cumplido su contrato.

Invariantes de clase: Este tipo de aserción debe cumplirse al finalizar cualquier método de una clase. Su programación en Delphi es bastante laboriosa.

Variantes de bucles: Comprueban que cierta expresión numérica asociada a un bucle siempre disminuya y sea mayor que cero: esta es la garantía de que el bucle finalice. Normalmente no se incluyen en Delphi, pues las expresiones pueden ser bastante rebuscadas y pintorescas.

Si quiere aprender más sobre las aserciones y sobre Programación Orientada a Objetos en general, le recomiendo uno de mis libros favoritos: *Object Oriented Software Construction*, de Bertrand Meyer (Prentice-Hall, 1988). En realidad, este libro describe una de las primeras versiones del lenguaje Eiffel, pero la forma en que justifica las técnicas orientadas a objetos lo convierte en un clásico de la literatura. Además, el autor concuerda conmigo en su aversión a ese lenguaje llamado C++.

Capítulo

12

Tipos de datos de Delphi

POR LA JUNGLA DE DELPHI vagan los más diversos tipos de datos. Algunos tipos de datos son "normales", en el sentido de que tienen equivalentes cercanos en otros lenguajes de programación, como los enteros, reales y valores lógicos. Algunos otros han sufrido "adaptaciones evolutivas" que los capacitan para la vida en la selva, como las cadenas de caracteres. Pero hay algunas piezas que se las traen, como los tipos variantes y los vectores abiertos. En este capítulo, intentaré presentar los tipos de datos de Delphi que satisfacen dos criterios: su carácter singular y su utilidad.

Tipos numéricos

¿He escrito "tipos singulares"? Realmente los tipos numéricos son de lo más normales. Pero Delphi ofrece una amplia variedad de ellos que puede confundir al programador. Y la migración del entorno de 16 bits al de 32 ha complicado un poco las cosas.

Comencemos con los tipos enteros. Delphi 1 ofrecía los siguientes tipos:

	8 bits	**16 bits**	**32 bits**
Con signo	*ShortInt*	*SmallInt*	*LongInt*
Sin signo	*Byte*	*Word*	No hay

Si el lector conoce algo de Pascal, puede sorprenderle la ausencia del famoso *Integer* y la presencia de un desconocido *SmallInt*. Esto se debe a que los tipos que he incluido en la tabla anterior tienen una implementación independiente de la plataforma. Borland ya estaba preparando la migración a 32 bits, y definió los siguientes tipos como dependientes de la plataforma:

- *Integer* = *SmallInt*
- *Cardinal* = *Word*

Delphi 2 y 3 mantienen los tipos de la primera tabla, cambiando solamente la definición de los tipos dependientes:

- *Integer* = *LongInt*
- *Cardinal* = *0..MaxLongInt*, donde *MaxLongInt* = 2^{31} - 1

Observe que, aunque uno esperaría un tipo *Cardinal* de 32 bits, Delphi ofrecía uno de 31 bits; nunca llegué a conocer la razón. Delphi 4, no obstante, ha introducido dos nuevos tipos de enteros, *Int64* y *LongWord*, y ha hecho equivalentes a los tipos *Cardinal* y *LongWord*, con lo que se ha restablecido la simetría en el sistema de tipos:

	8 bits	**16 bits**	**32 bits**	**64 bits**
Con signo	*ShortInt*	*SmallInt*	*LongInt*	*Int64*
Sin signo	*Byte*	*Word*	*LongWord*	No hay

- *Integer* = *LongInt*
- *Cardinal* = *LongWord*

En las versiones de 32 bits, las operaciones aritméticas se realizan directamente con instrucciones propias de la CPU. La versión de 16, sin embargo, podía en teoría producir aplicaciones para ser ejecutadas en los anticuados procesadores 80286. Una suma de dos valores de 32 bits se traducía en una llamada a una función interna. Si el procesador era un 80386 u otro más potente, la instrucción se pasaba directamente a la CPU; en caso contrario, la suma se realizaba mediante varias instrucciones de 16 bits. Un mecanismo ingenioso, pero ineficiente.

En contraste, los tipos reales casi no han cambiado desde las antiguas versiones de Turbo Pascal. Los tipos reales de Delphi son:

- *Single*: Ocupa 4 bytes. Rango: desde $1.5 * 10^{-45}$ hasta $3.4 * 10^{38}$.
- *Double*: Ocupa 8 bytes. Rango: desde $5.0 * 10^{-324}$ hasta $1.7 * 10^{308}$.
- *Extended*: Ocupa 10 bytes. Rango: desde $3.4 * 10^{-4932}$ hasta $1.1 * 10^{4932}$.

Estos tres tipos corresponden a tipos nativos de la FPU, que a su vez cumplen con el estándar definido por la IEEE. *Extended* es el tipo más completo, y toda la aritmética dentro de la FPU se efectúa en base a este formato, pues los otros valores se convierten antes de operar con ellos. Sin embargo, para la mayoría de las aplicaciones basta con utilizar *Double* para la declaración de variables, pues tiene la ventaja de adaptarse bien a la alineación por dobles palabras, que es la óptima para los procesadores de 32 bits de Intel.

El Pascal original, de Niklaus Wirth, utilizaba un único tipo para valores reales, *Real*. También Delphi define *Real*, pero hasta la versión 3 utilizaba un formato de 6 bytes. Este formato es propiedad de Borland y fue definido en las primeras versiones de Turbo Pascal, cuando no se había popularizado el uso del coprocesador 8087 de Intel. Las operaciones con este *Real* se realizaban completamente por software. Así que no era recomendable declarar variables de este tipo; incluso Borland desaconsejaba declarar propiedades *Real*. Delphi 4, finalmente, ha hecho que *Real* sea equivalente a *Double*, y ha declarado explícitamente un nuevo tipo, *Real48*, por compatibilidad con el pasado.

Otro tipo "exótico" es *Comp*. *Comp* es un formato nativo de la FPU, aunque se utiliza para representar enteros con signo de 64 bits. Su aplicación tradicional ha sido almacenar variables monetarias. No obstante, a partir de Delphi 2, Borland ha añadido un nuevo tipo al repertorio, *Currency*. Está basado en *Comp*, y reserva cuatro cifras para los decimales.

Fechas y horas

Las fechas y las horas se representan en Delphi mediante un mismo tipo de dato: el tipo *TDateTime*. Este tipo se declara como un *Double*, el valor real de 8 bytes de Pascal y C, y está definido, junto con sus operaciones, en la unidad *SysUtils*. La representación consiste en que la parte entera de este valor representa la fecha, y la parte fraccionaria la hora. La fecha se codifica como la cantidad de días transcurridos desde el 30 de diciembre de 1899. La elección de la fecha del *big-bang* de Delphi es, cuando menos, curiosa: el primer día es un 31 de diciembre, cuando más natural hubiera sido elegir un 1 de enero. Además, el siglo XX no comenzó en 1900, sino en 1901, un verdadero desliz cultural[15]. Esto, de todos modos, no nos afecta en nada. Incluso podemos utilizar el tipo *TDateTime* para representar fechas de siglos anteriores, siempre que la fecha sea posterior a la implantación del calendario gregoriano. La hora, en cambio, se representa como la fracción del día transcurrida. Por ejemplo, 0.5 significa las doce del mediodía.

Gracias a esta representación, la mayor parte de la aritmética de fechas puede realizarse directamente. Por ejemplo, se pueden comparar dos valores de tipo *TDateTime* para ver cuál es anterior o posterior:

```
if Frac(Now) > 0.5 then
   ShowMessage('¡Es hora de comer!');
```

---

[15] La culpa no es de Borland/Inprise sino ... ¿adivina quién es el culpable? En realidad, *TDateTime* imita al tipo de fechas de la Automatización OLE, que a su vez imitó a un tipo original de Visual Basic.

He utilizado la función *Now*, que da la fecha y la hora del sistema; para extraer la parte de la hora se emplea la función *Frac*, predefinida en Pascal. La función *Int*, en cambio, devuelve la parte entera como un valor real; es importante el tipo de valor que devuelve, pues si retorna un entero podemos tener problemas de desbordamiento.

También se pueden restar dos valores *TDateTime* para obtener la diferencia en días entre dos fechas:

```
if Round(Date - FechaNacimiento) mod 23 = 0 then
   MessageDlg('Hoy es el día crítico de su biorritmo'#13 +
      '¿Está seguro de que desea trabajar hoy?',
      mtWarning, mbYesNoCancel, 0);
```

La función *Date*, utilizada en el ejemplo anterior, retorna la fecha actual, sin la hora. En este caso ha sido necesario utilizar la función *Round* para obtener un entero, pues el operador **mod** de Pascal requiere que sus operandos pertenezcan a este tipo.

La facilidad en el uso de las fechas en este formato se paga con la complejidad presente en la conversión en ambos sentidos. Las funciones *EncodeDate* y *EncodeTime* se encargan de combinar los componentes de fecha y hora, y producir el valor resultante de tipo *TDateTime*:

```
function EncodeDate(Year, Month, Day: Word): TDateTime;
function EncodeTime(Hour, Min, Sec, MSec: Word): TDateTime;
```

Y todo hay que decirlo: la implementación de *EncodeDate* es algo chapucera, pues con un poco más de cuidado hubiera sido más eficiente; examine el código fuente si no me cree, y piense en lo fácil que sería eliminar el bucle **for** de la función.

La conversión en sentido contrario la realizan los métodos *DecodeDate* y *DecodeTime*:

```
procedure DecodeDate(Date: TDateTime; var Year, Month, Day: Word);
procedure DecodeTime(Time: TDateTime; var Hour, Min, Sec,
   MSec: Word);
```

A propósito, mi fecha de nacimiento es 23562,58. Pruebe el siguiente código, que utiliza la función *DayOfWeek*, para averiguar qué día de la semana ha nacido; si le tocó un sábado, le felicito sinceramente:

```
procedure TForm1.Button1Click(Sender: TObject);
const
   DiaSemana: array [1..7] of string =
   ('Domingo', 'Lunes', 'Martes', 'Miércoles',
    'Jueves', 'Viernes', 'Sábado');
var
   S: string;
begin
   S := '';
```

```
   if InputQuery('Calendario', 'Cumpleaños:', S) then
      ShowMessage(DiaSemana[DayOfWeek(StrToDate(S))]);
end;
```

He utilizado la función *StrToDate* para convertir una cadena de caracteres en un valor de tipo *TDateTime*. Hay una batería completa de funciones para realizar conversiones entre valores de fecha y hora y cadenas de caracteres. Son las siguientes:

```
function StrToDate(const Cadena: string): TDateTime;
function StrToTime(const Cadena: string): TDateTime;
function StrToDateTime(const Cadena: string): TDateTime;
function DateToStr(Fecha: TDateTime): string;
function TimeToStr(Hora: TDateTime): string;
function DateTimeToStr(FechaHora: TDateTime): string;
function FormatDateTime(const Formato: string;
   FechaHora: TDateTime): string;
```

De las funciones de conversión a cadenas de caracteres, la más potente es *FormatDateTime*, que permite especificar una cadena de formato. Sobre la sintaxis de estas cadenas de formato hablaremos en el capítulo sobre componentes de campos. Como adelanto, le ofrezco otra versión del código que muestra el día de la semana en que nació el usuario:

```
procedure TForm1.Button1Click(Sender: TObject);
var
   S: string;
begin
   S := '';
   if InputQuery('Calendario', 'Cumpleaños:', S) then
      ShowMessage(FormatDateTime('dddd', StrToDate(S)));
end;
```

He aprovechado que los cuatro caracteres *dddd* en una cadena de formato de fechas indican el nombre completo del día de la semana.

Cadenas de caracteres cortas

El Pascal original de Niklaus Wirth no definía un tipo especial para trabajar con cadenas de caracteres; por supuesto, éstas se podían representar mediante un vector de caracteres, pero no existían funciones y procedimientos predefinidos para manipularlas. Por lo tanto, las diferentes implementaciones históricas de Pascal ofrecieron sus propias variantes de cadenas de caracteres. En particular, Turbo Pascal definía un tipo **string**, que permitía almacenar cadenas de longitud variable de hasta 255 caracteres de longitud. El formato utilizado por estas cadenas era diferente del utilizado por el lenguaje C; como resultado, algunas operaciones eran más eficientes en Pascal que en C, y viceversa. Con el advenimiento de nuevas arquitecturas de procesadores y la aparición de Windows, las cadenas de Turbo Pascal se tornaron obsoletas, pues el límite de 255 caracteres se hizo demasiado pequeño, y las funciones de Windows tra-

bajan todas con el formato de cadenas de C. La consecuencia de esta evolución del lenguaje es que actualmente coexisten en Delphi dos formatos diferentes para representar cadenas de caracteres.

En Delphi 1 sólo disponemos de las cadenas cortas. El tipo **string** de Delphi 1, o el *ShortString* de Delphi 2, se representa mediante un vector de hasta 256 bytes. El primer byte del vector contiene la longitud de la cadena; de aquí la limitación a 255 caracteres como máximo. Si *S* es una variable de tipo *ShortString*, entonces *S[0]* representa la longitud de la cadena almacenada en *S*… con la salvedad de que este valor es de tipo *Char*. Si queremos obtener la longitud como un valor entero, que es lo apropiado, tendremos que utilizar la expresión *Ord(S[0])*, para convertir el tipo *Char* a un *Integer*. Por supuesto, es más sensato utilizar directamente la función *Length*, que devuelve lo mismo y hace el código más claro.

El resto de los caracteres de la cadena *S* serían *S[1]*, *S[2]*, *S[3]*, y así sucesivamente; el último carácter de la cadena es *S[Length(S)]*. Por ejemplo, en los heroicos tiempos del Turbo Pascal, para llevar una cadena a mayúsculas, teníamos que definir una función parecida a la siguiente:

```
// ¡¡¡Función obsoleta, incluso incorrecta (en Windows)!!!
// Utilice AnsiUpperCase para llevar a mayúsculas
function Mayusculas(S: string): string;
var
   I: Integer;
begin
   for I := Length(S) downto 1 do
      if S[I] in ['a'..'z'] then
         S[I] := UpCase(S[I]);
         // … o S[I] := Char(Ord(S[I]) - Ord('a') + Ord('A')),
         // … si le va el fundamentalismo.
   Mayusculas := S;
end;
```

Las siguientes funciones son consideradas "clásicas", pues existen desde las primeras versiones de Turbo Pascal, y están definidas dentro de la propia unidad *System* de Delphi:

```
function Length(const S: string): Integer;
function Pos(const SubCadena, Cadena: string): Integer;
function Concat(const S1, S2: string): string;
function Copy(const S: string; Desde, Longitud: Integer): string;
procedure Delete(var S: string; Desde, Longitud: Integer);
procedure Insert(const SubCadena: string; var Cadena: string;
   Posicion: Integer);
```

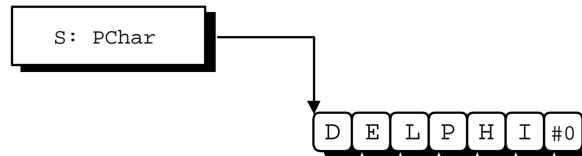
La función *Concat* se utiliza muy poco, pues es más cómodo utilizar el operador + para concatenar cadenas. En las versiones modernas de Pascal se ha añadido una amplia variedad de funciones y procedimientos que automatizan las tareas más usua-

les. Los prototipos de estas funciones pueden encontrarse en la interfaz de la unidad *SysUtils*.

> En realidad, Delphi también ofrece *WideString*, que admite alfabetos con caracteres de más de un byte. Pero como no creo que este libro llegue a ser un éxito de ventas en Japón...

Punteros a caracteres, al estilo C

En Delphi 1, y en las versiones de Pascal para Windows, las cadenas propias de Pascal tenían que coexistir con las cadenas en formato C, representadas mediante el tipo *PChar*, para poder realizar llamadas a funciones del API que necesitan cadenas. Una cadena en este formato se representa como un puntero a un vector de caracteres, pero, a diferencia de lo que sucede en Pascal, el primer carácter del vector contiene directamente el primer carácter de la cadena. Para indicar el final de ésta, se coloca un carácter nulo, cuyo código es 0, inmediatamente después del último carácter. El siguiente gráfico muestra la forma de representación de cadenas al estilo C.



Es importante darse cuenta de que una variable declarada de tipo *PChar* solamente contiene 4 bytes: el espacio necesario para el puntero. La zona de memoria a la cual apunta esta variable puede suministrarse de varias formas. Puede ser, por ejemplo, memoria reservada en la memoria dinámica (*heap*), pero también puede ser un vector declarado en memoria global o de pila:

```
var
   S: PChar;
   MyBuffer: array [0..31] of Char;
begin
   // Buffer reservado por el sistema: no hay que destruirlo
   S := 'The Delphi Obsession';
   // Buffer en memoria dinámica: hay que destruirlo al terminar
   S := StrNew('SchizoDelphia');
   StrDispose(S);
   // Buffer en memoria de pila: se libera automáticamente
   S := @MyBuffer;
   StrCopy(S, 'Altered Delphi States');
end;
```

Delphi, imitando a C, complica un poco más las cosas pues permite utilizar los vectores de caracteres con cota inicial 0 en cualquier contexto en que puede ir un valor

de tipo *PChar*. Según esto, la última asignación del procedimiento anterior puede escribirse también del siguiente modo:

```
S := MyBuffer;
```

En realidad, si tenemos un vector de caracteres con base cero reservado por Delphi, como en este último caso, no necesitamos una variable explícita de tipo *PChar*.

El programador típico de Delphi necesita trabajar con *PChar* solamente para llamar directamente a las funciones de Windows que trabajan con cadenas. Lo típico es que las cadenas que se van a utilizar estén en formato de Delphi y haya que convertirlas, y viceversa. Esta conversión es muy sencilla si estamos utilizando cadenas largas de Delphi, como veremos en la siguiente sección. Pero es más engorrosa si tenemos que utilizar Delphi 1. Las funciones necesarias se encuentran en la unidad *SysUtils*:

```
// Conversión de C a Pascal
function StrPas(S: PChar): string;
// Conversión de Pascal a C
function StrPCopy(Buffer: PChar; Origen: string): PChar;
```

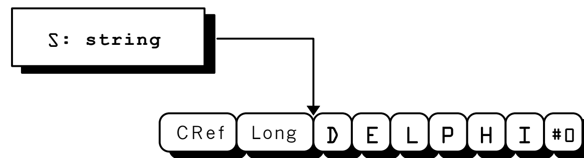
Para mostrar un mensaje estándar de Windows necesitamos utilizar el método *MessageBox*, de la clase *TApplication*. Este método, por estar basado directamente en una función del API de Windows, necesita como primer parámetro un valor de tipo *PChar*. Si vamos a utilizar este método con mucha frecuencia en Delphi 1, es recomendable declarar una función auxiliar como la siguiente:

```
procedure MostrarMensaje(const Mensaje: string);
var
   Buffer: array [0..255] of Char;
begin
   StrPCopy(Buffer, Mensaje);
   Application.MessageBox(Buffer, 'Atención',
      MB_ICONEXCLAMATION + MB_OK);
end;
```

Cadenas de caracteres largas

Con Delphi 2 se hizo necesario ampliar la capacidad de las cadenas de caracteres. Para ello se introdujo el nuevo tipo de datos *AnsiString*, que permite almacenar hasta $2^{31} - 1$ caracteres. También se ha cambiado el significado del tipo predefinido **string**. A partir de la versión 2, el tipo **string** es un sinónimo del tipo *AnsiString*. Sin embargo, es posible modificar este comportamiento mediante la opción de compilación *Huge strings*, si utilizamos el comando de menú *Project|Options*, o *{$H+}* y *{$H-}*, si queremos incorporar directamente la opción en el código fuente.

La implementación de las cadenas *AnsiString* es una mezcla de la implementación de las cadenas de C y de las antiguas cadenas cortas de Pascal. Una variable *AnsiString* ocupa exactamente 4 bytes, pues contiene un puntero. Si la cadena está vacía, el puntero es nulo; en caso contrario, apunta a una cadena de caracteres en estilo C, esto es, terminada en un carácter nulo. El espacio de memoria en el que se representa la cadena reside en la memoria dinámica de Pascal. Hasta aquí el parecido con C. Pero el bloque de memoria donde se guardan los datos contiene también la longitud de la cadena y un contador de referencias, sobre el que trataremos más adelante. ¿Dónde se guarda esta información? ¿Al final de la cadena? ¡No, se guarda antes! El siguiente gráfico ayuda a comprender la situación:



Como vemos, el puntero almacenado en la variable de cadena no apunta al principio del bloque de memoria, sino 8 bytes más adelante. El objetivo de este truco es facilitar la conversión de estas cadenas para utilizarlas en llamadas al API de Windows pues, ignorando el prefijo que contiene la longitud y el contador de referencias, el diseño en memoria de este tipo de datos lo hace indistinguible de las cadenas C.

Para evitar que el programador tenga que ocuparse directamente de pedir y liberar memoria para el *buffer* de datos, Delphi implementa un ingenioso mecanismo basado en contadores de referencia: cada cadena recuerda la cantidad de variables que apuntan a la misma. Cada vez que se asigna un cadena a una variable, se incrementa el contador de la cadena asignada. Si la variable apuntaba antes a otra cadena, la cadena que se "pierde" decrementa su contador; cuando el contador llega a cero, se libera automáticamente la memoria reservada. El decremento del contador de referencias también ocurre para las variables locales, al terminar su tiempo de vida:

```
var
   S: string;
begin
   S := 'Ars longa, vita brevis';
   ShowMessage(S);
    // Aquí, al finalizar el bloque, se libera automáticamente …
    // … la memoria reservada por S
end;
```

Esta liberación automática, en el mejor estilo C++, tiene una consecuencia importante: a partir de Delphi 2, las variables de cadena se inicializan automáticamente como cadenas vacías, no importa si se declaran como variables globales, de pila o dentro de objetos.

Por lo demás, el trabajo con estas cadenas es casi igual que antes; únicamente evite trucos sucios como los basados en el "byte cero". Además, es bueno que sepa utilizar el nuevo procedimiento *SetLength*:

```
procedure SetLength(var S: string; Longitud: Integer);
```

Con este procedimiento, Delphi asigna un nuevo *buffer* para la variable de cadena, copiando en éste cualquier información anterior. Por ejemplo, supongamos que queremos una cadena con 32 espacios en blanco. El siguiente algoritmo podía valer para Delphi 1, pero es demasiado ineficiente en las versiones actuales:

```
S := '';
for I := 1 to 32 do S := S + ' ';
```

Realmente, ahora utilizaríamos el procedimiento *AppendStr*, pero no este el problema, sino que cada carácter concatenado puede potencialmente hacer que el *buffer* de la cadena se libere y se vuelva a pedir memoria para el mismo. La forma correcta de hacerlo es la siguiente:

```
SetLength(S, 32);
FillChar(S[1], 32, ' ');
// O también:
// for I := 1 to 32 do S[I] := ' ';
```

En el ejemplo se ha utilizado el procedimiento de bajo nivel *FillChar*, que permite rellenar una zona de memoria con determinado byte.

Vectores abiertos

Los vectores abiertos no son, en realidad, un nuevo tipo de datos, sino una forma de traspaso de parámetros. O al menos hasta Delphi 4, como veremos más adelante. Este tipo de datos resuelve un par de limitaciones básicas del Pascal original: la posibilidad de trabajar con vectores de diferentes tamaños utilizando una misma rutina, y la posibilidad de utilizar listas dinámicas, construidas "al vuelo".

Por regla, la cota o índice inferior aplicable a este tipo de parámetros es *0*. Sin embargo, yo prefiero utilizar la función *Low* para conocer la cota inferior de un vector. La cota superior de un vector abierto se puede conseguir mediante la función *High*. El siguiente código muestra la implementación de una función que suma un vector con un número arbitrario de valores enteros:

```
function SumarEnteros(const Lista: array of Integer): Integer;
var
   I: Integer;
begin
   Result := 0;
```

```
      for I := High(Lista) downto Low(Lista) do
         Inc(Result, Lista[I]);
   end;
```

Volvemos a tropezar con mi vieja manía de comenzar, siempre que sea posible, un bucle **for** desde el valor variable hasta el valor constante, invirtiendo el sentido del incremento de la variable de control si es necesario (**downto** en vez de **to**); recuerde que de esta forma el código generado es más corto y rápido. Observe también el uso de *Result* para el valor retornado por la función. En Pascal estándar esta función hubiera requerido otra declaración para una variable entera temporal, y una asignación adicional al final de la rutina.

Veamos ahora posibles formas de uso de esta función:

```
const
   Vector1: array [1..3] of Integer = (1905, 1916, 1927);
var
   Vector2: array [1995..1998] of Integer;
begin
   // Un vector de tres componentes; cota inferior = 1
   ShowMessage(IntToStr(SumarEnteros(Vector1)));
   // Un vector de cuatro componentes; cota inferior = 1995
   Vector2[1995] := 1;
   Vector2[1996] := 2;
   Vector2[1997] := 3;
   Vector2[1998] := 4; // ¿Versiones de Delphi, quizás?
   ShowMessage(IntToStr(SumarEnteros(Vector1)));
   // Un vector construido dinámicamente; cota inferior = 0
   ShowMessage(IntToStr(SumarEnteros([1,2,10,20,1000])));
end;
```

Quizás la parte más sorprendente del ejemplo anterior, al menos para un viejo programador de Pascal, sea el uso de parámetros vectoriales construidos de forma dinámica. Los valores de una expresión de este tipo se encierran entre corchetes, separados por comas, con una sintaxis similar a la de los conjuntos. Aunque he utilizado en el ejemplo valores constantes dentro del vector dinámico, es posible utilizar también variables y expresiones en general:

```
   SumarEnteros([Paga1, Paga2, Paga3 div 3]);
```

Otro detalle: aunque cada llamada a la función *SumarEnteros* se realiza con vectores con cotas inferiores diferentes, dentro de la función la llamada a *Low(Lista)* siempre devuelve el valor *0*, y *High(Lista)* devuelve el número de elementos en el vector menos uno.

El lenguaje de programación más popular que conozco que tiene una construcción similar es Modula II.

Vectores dinámicos

En definitiva, mezclando las técnicas utilizadas en los parámetros de vectores abiertos y las de uso de cadenas largas, se puede implementar tipos de vectores dinámicos. El equipo de desarrollo de Delphi se percató de lo anterior, e introdujo los vectores dinámicos en la versión 4. Con el nuevo Delphi, podemos realizar algoritmos como el siguiente:

```
procedure TentarAlAzar(N: Integer);
var
   I: Integer;
   A: array of Real;
   Mean, StdDev: Real;
begin
   ASSERT(N > 0);
   SetLength(A, N);
   Mean := 0;
   for I := Low(A) to High(A) do
   begin
      A[I] := Random;
      Mean := Mean + A[I];
   end;
   Mean := Mean / N;
   StdDev := 0;
   for I := Low(A) to High(A) do
      StdDev := StdDev + Sqr(Mean - A[I]);
   StdDev := Sqrt(StdDev / (N - 1));
   ShowMessage(Format('Media: %f, Desviación estándar: %f',
      [Mean, StdDev]));
end;
```

La variable *A* ha sido declarada como un vector de reales, pero sin especificar sus cotas. En realidad, Delphi representa esta variable como un puntero, al cual inicializa como vacío, **nil**. El procedimiento *SetLength* es el responsable de pedir memoria para un *buffer* de *N* elementos y modificar el puntero. De ahí en adelante, el trabajo con el vector es predecible. Se pueden utilizar con él las funciones *Low*, *High*, *Length* y *Copy* (¡sí, como si fuera una cadena de caracteres!). Observe que al final del algoritmo no se ha liberado explícitamente la memoria asociada al vector, pues Delphi utiliza también un mecanismo de contadores de referencia y de liberación automática con este tipo de datos.

También pueden declararse vectores dinámicos multidimensionales:

```
var
   Matriz: array of array of Real;
begin
   SetLength(Matriz, 4, 16);
   // …
end;
```

Vectores abiertos variantes

Un caso particular de parámetro vectorial abierto es el *vector abierto variante*, que permite pasar dentro de un mismo vector valores pertenecientes a varios tipos. Esta construcción, introducida en Delphi 1, permite simular los procedimientos con cantidad variable de parámetros de C/C++. Los parámetros de este tipo se declaran como se muestra en el segundo parámetro de la siguiente función:

```
function Format(const Format: string;
   const Args: array of const): string;
```

Esta función se utiliza de la siguiente forma:

```
ShowMessage(Format('%d: %s', [666, 'El Número De La Bestia']));
```

Si el lector conoce algo de C, reconocerá en esta función un sustituto a la función *sprintf* de este último lenguaje.

Los vectores abiertos variantes son utilizados por varias funciones de Delphi, en especial las que trabajan con búsquedas sobre índices, rangos de valores, etc. Para trabajar con estos parámetros internamente, tenemos que hacernos la idea de que han sido declarados del siguiente modo:

```
array of TVarRec
```

El tipo *TVarRec*, por su parte, se define de esta manera:

```
type
   TVarRec = record case Byte of
      vtInteger:    (VInteger: Integer; VType: Byte);
      vtBoolean:    (VBoolean: Boolean);
      vtChar:       (VChar: Char);
      vtExtended:   (VExtended: PExtended);
      vtString:     (VString: PShortString);
      vtPointer:    (VPointer: Pointer);
      vtPChar:      (VPChar: PChar);
      vtObject:     (VObject: TObject);
      vtClass:      (VClass: TClass);
      vtWideChar:   (VWideChar: WideChar);
      vtPWideChar:  (VPWideChar: PWideChar);
      vtAnsiString: (VAnsiString: Pointer);
      vtCurrency:   (VCurrency: PCurrency);
      vtVariant:    (VVariant: PVariant);
   end;
```

Variantes

A partir de Delphi 2, se incorporan al repertorio de tipos de datos los valores variantes, que se declaran mediante el tipo *Variant*. Una variable de tipo variante sirve de contenedor para un valor que puede pertenecer a los tipos más diversos: enteros, cadenas, reales, matrices dinámicas, objetos OLE, etc. Además, el valor variante almacena en un campo interno un indicador de a qué tipo concreto pertenece el valor almacenado en un momento dado.

La presencia de un tipo de datos que pueda almacenar "cualquier cosa" no parece muy acorde a la filosofía de Delphi de comprobación fuerte de tipos (*strong type checking*), y puede parecer un paso atrás. Sin embargo, estos tipos no han sido añadidos al lenguaje por "capricho", siendo el motivo principal de su incorporación su uso por OLE. Por otra parte, el peligro que corremos de mezclar inadvertidamente tipos incompatibles es mínimo porque, como hemos dicho, cada valor viene marcado con su indicador de tipo. Esto permite realizar las asignaciones y conversiones más bizarras:

```
var
   V: Variant;
   I: Integer;
   S: string;
begin
   V := 123;
   S := V;     // A S se le asigna la cadena '123'
   V := '321';
   I := V;     // A I se le asigna el número 321
   V := 'Surprise!';
   I := V;     // Esto produce una excepción
end;
```

Las variables de tipo *Variant* disfrutan de inicialización y destrucción automática, del mismo modo que las cadenas de caracteres. El valor inicial que reciben es *UnAssigned*. Este valor no pertenece a ninguno de los tipos convencionales. El otro valor especial que puede asumir un variante es *Null*, que se utiliza en procedimientos de bases de datos. Se pueden comparar directamente variantes con las constantes *Null* y *UnAssigned*, o utilizar las siguientes funciones:

```
function VarIsEmpty(const V: Variant): Boolean;
function VarIsNull(const V: Variant): Boolean;
```

Además de poder contener tipos básicos, como cadenas, reales y enteros, y los valores atómicos especiales que acabamos de ver, un variante puede almacenar una *matriz variante*. Algunas funciones de Delphi pueden devolver variantes en este estilo; veremos estas funciones al estudiar los componentes de acceso a campos y los métodos de búsqueda. Las funciones que devuelven información acerca de una matriz variante son:

```
function VarIsArray(const V: Variant): Boolean;
function VarArrayDimCount(const V: Variant): Integer;
function VarArrayLowBound(const V: Variant; Dim: Integer): Integer;
function VarArrayHighBound(const V: Variant; Dim: Integer): Integer;
```

Para acceder a los elementos de una matriz variante, se pueden utilizar directamente los corchetes:

```
ShowMessage(V[0] + V[1]);
```

Ciertas funciones de Delphi necesitan que les suministremos valores empaquetados en matrices variantes. Hay distintas formas de crearlas, y la más sencilla es utilizar la función *VarArrayOf*:

```
function VarArrayOf(const Valores: array of const): Variant;
```

El otro tipo de valor que se puede almacenar en una variable de tipo variante es un objeto OLE. Los objetos OLE se asignan generalmente mediante llamadas a la función *CreateOleObject*:

```
var
   V: Variant;
begin
   V := CreateOleObject('Word.Basic');
   // Se activa el servidor de automatización
   // y se desactiva al terminar el bloque de instrucciones
end;
```

Clases para la representación de listas

Los tipos de datos que estudiaremos en esta sección son clases de objetos, a diferencia de los tipos que hemos visto hasta el momento. Se utilizan con mucha frecuencia, por lo cual es necesario saber aprovechar sus posibilidades. Me refiero a las clases *TStrings* y *TStringList*, que representan listas de cadenas de caracteres, y la clase *TList*, que es la clase de lista más general de Delphi, y permite almacenar punteros a objetos arbitrarios.

Comencemos con las listas de cadenas. ¿Por qué dos clases diferentes? La respuesta es que *TStrings* es una clase abstracta, con métodos virtuales abstractos, y que no se compromete con un tipo de implementación particular. En cambio, *TStringList* es una clase concreta, que hereda de *TStrings* e implementa la lista mediante un vector de punteros a cadenas con reasignación de espacio dinámica. Casi todas las propiedades y parámetros que utilizan listas de cadenas se declaran de tipo *TStrings*, para permitir más generalidad, pero casi siempre tienen asociados objetos *TStringList* en tiempo de ejecución.

La creación de una lista de cadenas es muy sencilla:

```
var
   Lista: TStrings;
begin
   Lista := TStringList.Create;      // ¡Polimorfismo!
   try
      // Utilizar la lista
   finally
      Lista.Free;
   end;
end;
```

En el ejemplo anterior se destruye la lista en el mismo bloque de instrucciones, por lo que se ha utilizado la instrucción **try...finally** para garantizar la destrucción del objeto.

Una vez que tenemos una lista de cadenas, es fácil añadirle cadenas:

```
Lista.Add('...por su eslabón más débil');    // Añade al final
Lista.Insert(0, 'Toda cadena se rompe...'); // Inserta al principio
```

Para acceder a una cadena existente. o modificarla, se utiliza la propiedad vectorial *Strings*. Esta es también la propiedad por omisión de la clase, por lo cual los corchetes con el índice se pueden colocar directamente a continuación de la variable de lista:

```
ShowMessage('Línea 1: ' + Lista.Strings[0]);
ShowMessage('Línea 1: ' + Lista[0]);        // Igual a lo anterior
Lista[1] := Lista[1] + ', decía Confucio';
```

Los índices de las cadenas dentro de estas listas comienzan a partir de 0. La cantidad total de cadenas almacenadas se encuentra en la propiedad *Count*, de sólo lectura. Cualquier intento de utilizar un índice inválido provoca una excepción.

Cuando las cadenas almacenadas tienen el formato *Nombre*=*Valor*, como sucede con las líneas de un fichero de configuración, pueden utilizarse las propiedades *Names* y *Values*:

```
Lista.Values['Software'] := 'Delphi';
ShowMessage(Lista.Names[0] + ' igual a ' +
   Lista.Values[Lista.Names[0]]);
```

La posibilidad más interesante de estas listas, y la más desaprovechada, es poder almacenar punteros a objetos arbitrarios asociados con las cadenas. A estos punteros se accede mediante la propiedad vectorial *Objects*:

```
property Objects[Indice: Integer]: Pointer;
```

Existen métodos similares a *Insert* y *Add* que permiten añadir la cadena a la vez que su objeto asociado:

```
procedure InsertObject(Index: Integer; const S: string;
   AObject: TObject);
function AddObject(const S: string; AObject: TObject): Integer;
```

Cuando se destruye una lista de cadena, el espacio reservado para las cadenas se destruye automáticamente. Esto, sin embargo, no ocurre con los objetos asociados. Si deseamos que los objetos insertados en una lista de cadena se destruyan también, hay que programarlo antes:

```
for I := 0 to Lista.Count - 1 do
   TObject(Lista.Objects[I]).Free;
Lista.Free;
```

La función *IndexOf* permite buscar una cadena, devolviendo su posición; existe una función similar, *IndexOfObject*, para localizar un objeto dado su puntero:

```
I := Lista.IndexOf('OCX');
if I <> -1 then
   Lista[I] := 'ActiveX';
// La "rosa", con otro nombre, parece que no huele igual
```

Hasta aquí hemos hablado del tipo *TStrings* en general. La clase *TStringList* hereda todas las cualidades de su ancestro, añadiendo una implementación y las siguientes propiedades:

```
// Sorted permite listas ordenadas alfabéticamente
property Sorted: Boolean;
// Duplicates sólo funciona cuando Sorted está activa
property Duplicates: (dupIgnore, dupAccept, dupError);
```

Además, esta clase incluye un par de eventos, *OnChange* y *OnChanging* para avisarnos de los cambios que se produzcan. Estos eventos pueden interesarnos, si creamos un componente con propiedades de tipo *TStringList* y quisiéramos enterarnos de cualquier operación sobre el contenido de estas propiedades durante la ejecución.

Finalmente, si deseamos almacenar solamente punteros a objetos, podemos utilizar la clase *TList*. Esta clase es muy similar a las anteriores, sólo que en vez de tener una propiedad vectorial *Strings*, tiene una propiedad vectorial *Items*, de tipo puntero, que es la propiedad por omisión. Del mismo modo que con las listas de cadenas, hay que destruir explícitamente los objetos antes de liberar la lista, pues *TList* no se considera "propietaria" de sus objetos almacenados.

Streams: ficheros con interfaz de objetos

Una de las consecuencias de la larga historia de Delphi es la existencia de más de mil maneras de acceder a ficheros. Para terminar este capítulo, veremos el sistema de clases que ofrece Delphi para el trabajo con ficheros. Este sistema está basado en el concepto abstracto de *stream* (flujo, corriente; la mayoría lo traduce como *flujo de datos*). Delphi ofrece una clase abstracta *TStream*, cuyos métodos básicos son los siguientes:

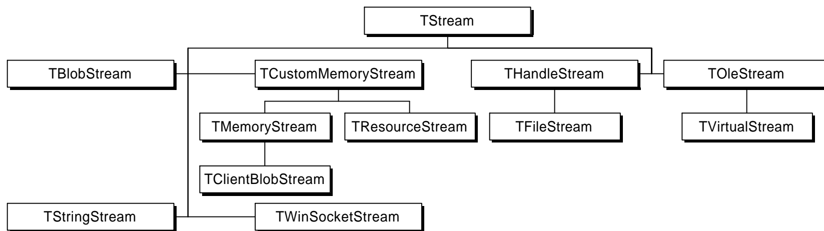
```
function Read(var Buffer; Count: Longint): Longint;
function Write(var Buffer; Count: Longint): Longint;
function Seek(Offset: Longint; Origin: Word);
```

Un *stream* contiene también estas dos propiedades:

```
property Size: Longint;
property Position: Longint;
```

La clase *TStream* define además varios métodos auxiliares basados en los métodos básicos; tal es el caso de *CopyFrom*, que permite copiar un fragmento de un *stream* en otro objeto de este tipo, de *WriteComponent*, que permite almacenar un componente en un *stream*, etc.

La jerarquía de clases derivadas de *TStream* es la siguiente, en Delphi 4:



De esta amplia variedad de clases, las que más nos interesan son:

- *TFileStream*: Para trabajar con ficheros del sistema operativo.
- *TBlobStream*: Para trabajar con el contenido de campos BLOB (imágenes, documentos), como si fueran ficheros.
- *TMemoryStream*: Para acceder a una zona de memoria como si se tratase de un fichero.

En el capítulo 17 tendremos oportunidad de trabajar con *TBlobStream*. Aquí vamos a mostrar un ejemplo sencillo, que trata sobre cómo buscar una cadena determinada dentro de un fichero. La función *Pos* de Delphi podría servir, pero es muy ineficiente para cadenas grandes. El algoritmo que utilizaremos será el de Rabin y Karp, que

permite realizar estas búsquedas de forma muy eficiente. No explicaré el funcionamiento del mismo, pero puede consultar, por ejemplo, *Algoritmos en C++*, de Robert Sedgewick. Esta es la implementación del algoritmo en Delphi:

```
function RabinKarp(const S1, S2: PChar; L1, L2: Integer): Integer;
   overload;
const
   D = 32;
   Q = 33554393;
var
   I, DM, H1, H2: Integer;
begin
   DM := 1;
   for I := 1 to L1 - 1 do
      DM := (D * DM) mod Q;
   H1 := 0;
   H2 := 0;
   for I := 0 to L1 - 1 do
   begin
      H1 := (H1 * D + Ord(S1[I])) mod Q;
      H2 := (H2 * D + Ord(S2[I])) mod Q;
   end;
   I := 0;
   while H1 <> H2 do
   begin
      H2 := (H2 + D * Q - Ord(S2[I]) * DM) mod Q;
      H2 := (H2 * D + Ord(S2[I + L1])) mod Q;
      if I > L2 - L1 then
      begin
         Result := -1;
         Exit;
      end;
      Inc(I);
   end;
   Result := I;
end;
```

El algoritmo recibe dos punteros a dos *buffers* de caracteres, y la longitud de ambas zonas de memoria. Se debe buscar la primera ocurrencia de los caracteres del primer *buffer* dentro del segundo. Si no se encuentra, la función devuelve *-1*; en caso contrario, devuelve la posición donde se ha encontrado, contando como *0* la primera posición. El algoritmo, en la versión que presento, considera como diferentes las mayúsculas y minúsculas.

¿Se ha fijado en la directiva **overload** que he puesto en la declaración de la función? Es que muchas veces nos interesa buscar directamente una cadena dentro de otra, al estilo de la función *Pos*. Si estuviéramos en Delphi 3, tendríamos que crear otra función llamada *RabinKarpStr* o algo por el estilo. En Delphi 4, utilizando la directiva mencionada, podemos volver a aprovechar el nombre en la definición de otra función:

```
function RabinKarp(const S1, S2: string): Integer; overload;
begin
   Result := RabinKarp(PChar(S1), PChar(S2),
      Length(S1), Length(S2)) + 1;
end;
```

Ahora ya estamos en condiciones de definir una función que busque una cadena dentro de un fichero:

```
function FindString(const AFileName, AStr: string): Boolean;
begin
   with TMemoryStream.Create do
   try
      LoadFromFile(AFileName);
      Result := RabinKarp(PChar(AStr), Memory,
         Length(AStr), Size) <> -1;
   finally
      Free;
   end;
end;
```

La clase *TMemoryStream* permite acceder a un *buffer* en memoria como si de un fichero se tratara. Este *buffer* lo hemos llenado en este ejemplo utilizando el método *LoadFromFile*, que lee el contenido del fichero en la memoria de la clase. La propiedad *Memory* permite acceder al *buffer*, y se utiliza como parámetro de la función *RabinKarp* que hemos definido antes.

Capítulo

13

Técnicas de gestión de ventanas

NO TODO HUECO EN UNA PARED es una ventana. En este capítulo, veremos trucos y técnicas para luchar contra el gran problema del programador en Windows 3.x y Windows 95: la posibilidad de agotar los recursos del sistema debido a una sobreabundancia de ventanas y controles. Un buen programador intenta mantener el número mínimo de ventanas creadas, para lo cual crea las ventanas en el momento en que las va a utilizar. Y hay también que liberar los recursos que consumen en el momento adecuado. Pero también mostraremos algunas posibilidades relacionadas con la gestión del teclado y ratón, útiles en la programación para bases de datos.

¿Qué hace Delphi cuando lo dejamos solo?

Los protagonistas de esta sección serán las propiedades *FormStyle*, *Visible* y el evento *OnClose*; son ellos los que determinan el comportamiento de la creación y destrucción de ventanas. Comencemos con una aplicación SDI en la cual las ventanas tienen en su propiedad *FormStyle* uno de los valores *fsNormal* ó *fsStayOnTop*. Al crear una de estas ventanas durante el diseño, el objeto correspondiente se inicializa con la propiedad *Visible* a *False*, incluso en el caso de la primera ventana. Como el lector puede comprobar fácilmente, por cada ventana creada se añade una instrucción en el fichero de proyecto, el de extensión *dpr*, para crear la ventana al iniciarse la ejecución de la aplicación, antes de entrar en el bucle de mensajes. Las ventanas se crean mediante el método *CreateForm* de la clase *TApplication*:

```
begin
   Application.Initialize;
   Application.CreateForm(TForm1, Form1);
   Application.CreateForm(TForm2, Form2);
   Application.Run;
end.
```

¿Por qué se utiliza *CreateForm*, en vez de utilizar directamente el constructor? El motivo principal es que *CreateForm* se encarga de designar a la primera ventana creada de este modo como la ventana principal de la aplicación, aquella ventana que cuando se cierra provoca el fin de la ejecución del programa. Así que en la secuencia de instruc-

ciones anterior, el formulario *Form1* se convierte en la ventana principal de la aplicación, mientras que *Form2*, aunque creada, queda invisible. ¿Algún misterioso comportamiento de *CreateForm*? No, sencillamente que *Form2* tiene el valor *False* en su propiedad *Visible*. Por lo general, para mostrar el segundo formulario hay que aplicarle el método *Show* o *ShowModal*. Pruebe a crear una aplicación sencilla con dos formularios, y cambie la propiedad *Visible* del segundo a *True*; en este caso, las dos ventanas serán visibles desde el principio. ¿Y cómo es que la ventana principal, cuya propiedad *Visible* contiene comúnmente *False*, se muestra automáticamente? Bueno, esta vez es *Run* el que se encarga de aplicar *Show* a esta ventana antes de zambullirse en el ciclo de mensajes.

Seguimos con las aplicaciones SDI. Tenemos una ventana abierta, diferente de la principal; puede que se haya mostrado automáticamente, cambiando desde el principio *Visible* a *True*, o por medio de *Show* ó *ShowModal*. Y ahora cerramos esta ventana. ¿Adónde van las almas de las ventanas al morir? Depende del tipo de ventana, según el valor de la propiedad *FormStyle*. Para las ventanas *fsNormal* y *fsStayOnTop*, el comportamiento por omisión es ocultarse. ¿Que cómo lo sé? Cree un manejador para el evento *OnClose* del formulario con el cual quiere experimentar, digamos, el segundo formulario:

```
procedure TForm2.FormClose(Sender: TObject;
   var Action: TCloseAction);
begin
   ShowMessage(GetEnumName(TypeInfo(TCloseAction), Ord(Action)));
end;
```

Aquí he hecho un truco sucio: he transformado un valor enumerativo en una cadena de caracteres; para poder utilizar las funciones *GetEnumName* y *TypeInfo* hay que utilizar la unidad *TypInfo*. En definitiva, cuando cerramos la ventana en cuestión, debe mostrarse un cuadro de mensajes con el texto *caHide*. Es decir, las ventanas "normales" se ocultan al cerrarse, pero no se destruyen. Por esta causa, para volver a mostrar la ventana correspondiente basta con aplicar nuevamente *Show* ó *ShowModal* al objeto sin necesidad de volverlo a crear, pues nunca fue destruido. Hay una pequeña excepción: la ventana principal. Independientemente de las intenciones de ésta, la operación de cierre provoca el fin de la aplicación. Qué se le va a hacer.

El evento *OnClose* puede utilizarse para cambiar el comportamiento por omisión durante el cierre de un formulario. Por este motivo, el parámetro *Action* se pasa por referencia: nos están pidiendo nuestra opinión al respecto. En este capítulo, más adelante, veremos cómo indicar que la ventana se destruya cuando se cierre, para evitar el despilfarro de recursos.

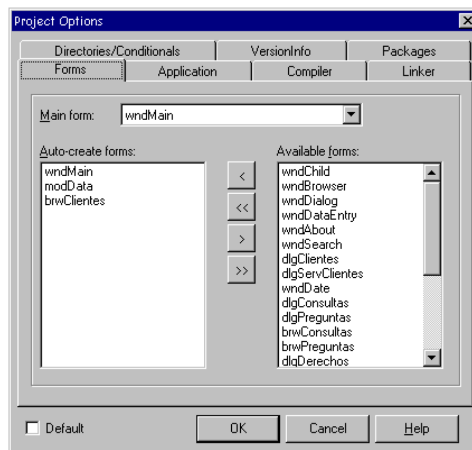
Si la aplicación es MDI las cosas cambian ligeramente. La ventana principal debe tener en *FormStyle* el valor *fsMDIForm*, mientras que cada ventana hija debe tener el estilo *fsMDIChild*. Esto no quita que existan ventanas dentro de la aplicación cuyo

estilo sea *fsNormal* o *fsStayOnTop*, principalmente cuadros de diálogo con ejecución modal. Cuando una ventana tiene el estilo *fsMDIForm* o *fsMDIChild* no puede cambiar su propiedad *Visible* a *False*. Además, el valor inicial del parámetro *Action* en el evento *OnClose* es *caMinimize*, con lo que al intentar cerrar una ventana hija MDI lo que lograremos es minimizarla, en realidad.

Creación inmediata de ventanas modales

Un buen programador de Delphi, trabajando en una aplicación de mediano a gran tamaño, debe tener sumo cuidado con el uso de recursos del sistema, y en particular con las ventanas. Si aceptamos el comportamiento habitual de Delphi, todas las ventanas posibles de la aplicación se crean durante la carga de la misma. Esto hace más lenta la operación, y deja además bastante mal parados los recursos de Windows. Más aún, al cerrar ventanas que no son MDI, como vimos en la sección anterior, no estamos realmente liberando los recursos consumidos. Debemos olvidar en lo posible la creación automática de ventanas de Delphi, y crearlas en el preciso momento en que las necesitamos, sin olvidar destruirlas en el instante en que nos dejen de hacer falta.

Es fácil desactivar la creación automática de ventanas. Aunque se puede editar manualmente el código de inicio de programa en el fichero de proyecto *dpr*, es preferible ejecutar el comando de menú *Project|Options* (*Options|Project* en Delphi 1), y en la página *Forms* del diálogo que se activa mover los formularios necesarios al cuadro de lista de la derecha.



En lo concerniente a la creación manual de ventanas, tenemos que distinguir entre crear ventanas para ejecución modal y no modal. Casi siempre se puede destruir una ventana modal en el mismo sitio en que se crea y ejecuta, lo cual nos facilitará las

cosas. En la siguiente sección adaptaremos esta técnica a la creación y ejecución de ventanas no modales.

El código inicial para la creación y ejecución modal de ventanas debe parecerse al siguiente:

```
procedure TForm1.MostrarForm2(Sender: TObject);
var
   F: TForm2;
begin
   F := TForm2.Create(Application);
   F.ShowModal;
   F.Free;
end;
```

Esto es programación de presupuestos bajos: se hace lo indispensable solamente. El principal problema consiste en que, de producirse una excepción durante la ejecución de la ventana, no se libera la memoria del objeto. Por lo tanto, introducimos una instrucción **try**…**finally** y aprovechamos para colar un **with**…**do**, un tanto maliciosamente:

```
procedure TForm1.MostrarForm2(Sender: TObject);
var
   F: TForm2;
begin
   F := TForm2.Create(Application);
   with F do
   try
      ShowModal;        // Es decir, F.ShowModal
   finally
      Free;             // O también, F.Free
   end;
end;
```

Si nos fijamos un poco, la variable temporal *F* solamente se utiliza, después de ser asignada, en la instrucción **with**. Ahora debe quedar claro el porqué de esta instrucción, ya que haremos desaparecer la variable temporal:

```
procedure TForm1.MostrarForm2(Sender: TObject);
begin
   with TForm2.Create(Application) do
   try
      ShowModal;
   finally
      Free;
   end;
end;
```

El toque final de elegancia consiste en crear un procedimiento que sirva para mostrar de forma modal *cualquier* tipo de ventana de la aplicación. Naturalmente, necesitamos un parámetro para el tipo de ventana, es decir, una referencia de clase. Aunque pu-

diéramos utilizar el tipo *TComponentClass*, para el cual ya está definido el constructor virtual *Create*, es más sano restringir el parámetro al tipo *TFormClass*:

```
procedure TForm1.MostrarModal(AClass: TFormClass);
begin
   with AClass.Create(Application) do
   try
      ShowModal;
   finally
      Free;
   end;
end;
```

Creación inmediata de ventanas no modales

Si solamente permitimos una instancia de cierto tipo de ventana, la situación se complica ligeramente. Por ejemplo, mi aplicación tiene un comando *Ver|Clientes*, que debe visualizar la tabla de clientes. Si utilizamos la instrucción:

```
TbrwClientes.Create(Application);
```

cada vez que ejecutemos el comando se creará una ventana independiente. Para evitar este comportamiento, debemos localizar primero el formulario, que puede estar creado y oculto o en un segundo plano. Si estamos en Delphi 2 o posterior, el código necesario es el siguiente:

```
procedure TForm1.MostrarForm2(Sender: TObject);
var
   F: TForm;
begin
   F := Application.FindComponent('Form2') as TForm;
   if Assigned(F) then
      F.Show
   else
      TForm2.Create(Application);
end;
```

El método *FindComponent* devuelve, cuando se aplica al objeto *Application*, el puntero al objeto cuyo nombre se especifica como parámetro. El tipo del valor de retorno de este método es *TComponent*, por lo cual hay que realizar la conversión de tipo; para asegurarnos contra sorpresas desagradables, utilizamos el operador **as**. Si se encuentra la ventana de nombre *Form2*, a la variable *F* se le asigna el puntero a la misma; de lo contrario, se le asigna el puntero nulo, **nil**. En el primer caso, la ventana encontrada pasa a primer plano gracias a la aplicación del método *Show*. En el segundo, se crea un objeto de este tipo.

También es posible parametrizar este método para obtener un método genérico de crear y mostrar ventanas no modales. He aquí nuestra primera versión:

```
procedure TForm1.MostrarNoModal(AClass: TFormClass;
   const AName: string);
var
   F: TForm;
begin
   F := Application.FindComponent(AName) as TForm;
   if Assigned(F) then
      F.Show
   else
      AClass.Create(Application);
end;
```

Esta ha sido una sustitución mecánica de valores por parámetros. Se puede eliminar la necesidad del segundo parámetro del procedimiento, *AName*, pues el nombre de un objeto de formulario puede deducirse directamente del nombre de la clase. El procedimiento queda de este modo:

```
procedure TForm1.MostrarNoModal(AClass: TFormClass);
var
   F: TForm;
begin
   F := Application.FindComponent(Copy(AClass.ClassName, 2, 255))
      as TForm;
   if Assigned(F) then
      F.Show
   else
      AClass.Create(Application);
end;
```

ClassName es un método que, aplicado a una referencia de clase, nos ofrece el nombre de la clase. De este nombre nos sobra la *T* inicial, la cual es eliminada mediante la función *Copy*. El resto del procedimiento es el mismo.

Sin embargo, existe un pequeño problema en relación con la implementación del método anterior: no funciona en Delphi 1. Por algún extraño motivo, solamente a partir de la versión 2 se permite la búsqueda de ventanas mediante el método *FindComponent* de la clase *TApplication*. Pero siempre podemos echar mano de la clase *TScreen*, más bien, de la variable global *Screen*, que pertenece a este tipo. Esta clase tiene un par de propiedades, *FormCount* y *Forms*, la segunda de ellas vectorial, que llevan la cuenta de las ventanas existentes durante la ejecución de la aplicación. La solución es realizar una búsqueda mediante estas propiedades, en vez de confiar en *FindComponent*. La versión final de *MostrarNoModal* es ésta:

```
procedure TForm1.MostrarNoModal(AClass: TFormClass);
var
   I: Integer;
begin
   for I := Screen.FormCount - 1 downto 0 do
      if Screen.Forms[I] is AClass then
      begin
         Screen.Forms[I].Show;
         Exit;
      end;
```

```
      AClass.Create(Application);
   end;
```

Un detalle: estamos resolviendo la construcción y visualización de la ventana con una única llamada al constructor:

```
      AClass.Create(Application);
```

Esto es posible si el formulario tiene *True* en su propiedad *Visible*, lo cual se cumple automáticamente para las ventanas hijas MDI. Si no es este el caso, necesitaremos aplicar el método *Show* al objeto recién creado:

```
      AClass.Create(Application).Show;
```

Y como todo lo que comienza debe acabar, hay que preocuparse por la destrucción y liberación del objeto de ventana, que deben ocurrir al cerrarse el formulario. Esto, sin embargo, es muy fácil de lograr, pues basta ejecutar la siguiente asignación en respuesta al evento *OnClose* del formulario:

```
procedure TForm1.FormClose(Sender: TObject);
begin
   Action := caFree;
end;
```

Métodos de clase para la creación dinámica

En la programación orientada a objetos se estimula el encapsulamiento de los algoritmos dentro de los objetos sobre los cuales trabajan. Las técnicas mostradas anteriormente tienen el defecto de necesitar una función externa para su llamada; *MostrarModal* y *MostrarNoModal* son funciones que hay que definir en una tercera ventana. Esto complica el asunto, pues para mostrar una ventana hay que tener acceso a esta tercera ventana "lanzadera". Además, cada vez que se inicia un nuevo proyecto hay que teclear una y otra vez el código de estas funciones, o, si estamos utilizando plantillas, necesitamos una plantilla adicional para este tercer tipo de ventana.

¿Por qué no definimos los métodos *MostrarModal* y *MostrarNoModal* precisamente dentro de las ventanas que queremos crear dinámicamente? Pues por dos razones elementales:

- Un método normalmente se aplica a una instancia de la clase, esto es, a un objeto existente. Si quiero un método para crear un objeto, me estoy metiendo en un círculo vicioso.
- Si mi proyecto tiene tres ventanas modales, estas tres clases de ventana tendrán que duplicar el código de creación automática. Esto no sucedía antes.

Claro está, ya tenemos una solución a estos dos problemas. El segundo obstáculo es realmente un problema si todavía estamos trabajando en Delphi 1; a partir de Delphi 2 lo solucionaremos utilizando *herencia visual*: las tres clases de ventanas modales deben heredar de la plantilla en la que definimos el código de lanzamiento de ventanas. Los módulos de datos y la herencia visual serán estudiados en un capítulo posterior.

En cuanto al primer inconveniente, no es tal, si recordamos la existencia de los métodos de clase, que se aplican a una referencia de clase, sin necesidad de que exista un objeto perteneciente al tipo dado. A continuación muestro la forma en que se pueden reescribir los métodos anteriores utilizando este recurso. Incluso ampliaremos un poco las posibilidades de los métodos. Comenzamos con *MostrarModal*, que es el más fácil:

```
class function TVentanaModal.Mostrar(ATag: Integer): TModalResult;
begin
   with Create(Application) do
   try
      Tag := ATag;
      Result := ShowModal;
   finally
      Free;
   end;
end;
```

Lo primero que llama la atención es la inclusión del parámetro *ATag*, de tipo entero, y el valor de retorno, de tipo *TModalResult*. Este último es comprensible, pues de este modo nos enteramos con qué acción el usuario cerró el cuadro de diálogo. En cuanto a *ATag*, lo estamos asignando a la propiedad *Tag* del nuevo formulario. El propósito de esto es indicar de algún modo al cuadro de diálogo las condiciones en que ha sido llamado. Por ejemplo, es común utilizar un mismo cuadro de diálogo para introducir los datos de un nuevo cliente y para modificar los datos de un cliente existente. Se puede utilizar como convenio que un valor de *0* en *Tag* signifique altas, mientras que *1* signifique modificaciones. De este modo, podemos inicializar el título del formulario durante la visualización del mismo:

```
type
   TdlgClientes = class(TVentanaModal)
      // …
   end;

procedure TdlgClientes.FormShow(Sender: TObject);
begin
   if Tag = 0 then
      Caption := 'Introduzca los datos del nuevo cliente'
   else
      Caption := 'Modifique los datos del cliente';
end;
```

La forma de activar esta ventana es ahora la siguiente:

```
    if TdlgClientes.Mostrar(0) then   // Para inserción
       // … etc …
    TdlgClientes.Mostrar(1);          // Para modificación
```

El otro detalle de interés es quizás la forma en que se llama al método *Create*, sin ningún tipo de prefijo. Bueno, recuerde que estamos dentro de un método de clase, y que el parámetro *Self* de estos es una referencia de clase, no un objeto o instancia. El código es más comprensible si reescribimos esa línea en la siguiente forma equivalente:

```
    with Self.Create(Application) do
```

Self será, en dependencia de la forma en que se llame a esta función, la referencia *TVentanaModal*, *TdlgCliente*, o cualquier otro tipo de clase derivado de *TVentanaModal*.

El algoritmo para crear ventanas no modales se cambia con la misma facilidad. Observe cómo hemos mejorado el procedimiento para tener en cuenta las condiciones de llamada y la posibilidad de que la ventana exista y esté en segundo plano o minimizada:

```
class procedure TVentanaNoModal.Mostrar(ATag: Integer);
var
   I: Integer;
   F: TForm;
begin
   LockWindowUpdate(Application.MainForm.Handle);
   try
      for I := Screen.FormCount - 1 downto 0 do
      begin
         F := Screen.Forms[I];
         if F is Self then
         begin
            if F.WindowState = wsMinimized then
               F.WindowState := wsNormal;
            F.BringToFront;
            Exit;
         end;
      end;
      F := Create(Application);
      F.Tag := ATag;
      F.Show;
   finally
      LockWindowUpdate(0);
   end;
end;
```

Adicionalmente, he encerrado el algoritmo básico entre llamadas a la función *LockWindowUpdate*, que pertenece al API de Windows. He querido evitar el incómodo parpadeo que causa la animación de ventanas en Windows, sobre todo cuando la ventana creada es una ventana hija MDI. *LockWindowUpdate* recibe como parámetro el *handle* o identificador de una ventana. Si es un *handle* válido, anula la actualización de su imagen en el monitor; si el *handle* es cero, actualiza la ventana en pantalla.

De esto se deduce que solamente puede haber una ventana "bloqueada" a la vez dentro de una aplicación. Al comenzar el procedimiento *Mostrar* desactivamos el dibujo en pantalla de la ventana principal de la aplicación, y lo restauramos al final. Si se trata de una aplicación MDI, esta acción evita el parpadeo porque las ventanas hijas se dibujan en el interior de la ventana principal. Y si se trata de una aplicación SDI este código no surte efecto alguno, pero tampoco hace daño.

Cómo transformar Intros en Tabs

Hay que lograr que la persona que introduce datos no tenga que apartar demasiado la mano del teclado numérico. Este es un requisito relativo, porque si la ficha de entrada contiene una mayoría de datos alfabéticos da lo mismo que se pase de un campo a otro con INTRO o con TAB.

Existen varias técnicas alternativas para esto, de las cuales describiré aquí la más sencilla. Se trata de interceptar la pulsación de la tecla INTRO *antes* de que llegue al control que tiene el foco del teclado y cambiar entonces el foco al control siguiente. El objeto que debe realizar esta operación es, por supuesto, el formulario. Y aquí tropezamos con el primer problema: normalmente el formulario recibe los eventos del teclado *después* de que los reciba el control, si es que éste no lo trata. En consecuencia, necesitamos un mecanismo para invertir este flujo de eventos. Para esto tenemos la propiedad lógica *KeyPreview* del formulario: cambie esta propiedad a *True*.

Próximo paso: crear un manejador de eventos para *OnKeyDown* o para *OnKeyPress*, da lo mismo; el ejemplo lo desarrollaré basado en el último. En este evento, interceptaremos las pulsaciones del carácter cuyo código ASCII es *#13* (INTRO), y enfocaremos el siguiente control utilizando el método … ejem … la propiedad … hum … No, no hay nada documentado al respecto. En realidad, Delphi tiene varios métodos para cambiar el control que tiene el foco, por ejemplo:

```
procedure SelectNext(CurControl: TWinControl;
   GoForward, CheckTabStop: Boolean);
procedure SelectFirst;
```

Pero estos métodos están declarados en la sección **protected** de la clase *TWinControl* y, aunque pueden utilizarse en el ejemplo anterior, no están documentados en la sección pública de la clase *TForm*. Con la ayuda de *SelectNext* nuestro manejador de eventos queda del siguiente modo:

```
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
begin
   if Key = #13 then
   begin
```

```
      SelectNext(ActiveControl, True, True);
      Key := #0;
   end;
end;
```

¿Y qué sucede si, como es de esperar, el diálogo tiene botones? En previsión de este caso, aumentamos el método para que pregunte primero por el tipo del control activo:

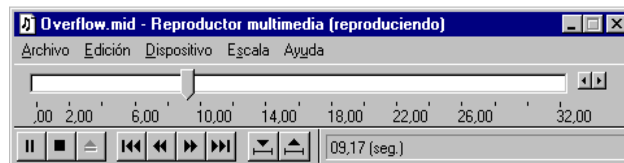
```
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
begin
   if (Key = #13) and not (ActiveControl is TButton) then
   begin
      SelectNext(ActiveControl, True, True);
      Key := #0;
   end;
end;
```

Aunque preguntamos directamente por el tipo *TButton*, se incluyen automáticamente gracias al polimorfismo todos los controles pertenecientes a clases derivadas, como *TBitBtn*.

Se puede complicar más el ejemplo anterior para tener en cuenta la presencia de rejillas. Por ejemplo, se puede intentar que INTRO pulsada sobre una rejilla seleccione la siguiente columna, o pase a la siguiente fila, o algo parecido. También hay que tener en cuenta a los objetos *TMemo* y *TDBMemo*. Le recomiendo al lector que pruebe a mejorar el algoritmo mostrado.

Cómo limitar el tamaño de una ventana

¿Ha visto el reproductor multimedia de Windows 95? La ventana principal de esta aplicación no puede crecer verticalmente, ni tiene sentido que lo haga. En cambio, es posible redimensionarla horizontalmente. Puede que nos interese simular este comportamiento para ciertas ventanas de nuestra aplicación. Por ejemplo, una ventana de entrada de datos de un pedido debe tener un ancho fijo, determinado por el formato de los campos de edición de la cabecera del pedido. Pero debe ser posible aumentar su altura, para dar cabida a más o menos líneas de detalles.



En Delphi 4 la solución es elemental. Diseñe su formulario y ejecute estas instrucciones en el evento *OnCreate*:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
   Constraints.MinHeight := Height;
   Constraints.MaxHeight := Height;
end;
```

He evitado asignar valores a la propiedad *Constraints* en tiempo de diseño. De este modo, tenemos completa libertad para cambiar el tamaño de la ventana sin preocuparnos de retocar esta otra propiedad.

Si estamos utilizando alguna versión anterior de Delphi, necesitamos tratar directamente con un mensaje de Windows: *WM_GETMINMAXINFO*. Este mensaje es enviado por Windows a una ventana cuando se intenta cambiar su tamaño. La ventana puede entonces decidir los límites que está dispuesta a aceptar. La estructura de parámetros de este mensaje se describe mediante el siguiente tipo de registro de Delphi:

```
type
   TWMGetMinMaxInfo = record
      Msg: Cardinal;                // El número del mensaje
      Unused: Integer;              // No se utiliza (por ahora)
      MinMaxInfo: PMinMaxInfo;      // Un puntero a otro registro
      Result: LongInt;              // El resultado del mensaje
   end;
```

El resultado del mensaje, el campo *Result*, debe cambiarse a *0* si la ventana se encarga de manejar el mensaje. Por su parte, el tipo de puntero *PMinMaxInfo* se define como sigue:

```
type
   PMinMaxInfo = ^TMinMaxInfo;
   TMinMaxInfo = record
      ptReserved: TPoint;
      ptMaxSize: TPoint;
      ptMaxPosition: TPoint;
      ptMinTrackSize: TPoint;
      ptMaxTrackSize: TPoint;
   end;
```

El segundo y tercer campo del registro se utilizan para dar el tamaño y las coordenadas de la ventana al ser maximizada. En cuanto a los dos últimos campos, indican el tamaño mínimo y el tamaño máximo de la misma, logrado mediante el arrastre del borde.

Pongamos por caso que queremos limitar el ancho de una ventana; la ventana podrá maximizarse y modificar su altura, pero el ancho siempre será el mismo con el que fue diseñada. Declaramos el receptor del mensaje en la sección **private** de la declaración del formulario:

```
type
   TForm1 = class(TForm)
      // …
   private
      procedure WMGetMinMaxInfo(var Msg: TWMGetMinMaxInfo);
         message WM_GETMINMAXINFO;
   end;
```

Más adelante, en la sección de implementación, colocamos el cuerpo de este método:

```
procedure TForm1.WMGetMinMaxInfo(var Msg: TWMGetMinMaxInfo);
begin
   with Msg.MinMaxInfo^ do
   begin
      ptMaxSize.X := Width;
      ptMaxTrackSize.X := Width;
      ptMinTrackSize.X := Width;
   end;
   Msg.Result := 0;
end;
```

Es fácil, como puede comprobar el lector, modificar este método para ajustarse a otras condiciones.

Cómo distribuir el área interior entre dos rejillas

Con la ayuda de la propiedad *Align*, no cuesta nada diseñar un formulario que reajuste automáticamente el tamaño de las áreas de su interior cuando cambia de tamaño. Y cuando digo que no cuesta nada me refiero a líneas de código. Por ejemplo, un formulario que muestre una rejilla y una barra de herramientas con una barra de navegación solamente requiere que el panel que sirve de barra de herramientas tenga *Align* con el valor *alTop*, mientras que la rejilla debe tener *alClient*. Al redimensionar la ventana, el panel sigue estando en la parte superior, ocupando todo el ancho de la ventana, mientras que la rejilla ocupa el resto del área disponible.

Pero muchas otras veces las relaciones entre los componentes que cubren el área interior de la ventana son más complejas. Este es el caso de una ventana en la que debemos mostrar una rejilla adicional, quizás con una tabla en relación *master/detail* con la primera tabla. O quizás con un control *TDBMemo*, que muestre un campo de este tipo correspondiente a la fila activa de la tabla que se visualiza en la rejilla. La solución más sencilla es la siguiente:

Objeto	**Valor de** Align
Panel de herramientas	*alTop*
Rejilla maestra	*alClient*
Rejilla dependiente	*alBottom*

Con esta disposición, el crecimiento vertical de la ventana se efectúa a expensas del tamaño de la rejilla maestra, manteniendo siempre su tamaño la rejilla dependiente. También se puede utilizar la siguiente disposición, que mantiene la rejilla maestra con altura constante:

Objeto	**Valor de** Align
Panel de herramientas	*alTop*
Rejilla maestra	*alTop*
Rejilla dependiente	*alClient*

No obstante, si queremos que las alturas de las rejillas mantengan cierta relación entre sí, como que siempre tengan la misma altura, no podemos basarnos únicamente en las propiedades *Align* de los controles. Hay que aprovechar entonces el evento *OnResize* del formulario para mantener mediante código la relación de tamaño. Como ejemplo tomaremos el que hemos mencionado, dos rejillas con el mismo tamaño, con la primera configuración de propiedades *Align*. Supongamos también que la rejilla maestra se nombra *gdMaster* y la dependiente *gdSlave*; sea también *pnTools* el nombre del panel con la barra de navegación. Las instrucciones necesarias en el evento *OnResize* son éstas:

```
procedure TForm1.FormResize(Sender: TObject);
begin
   gdSlave.Height := (ClientHeight - pnTools.Height) div 2;
end;
```

Solamente se cambia el tamaño de la rejilla dependiente, que tiene alineación *alBottom*; la rejilla maestra ocupará inmediatamente el área sobrante. Observe también que se utiliza la propiedad *ClientHeight* del formulario, y no *Height* a secas. El motivo es que *Height* contiene la altura total del formulario, incluyendo el borde, barras de desplazamiento y la barra de título, mientras que *ClientHeight* se limita al área útil del interior de la ventana.

A partir de Delphi 3, podemos utilizar también el nuevo componente *TSplitter*, de la página *Additional* de la Paleta de Componentes, para cambiar dinámicamente la distribución del área interior de un formulario. Este control funciona de modo similar a la línea de separación vertical entre paneles del Explorador de Windows, aunque también es posible utilizarlo con orientación horizontal. Si tiene que trabajar todavía con Delphi 2, puede extraer la implementación de este componente del código fuente e instalarlo en su Delphi.

Ventanas de presentación

Otro tipo de ventana que se utiliza con frecuencia son las ventanas de presentación (*splash windows*), esas ventanas que aparecen al principio de ciertos programas donde se muestra el logotipo de nuestra compañía, el nombre del programa y se le da gracias a nuestra abuelita por ser siempre tan comprensiva.

Para una ventana de presentación se puede utilizar cualquier tipo de ventana, pero es preferible realizar los siguientes cambios en las propiedades del formulario :

Propiedad	**Valor**	**Observaciones**
BorderStyle	*bsNone*	Eliminar borde y barra de título
Position	*poScreenCenter*	Centrar la ventana en la pantalla
Visible	*True*	Visibilidad automática

Ahora tiene completa libertad para cometer cualquier atrocidad en el diseño del interior de la ventana. Pero no olvide *quitar la ventana de la lista de ventanas con creación automática*.

La otra cara de la técnica consiste en cómo presentar esta ventana. Habitualmente, se aprovecha la creación automática de las restantes ventanas de la aplicación para este fin. Como esta creación se produce en el fichero de proyecto, de extensión *dpr*, tenemos que abrir este fichero para editarlo, preferentemente con el comando de menú *View|Project source*. La secuencia normal de instrucciones del código inicial de un programa es parecida a esto:

```
begin
   Application.Initialize;
   Application.CreateForm(TForm1, Form1);
   Application.CreateForm(TForm2, Form2);
   // ... otras ventanas ...
   Application.CreateForm(TFormN, FormN)
   Application.Run;
end.
```

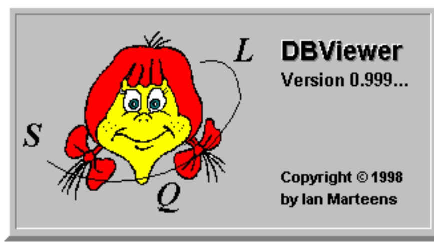
Supongamos que hemos nombrado a nuestra ventana como *Splash* ("salpicadura", el nombre usual en inglés). La ventana de presentación se debe crear *antes* de la creación de la ventana principal. Por supuesto, *no* se puede utilizar *CreateForm* para esto, porque entonces la primera ventana creada con este método se registra como la ventana principal de la aplicación. Por lo tanto, hay que crear directamente la ventana con su constructor:

```
   Splash := TSplash.Create(nil);
```

Como hemos establecido a *True* la propiedad *Visible* de la ventana de presentación, no hace falta aplicarle el método *Show*. Pero tenemos un problema: la actualización

visual de la ventana se debe producir de forma asíncrona, mediante un mensaje *WM_PAINT* colocado en la cola de mensajes de la aplicación. Y al ser éste un mensaje de baja prioridad, podemos estar seguros de que la actualización no tiene lugar mientras Windows tiene algo más importante que hacer, como crear el resto de nuestras ventanas. Por lo tanto, tenemos que forzar la actualización de la ventana de presentación mediante la siguiente instrucción:

```
Splash.Update;
```



Después de que se hayan creado todas las ventanas, hay que ocultar y liberar la ventana de presentación; esto es fácil si se llama al método *Free*. El código completo para la presentación de una ventana durante la carga queda del siguiente modo:

```
begin
   Application.Initialize;
   Splash := TSplash.Create(nil);
   Splash.Update;
   Application.CreateForm(TForm1, Form1);
   Application.CreateForm(TForm2, Form2);
   // ... otras ventanas ...
   Application.CreateForm(TFormN, FormN)
   Splash.Free;
   Application.Run;
end.
```

Capítulo

14

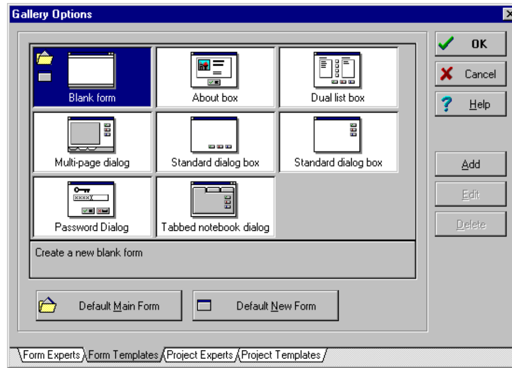
Los módulos de datos y el Depósito de Objetos

DELPHI PONE A NUESTRA DISPOSICIÓN unas cuantas técnicas para que podamos reutilizar código y datos de un proyecto a otro, e incluso dentro de un mismo proyecto. La técnica más potente y general es la creación de nuevos tipos de componentes, ya sea partiendo de cero o derivando por herencia de un componente existente. Sin embargo, la escritura de componentes es algo complicada ya que, además de exigir mayores conocimientos de las bibliotecas de Delphi y de la interfaz de Windows, consume bastante tiempo de programación. No obstante, existen otras vías para ayudar al desarrollador a evitar tareas repetitivas, y una de ellas es el empleo de plantillas y expertos. Esta técnica ha evolucionado con las sucesivas versiones de Delphi, añadiendo recursos revolucionarios dentro de los lenguajes RAD, como la herencia visual de formularios.

En el presente capítulo presentaré los *módulos de datos*, que permiten al programador separar la parte visual de sus aplicaciones de las reglas de consistencia no visuales de sus bases de datos. De esta forma, es posible aislar la implementación del acceso a las bases de datos para su posterior reutilización en otros proyectos.

La Galería de Plantillas de Delphi 1

El mecanismo original de plantillas se incluyó desde la primera versión de Delphi. En esta versión, el programador puede seleccionar, en el diálogo de configuración del entorno (*Options|Environment*) las opciones *Use on new form* y *Use on new project*, en la página *Preferences*. Cuando las opciones mencionadas están activas y se crea un nuevo formulario o una aplicación, aparece un cuadro de diálogo con *plantillas* y *expertos*. Las plantillas son modelos de formularios o de aplicaciones, de los cuales Delphi almacena el código, en ficheros *pas*, y el diseño de componentes y sus propiedades, en ficheros *dfm*. Cuando se selecciona una plantilla de formulario, se realiza una copia de estos ficheros y se incluyen en nuestro proyecto. De este modo, nos podemos ahorrar el traer los componentes de la plantilla uno por uno, y el tener que configurar sus propiedades y eventos.

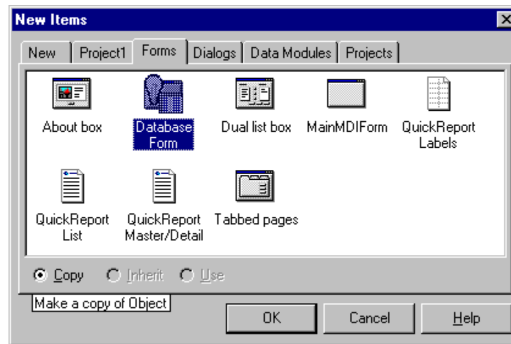


Las plantillas sirven como esqueletos de aplicaciones y formularios pero, aunque tenemos completa libertad de modificar el resultado después de creado, su estilo de uso es: "o lo tomas todo, o lo dejas". En cambio, los expertos son diálogos interactivos que nos permiten especificar las características del formulario o aplicación que queremos generar antes de ser creados. Delphi implementa los expertos mediante DLLs. El programador puede desarrollar sus propios expertos y registrarlos para su uso por Delphi, pero las técnicas necesarias están fuera del alcance de este libro.

Crear una nueva plantilla, por el contrario, es algo sumamente fácil: basta con pulsar el botón derecho del ratón sobre el formulario que se desea guardar como plantilla, y ejecutar el comando *Save as template*. Aparece entonces un cuadro de diálogo para que tecleemos el nombre de la plantilla y una breve descripción de la misma. Cuando terminamos de introducir estos datos, Delphi 1 hace una copia de los ficheros *pas* y *dfm* para su uso posterior desde la Galería. Este detalle es importante: si posteriormente modificamos el formulario que sirvió de origen a la plantilla, los cambios no se reflejan en la misma. A partir de Delphi 2, como veremos, el comportamiento es muy diferente.

El Depósito de Objetos

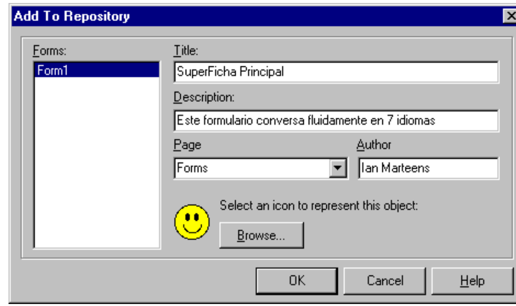
Delphi 2 aumentó la cantidad de tipos de proyectos y formularios con los que el programador podía trabajar. Se introdujeron, por ejemplo, los módulos de datos, que estudiaremos más adelante en este capítulo. Aparecieron también los objetos para la programación concurrente (*thread objects*): la programación de estos objetos no visuales requiere teclear mecánicamente bastante código. En respuesta a la mayor complejidad, Borland modificó el concepto de Galería, transformándose ésta en el *Depósito de Objetos* (*Object Repository*), que asume todas las tareas anteriores de la Galería y añade nuevas posibilidades.



El Depósito de Objetos puede activarse en Delphi mediante el comando de menú *File|New*. Consiste en un cuadro de diálogo modal dividido en páginas. La primera página, titulada simplemente *New*, contiene plantillas de código para aplicaciones, formularios vacíos, componentes, DLLs, unidades vacías, unidades con objetos de concurrencia, etc. Las páginas presentes en el Depósito, y los objetos que se encuentran en las mismas, dependen de la versión de Delphi y de la configuración personal que hayamos realizado al Depósito. Una característica que en mi opinión es negativa, es que no es fácil distinguir ahora las plantillas de los expertos, pues todos estos objetos están mezclados. Una forma de diferenciarlos es observar los botones de radio de la parte inferior del Depósito: si solamente está disponible *Copy* es que hemos seleccionado un experto o una plantilla de proyectos. Más adelante explicaré el uso de estos botones.

Añadir plantillas al Depósito sigue siendo algo sencillo. Se pulsa el botón derecho del ratón y se ejecuta el comando *Add to Repository*. Los datos que se suministran ahora son:

- *Title*: El título de la plantilla. Debe ser corto, pero puede contener espacios en blancos.
- *Description*: Una descripción del contenido y funciones de la plantilla.
- *Page*: Es el nombre de la página donde se va a ubicar inicialmente la plantilla. Se puede crear una página nueva tecleando un nombre no existente, y se puede cambiar posteriormente la localización de la plantilla.
- *Author*: Siempre es agradable que reconozcan nuestros méritos.
- *Icon*: El icono que se muestra asociado al objeto.

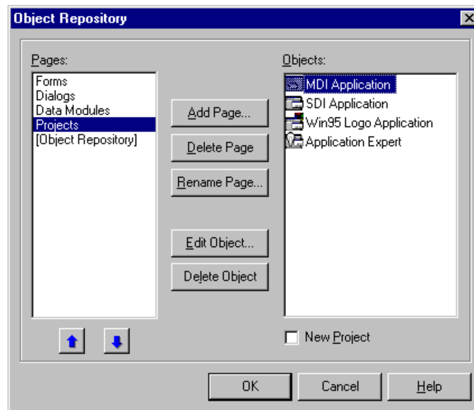


También se pueden guardar proyectos enteros en el Depósito, mediante el comando de menú *Project|Add to Repository*. Los datos que hay que teclear son similares a los de una plantilla de formulario. En este caso se recomienda que la plantilla se sitúe en la página *Projects*.

Las propiedades del Depósito

El Depósito de Objetos se configura mediante un diálogo que aparece al ejecutar el comando de menú *Options|Repository*. El diálogo está dividido en dos cuadros de listas, uno que contiene las páginas existentes en el Depósito, y otro que muestra las plantillas y expertos que corresponden a la página seleccionada. Además de las páginas existentes, se incluye una especie de "super-página" nombrada *Object Repository* que, al estar seleccionada, muestra en la segunda lista todas las plantillas y expertos del Depósito. Mediante los tres botones superiores que quedan entre las dos listas, podemos crear páginas, cambiar sus nombres o eliminarlas; sólo se pueden eliminar páginas cuando ya no contienen objetos.

Igual de sencilla es la configuración de los objetos de página. Con el botón *Edit object* se pueden modificar el título, la descripción, la página, el autor y el icono de una plantilla; si el objeto seleccionado es un experto, no se puede modificar mediante este comando. Ahora bien, si lo único que pretendemos es modificar la página a la que pertenece el objeto, basta con arrastrarlo sobre la página de destino en el primer cuadro de lista. Por último, podemos eliminar plantillas mediante el botón *Delete object*.



En el diálogo de configuración del Depósito de Objetos se puede también especificar cuál es la plantilla de formulario que se utiliza cuando creamos una ficha "en blanco" (comando *File|New form*), y cuál es la ficha que se utiliza por omisión cuando se crea una nueva aplicación (comando *File|New application*). Las casillas *Main form* y *New form*, bajo la lista de objetos, se utilizan para estos fines. Si se selecciona la página *Projects* y se elige una plantilla de proyectos, estas dos casillas se sustituyen por la casilla *New project*. Si marcamos esta casilla para algún proyecto, cuando se cree un nuevo proyecto se utilizará la plantilla seleccionada. Por supuesto, cuando se marca un proyecto como el proyecto por omisión, no tiene sentido tener un formulario con la opción *Main form*.

En cualquier caso, también puede utilizarse un experto como formulario o proyecto por omisión. De ser así, el experto se ejecutaría automáticamente al crearse un nuevo proyecto o formulario.

La ubicación del Depósito de Objetos

El fichero de configuración del Depósito de Objetos se denomina *delphi32.dro*, y reside inicialmente en el directorio *bin* de Delphi. El formato de este fichero es similar al de los ficheros *ini* de configuración de Windows 3.1. El siguiente listado muestra algunas de las opciones almacenadas en *delphi32.dro* para Delphi 3:

```
[C:\ARCHIVOS DE PROGRAMA\BORLAND\DELPHI 3\OBJREPOS\OKCANCL1]
Type=FormTemplate
Name=Standard Dialog
Page=Dialogs
Icon=C:\ARCHIVOS DE PROGRAMA\BORLAND\DELPHI 3\OBJREPOS\OKCANCL1.ICO
Description=OK, Cancel along bottom of dialog.
Author=Borland
DefaultMainForm=0
DefaultNewForm=0
Ancestor=
```

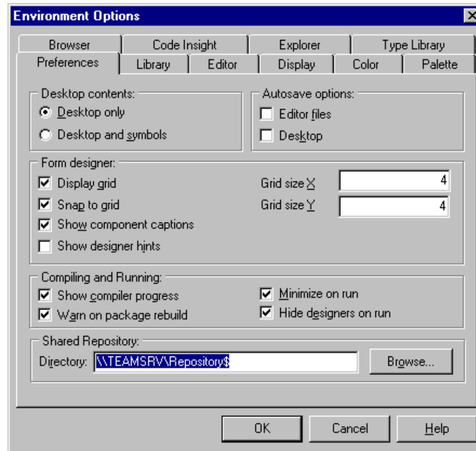
```
[C:\ARCHIVOS DE PROGRAMA\BORLAND\DELPHI 3\OBJREPOS\MDIAPP\MDIAPP]
Type=ProjectTemplate
Name=MDI Application
Page=Projects
Icon=C:\ARCHIVOS DE PROGRAMA\BORLAND\DELPHI
3\OBJREPOS\MDIAPP\MDIAPP.ICO
Description=Standard MDI application frame.
Author=Borland
DefaultProject=0

[Borland.DataBaseFormTemplate]
Type=FormExpert
Page=Business
DefaultMainForm=0
DefaultNewForm=0
```

Como se puede apreciar, las plantillas de formularios se identifican por el nombre del fichero, las de proyecto, por el nombre del directorio donde están ubicados los ficheros, mientras que los expertos se identifican mediante el nombre de la compañía y el nombre del experto. Es evidente que, para los expertos, Delphi necesita otro tipo de información de registro. Esta información, en el caso de Delphi 4, se almacena en la siguiente clave del registro de Windows:

```
[HKEY_CURRENT_USER\Software\Borland\Delphi\4.0\Experts]
```

Para registrar un nuevo experto en Delphi es necesario incluir un valor, bajo la clave anterior, que identifique el nombre y localización del fichero *dll* que contiene al experto.

En Delphi 3 y 4 se puede cambiar el modo de trabajo con el Depósito, permitiendo el uso de un *Depósito Compartido* global (*Shared Repository*). Para activar el Depósito Compartido, hay que ejecutar el comando de menú *Tools|Environment options*, y seleccionar un directorio de la red en la opción *Shared Repository*. El directorio indicado sirve para crear o abrir un fichero *delphi32.dro* con los expertos y plantillas registrados. La ventaja de utilizar un Depósito compartido por un grupo de trabajo de programadores es que el registro de nuevas plantillas se puede realizar desde un puesto arbitrario de la red, con la garantía de que los cambios son percibidos automáticamente por el resto del equipo. La dirección del directorio compartido se almacena de forma persistente en el registro de Windows.



¿Usar, copiar o heredar?

La diferencia más revolucionaria entre la Galería de Delphi 1 y el Depósito de Objetos es la posibilidad de elegir entre distintos modos de uso de una plantilla. El diálogo del Depósito muestra, en su esquina inferior izquierda, tres botones de radio con los títulos *Copy*, *Inherit* y *Use*. La opción *Copy* es la más común, pues era el modo en que funcionaba la Galería de Plantillas de Delphi 1; las plantillas de proyectos, además, solamente admiten esta opción. Consiste en sacar una nueva copia de los ficheros asociados a la plantilla, tanto los *pas* como los *dfm*. El programador puede entonces modificar el código del formulario o proyecto para adaptarlo a sus necesidades. Los problemas surgen si necesitamos modificar la plantilla original. Por ejemplo, la plantilla puede ser algo tan simple como la pantalla de presentación estándar de las aplicaciones de nuestra empresa. Alguien diseñó originalmente la plantilla, y la hemos copiado en diez proyectos. Naturalmente, hay que adaptar la plantilla al proyecto, pues se necesita al menos cambiar el nombre de la aplicación. Entonces, algún desocupado decide, para justificar su salario, que la empresa necesita un nuevo logotipo, y se modifica la plantilla. Tenemos que modificar manualmente las diez pantallas de presentación de los diez proyectos, o comenzar de nuevo.

Cree una nueva aplicación con el comando de menú *File|New application*. Ejecute el comando *File|New* para trabajar con el Depósito de Objetos. Active la página *Forms*, seleccione el primer icono, *About*, y asegúrese de que esté seleccionado el botón *Copy* en la porción inferior izquierda del diálogo. Pulse entonces el botón *Ok*. Debe aparecer entonces, ya como parte de nuestro proyecto, un cuadro de diálogo para mostrar los créditos de la aplicación. Si vamos al Editor de Código, podremos ver la declaración del nuevo tipo de formulario:

```
type
  TAboutBox = class(TForm)
    Panel1: TPanel;
    ProgramIcon: TImage;
    ProductName: TLabel;
    Version: TLabel;
    Copyright: TLabel;
    Comments: TLabel;
    OKButton: TButton;
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

Como se puede apreciar, el código generado es idéntico al que se produciría si el formulario se hubiera creado manualmente y paso a paso dentro del propio proyecto. Abra ahora el Administrador de Proyectos, con el comando de menú *View|Project manager*. La columna *Path* contiene los directorios donde residen los formularios; cuando el formulario está en el mismo directorio que el proyecto (indicado en la barra de estado del Administrador), el valor correspondiente a esa fila aparece vacío. En este caso, comprobamos que tenemos realmente una nueva copia del formulario en nuestro proyecto:

Unidad	Formulario	Directorio
Unit2	*AboutBox1*	

El otro extremo es la opción *Use*. Cuando una de las plantillas del Depósito se "usa", se incluye el fichero original en el proyecto actual. El mismo efecto puede lograrse mediante el comando de menú *Add to project*. Volviendo al ejemplo anterior: las modificaciones de la plantilla se verían inmediatamente en las aplicaciones que la utilizan. Pero no se podría modificar impunemente la plantilla, pues todos estarían trabajando sobre el mismo fichero.

Para que vea el efecto de utilizar una plantilla con la opción *Use*, inicie una nueva aplicación, active el Depósito de Objetos, seleccione la página *Forms*, la opción *Use* y la plantilla *About box*; cierre entonces el cuadro de diálogo con el botón *Ok*. Si vamos ahora al código de la unidad incluida, no encontraremos diferencias con respecto al ejemplo anterior. Ejecute entonces el comando de menú *View|Project manager*, y observe la columna *Path*: solamente el nuevo formulario tiene un valor diferente, indicando el directorio donde se encuentra la plantilla. Para Delphi 4 es:

Unidad	Formulario	Directorio
About	*AboutBox*	*C:\Archivos de programa\Borland\Delphi 4\ObjRepos*

Esto quiere decir que estamos trabajando directamente con el fichero de plantilla, que se encuentra en el directorio utilizado por Delphi para sus plantillas predefinidas. Cualquier modificación en el formulario se realiza sobre el original.

Como en casi todos los problemas, la solución ideal es la "senda intermedia": la opción *Inherit*. Cuando heredamos de una plantilla *incluimos* la plantilla original dentro de nuestro proyecto. Pero la plantilla incluida no se coloca en la lista de creación automática. En cambio, se crea un nuevo formulario o módulo de datos, perteneciente a una clase *derivada por herencia* del módulo o formulario original; así nos llevamos dos productos por el precio de uno. Es sobre este nuevo módulo o formulario sobre el que debemos trabajar.

Repita los pasos anteriores: cree una nueva aplicación, incluya la plantilla anterior desde el Depósito, esta vez utilizando a opción *Inherit*, y active el Administrador de Proyectos. Esta vez tenemos dos nuevas unidades y formularios en el proyecto:

Unidad	Formulario	Directorio
About	*AboutBox*	*C:\Archivos de programa\Borland\Delphi 4\ObjRepos*
Unit2	*AboutBox2*	

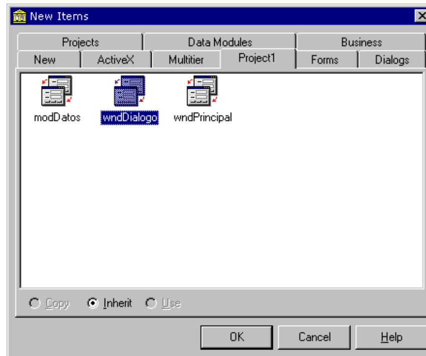
El código generado para el formulario del fichero *unit2.pas* es el siguiente:

```
type
   TAboutBox2 = class(TAboutBox)
   private
      { Private declarations }
   public
      { Public declarations }
   end;
```

La clase *TAboutBox2*, que es el formulario con el cual vamos a trabajar, hereda todos los atributos y métodos de la clase original *TAboutBox*. Los objetos de este nuevo formulario se inicializan con los valores de propiedades y eventos almacenados en el fichero *dfm* de la plantilla original. El mecanismo de herencia impone ciertas restricciones a lo que podemos realizar en el nuevo formulario. Por ejemplo, se puede modificar cualquier propiedad o manejador de eventos, podemos añadir nuevos objetos, pero no podemos eliminar un objeto existente en la plantilla, ni podemos cambiar la propiedad *Name* de los mismos. Otro hecho del que tenemos que tener conciencia es que la plantilla original, aunque se incluye en el proyecto, no aparece en la lista de formularios con creación automática. Para comprobarlo, abra el cuadro de diálogo *Project|Options*, en la página *Forms*, y verifique el contenido de las listas de formularios.

Herencia visual dentro del proyecto

También podemos crear formularios en un proyecto que hereden directamente de otro formulario definido en el mismo proyecto, sin necesidad de que este último haya sido guardado como plantilla. Cuando hay un proyecto abierto en el entorno de desarrollo, el Depósito de Objetos muestra una página con el nombre del proyecto: la segunda en Delphi2, y la tercera en Delphi 3. En la edición anterior de este libro, pronostiqué en broma que sería la cuarta página en Delphi 4, y se cumplió la profecía. La página muestra un icono por cada formulario de la aplicación, y solamente deja utilizar la opción *Inherit*.



Cuando utilizamos la herencia visual dentro de un mismo proyecto, es mucho más fácil modificar la plantilla original que contiene la clase base. En este caso, además, no es necesario eliminar la plantilla base de la lista de formularios con creación automática.

Herencia de eventos

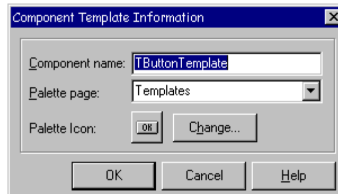
Cuando un formulario hereda de una plantilla cambia la forma en que se crean los manejadores de eventos por omisión, al realizar una doble pulsación en un evento en el Inspector de Objetos. Por ejemplo, abra el Depósito de Objetos, seleccione cualquier plantilla de la página *Forms*, marque también la opción *Inherit* y cree un nuevo formulario. Ahora vaya al Inspector de Objetos y cree un nuevo manejador para un evento arbitrario del nuevo formulario, digamos que sea el evento *OnCreate*. He aquí el código que genera Delphi:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
   inherited;
   { Aquí se sitúa el cursor }
end;
```

La instrucción **inherited** permite ejecutar algún posible método asociado a este evento en la clase ancestro. ¿Por qué escribimos solamente **inherited**, sin aclarar siquiera el nombre del método o los parámetros del mismo? El problema es que puede que no sepamos si existe un manejador anterior para el evento que se está interceptando y, si existe, cuál es su nombre. El código generado para la instrucción **inherited** se hace cargo automáticamente de este asunto.

Las plantillas de componentes

Las plantillas de componentes son un recurso, añadido a partir de Delphi 3, que nos permite guardar grupos de componentes junto con sus propiedades y eventos asociados. Los grupos no se almacenan en el Depósito de Objetos, sino directamente en la Paleta de Componentes, como si fueran componentes normales. La creación de una plantilla de componentes es muy fácil: cuando haya configurado los componentes necesarios, modificando sus propiedades y creando respuestas para sus eventos, realice una selección múltiple de los mismos y ejecute el comando de menú *Components|Create component template*. Aparece un cuadro de diálogo para que indiquemos el nombre que se le va a dar a la plantilla, en qué página de la Paleta debe colocarse (por omisión, la página *Templates*), y el icono asociado a la plantilla. Para este icono, Delphi utiliza por omisión el del primer componente seleccionado.



Para borrar una plantilla de componentes de la Paleta, utilice el diálogo de configuración de este objeto, que se puede ejecutar pulsando el botón derecho del ratón sobre la Paleta de Componentes y seleccionando el comando *Properties*. Se busca la página y el objeto, y se pulsa la tecla SUPR.

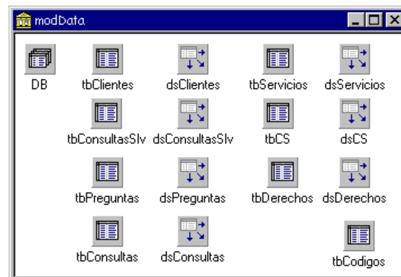
> Las plantillas de componentes de Delphi 3 tenían una característica bastante irritante. Se tomaban demasiado al pie de la letra lo de guardar *todas* las propiedades asignadas, por lo cual también memorizaban la posición absoluta dónde estaban situados los componentes originales. Delphi 4 ha corregido este pequeño defecto, y ahora el comportamiento de las plantillas de componentes se asemeja más al de los componentes verdaderos.

Los módulos de datos

Los módulos de datos se incluyeron en Delphi 2, y ofrecen un mecanismo idóneo para separar los elementos no visuales de aplicación de la interfaz puramente visual. Desde el punto de vista del programador son una especie de "formularios invisibles", en los que se puede colocar cualquier componente que no sea un control. Para crear un módulo de datos en un proyecto, podemos activar el diálogo del Depósito y seleccionar *Data module* en la primera página. En Delphi 2 y 3 es preferible ir directamente al comando de menú *File|New data module*, cuyo efecto es el mismo; lamentablemente, este comando ha desaparecido en Delphi 4. Realice el siguiente experimento: cree un módulo de datos en un proyecto, e intente añadir un botón al módulo. Debe obtener un error con el siguiente mensaje: *"Controls cannot be added to a data module"*. No obstante, sí admite que se coloque un componente *TMainMenu*, pues este componente no es técnicamente un control: no se deriva de *TControl*.

Desde el punto de vista de su implementación, los módulos de datos son objetos pertenecientes a la clase *TDataModule*, que desciende de *TComponent*. Las únicas propiedades publicadas son el nombre (*Name*) y la omnipresente propiedad *Tag*, aunque Delphi 4 añade *OldCreateOrder*. También son dos los eventos asociados: *OnCreate* y *OnDestroy*, para poder inicializar los componentes asociados o realizar alguna acción antes de la destrucción del módulo.

Los módulos de datos se utilizan típicamente para contener componentes de acceso a bases de datos. En un módulo se puede situar un componente *TDatabase*, para controlar la conexión a la base de datos, y varios objetos *TTable* para utilizar tablas de la BD. A partir de las tablas, a su vez, se crean objetos no visuales para acceder a los campos. En estos objetos configuramos valores por omisión, verificaciones y formatos de visualización. También se pueden colocar en el módulo componentes *TQuery* para realizar consultas en la base de datos. Todos los componentes mencionados se estudiarán más adelante.



El sentido de esta "aglomeración" de componentes es que el resto de los formularios de la aplicación pueden acceder posteriormente a los mismos. Por ejemplo, una rejilla de datos de Delphi (el componente *TDBGrid*) necesita un puntero a un objeto no

visual, de tipo *TDataSource*, para extraer los datos que se visualizan; el componente *TDataSource*, por su parte, necesita un puntero a una tabla (*TTable*). En los lejanos tiempos de Delphi 1 estos tres componentes tenían que residir dentro del mismo formulario, casi de forma obligatoria, por una limitación fundamental que padecía esta versión:

"En Delphi 1 no puede haber referencias, en tiempo de diseño, entre componentes situados en distintos formularios"

En cambio, las versiones de Delphi a partir de la 2 *sí* permiten que un componente tenga como valor de una propiedad un puntero a otro componente situado en otro formulario; únicamente es necesario que la unidad externa esté en uso por la unidad en que se encuentra el primer componente. Además, esta posibilidad no está limitada a referencias a módulos de datos, sino que puede realizarse entre formularios o módulos arbitrarios.

Hagamos una pequeña demostración sin utilizar componentes de bases de datos. Cree un aplicación y coloque en la ventana principal, *Form1*, un componente *TDirectoryListBox*, de la página de la Paleta *Win3.1*. Cree un nuevo formulario, *Form2*, e incluya en su interior un componente *TFileListBox*, que se encuentra en la misma página: *Win3.1*. Cambie también la propiedad *Visible* del formulario a *True*. Regrese ahora a *Form1* y seleccione la lista de directorios. Este componente tiene una propiedad *FileList*, que contiene un puntero al componente *TFileListBox* que debe ser controlado. Como en todas las propiedades de este tipo, el editor de la propiedad en el Inspector de Objetos tiene un botón con una flecha para activar la lista desplegable con los posibles valores. Consecuentemente, despliegue esa lista: no aparecerán valores. La explicación es que dentro del formulario en edición no hay componentes del tipo necesario.

Ahora active el comando de menú *File|Use unit*, e incluya la unidad *Unit2*, que contiene al objeto *Form2*. Con esta operación, Delphi añade la siguiente línea dentro de la sección de implementación de la primera unidad:

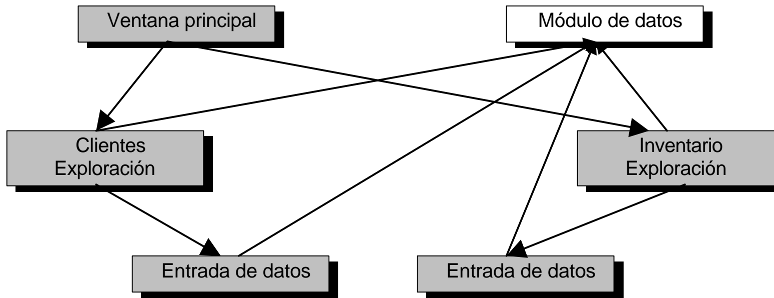
```
uses Unit2;
```

Regrese al Inspector de Objetos y vuelva a desplegar la lista de valores de la propiedad *FileList*. Esta vez, aparece el objeto situado en la segunda unidad: *Form2.FileListBox1*. Si estuviéramos en Delphi 1, no podríamos efectuar en tiempo de diseño la operación anterior, y estaríamos obligados a enlazar los objetos en tiempo de ejecución, quizás en el evento *OnCreate* de *Form2*:

```
uses Unit1;
```

```
procedure TForm2.FormCreate(Sender: TObject);
begin
   Form1.DirectoryListBox1 := FileListBox1;
end;
```

El esquema que presento a continuación muestra un ejemplo típico de aplicación que agrupa sus tablas en módulos de datos:



Estoy suponiendo que la aplicación realizará un mantenimiento de dos tablas: una de clientes y otra de inventario. Desde la ventana principal se lanzan las ventanas de exploración (*browsing*), que permiten localizar y filtrar las tablas correspondientes; es común que estas ventanas estén basadas en rejillas de datos (*TDBGrid*). A su vez, la actualización de estas tablas (altas y modificaciones), se realiza en ventanas de entrada de datos, en las cuales se trabaja con sólo un registro a la vez. Lo interesante de esta estructura es que todas las ventanas extraen sus datos del módulo de datos. Si tuviéramos, como en Delphi 1, una tabla en la ventana de exploración y otra en la de entrada de datos, sería trabajoso mantener la sincronización entre ambas. La alternativa sería utilizar una sola tabla, quizás en la ventana de exploración, y conectar la tabla en tiempo de ejecución a los controles de datos de la segunda ventana. Pero de esta forma nos perderíamos la posibilidad de controlar, en tiempo de diseño, el aspecto de los datos.

El módulo de datos desarrollado para esta aplicación puede colocarse en el Depósito Compartido, y de este modo puede incluirse en aplicaciones posteriores. Es muy probable que, en un equipo de trabajo, la programación del módulo de datos quede a cargo de una parte del equipo, mientras que la parte visual sea responsabilidad de la otra parte.

3

Componentes para bases de datos

- Conjuntos de datos: tablas
- Acceso a campos
- Controles de datos y fuentes de datos
- Rejillas y barras de navegación
- Indices
- Métodos de búsqueda

Parte

Capítulo

15

Conjuntos de datos: tablas

UN *CONJUNTO DE DATOS*, para Delphi, es cualquier fuente de información estructurada en filas y columnas. Este concepto abarca tanto a las tablas "reales" y las consultas SQL como a ciertos tipos de procedimientos almacenados. Pero también son conjuntos de datos los *conjuntos de datos clientes*, introducidos en Delphi 3, que obtienen su contenido por medio de automatización OLE remota, o a partir de un fichero "plano" local. Y también lo son las tablas anidadas de Delphi 4, y los conjuntos de datos a la medida que desarrollan otras empresas para acceder a formatos de bases de datos no reconocidos por el Motor de Datos de Borland. Todos estos objetos tienen muchas propiedades, métodos y eventos en común. En este capítulo estudiaremos los conjuntos de datos en general, pero haremos énfasis en las propiedades específicas de las tablas. En capítulos posteriores, nos ocuparemos de las consultas y los procedimientos almacenados.

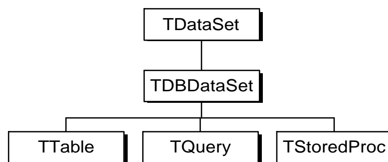
La jerarquía de los conjuntos de datos

La clase *TDataSet* representa una mayor abstracción del concepto de conjunto de datos, sin importar en absoluto su implementación física. Esta clase define características y comportamientos comunes que son heredados por clases especializadas. Estas características son, entre otras:

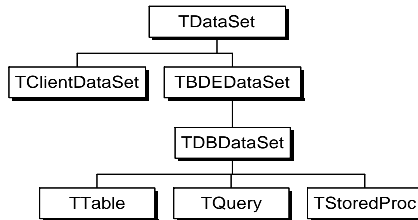
- *Métodos de navegación*: Un conjunto de datos es una colección de registros homogéneos. De estos registros, siempre hay un *registro activo*, que es el único con el que podemos trabajar directamente. Los métodos de navegación permiten gestionar la posición de este registro activo.
- *Acceso a campos*: Todos los registros de un conjunto de datos están estructurados a su vez en campos. Existen mecanismos para descomponer la información almacenada en el registro activo de acuerdo a los campos que forman la estructura del conjunto de datos.
- *Estados del conjunto de datos*: Los conjuntos de datos implementan una propiedad *State*, que los transforman en simples autómatas finitos. Las transiciones entre estados se utilizan para permitir las altas y modificaciones dentro de los conjuntos de datos.

- *Notificaciones a componentes visuales*: Uno de los subsistemas más importantes asociados a los conjuntos de datos envía avisos a todos los componentes que se conectan a los mismos, cada vez que cambia la posición de la fila activa, o cuando se realizan modificaciones en ésta. Gracias a esta técnica es posible asociar controles de edición y visualización directamente a las tablas y consultas.
- *Control de errores*: Los conjuntos de datos disparan eventos cuando detectan un error, que permiten corregir y reintentar la operación, personalizar el mensaje de error o tomar otras medidas apropiadas.

Es interesante ver cómo ha evolucionado la jerarquía de clases a través de la historia de Delphi. En las dos primeras versiones del producto, éstas eran las únicas clases existentes:

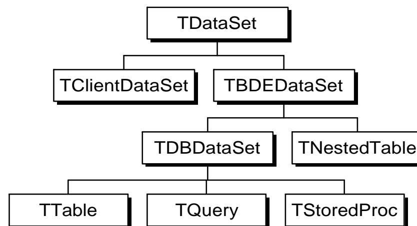


La implementación de la clase *TDataSet*, en aquella época, utilizaba funciones y estructuras de datos del BDE. Además, la relación mutua entre los métodos, eventos y propiedades de esta clase hacían que la implementación de descendientes de la misma fuera bastante fastidiosa. Estos problemas fueron reconocidos durante el desarrollo de Delphi 3, que modificó del siguiente modo la jerarquía de clases:



A partir de este momento, *TDataSet* pasó a ser totalmente independiente del BDE. La definición de esta clase reside ahora en la unidad *DB*, mientras que las clases que dependen del BDE (derivadas de *TBDEDataSet*) se han movido a la unidad *DBTables*. Si desarrollamos un programa que no contenga referencias a la unidad *DBTables* (ni a la unidad *BDE* de bajo nivel, por supuesto) no necesitaremos incluir al Motor de Datos en la posterior instalación de la aplicación. Esto es particularmente cierto para las aplicaciones clientes de Midas, que se basan en la clase *TClientDataSet*. Como esta clase desciende directamente de *TDataSet*, no necesita la presencia del BDE para su funcionamiento.

¿Por qué hay dos clases diferentes: *TBDEDataSet* y *TDBDataSet*, si la segunda se deriva directamente de la primera y no tiene hermanos? La clase *TDBDataSet* introduce la propiedad *Database* y otras propiedades y métodos relacionados con la misma. No todo conjunto de datos del BDE tiene que estar asociado a una base de datos. A mí se me ocurre pensar en las tablas en memoria del BDE (no encapsuladas aún en conjuntos de datos). A los desarrolladores de Delphi 4 se les ocurrió pensar en *tablas anidadas*, para representar el nuevo tipo de campo de Oracle 8, que puede contener una colección de registros anidados:



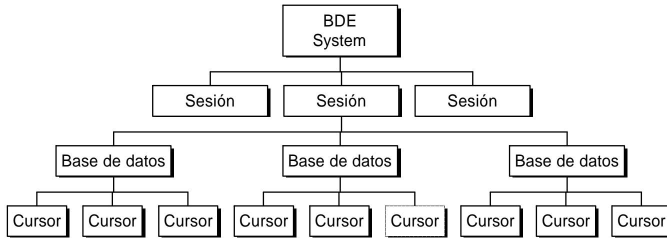
El cambio de arquitectura realizado en Delphi 3 ha sido determinante para la creación de sustitutos del BDE, como lo son en este momento Titan (Btrieve y Access), Apollo (Clipper y FoxPro), DOA (Oracle) y otros muchos.

La interacción de los subsistemas de los conjuntos de datos con los demás componentes de la VCL es bastante compleja, y existen muchas dependencias circulares. Para explicar el funcionamiento de los campos, necesitamos saber cómo funcionan las tablas, y viceversa. En este capítulo nos limitaremos al estudio de los métodos de navegación. El capítulo siguiente tratará sobre el acceso a campos. Cuando estudiemos los controles de datos, profundizaremos en las notificaciones a componentes visuales. Por último, al estudiar las actualizaciones veremos en detalle las transiciones de estado y el mecanismo de control de errores.

En vez de comenzar la explicación con la clase abstracta *TDataSet*, lo cual haría imposible mostrar ejemplos, utilizaremos la clase *TTable*, que es además el componente de acceso a datos que encontraremos con mayor frecuencia.

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. En realidad, el BDE reconoce internamente otro tipo de objetos: los clientes. Pero como estos objetos no son percibidos por el programador de Delphi, agruparemos sus funciones con las de las sesiones.

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.

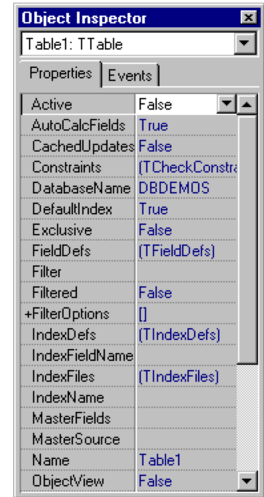
Por último, una vez que hemos accedido a una base de datos, estamos preparados para trabajar con los cursores. Un *cursor* es una colección de registros, de los cuales tenemos acceso a uno solo a la vez, por lo que puede representarse mediante los conjuntos de datos de Delphi. Existen funciones y procedimientos 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 tipos de objetos internos del BDE descritos en el párrafo anterior tiene un equivalente directo en la VCL de Delphi. La excepción es el nivel principal, el de sistema, algunas de cuyas funciones son asumidas por la clase de sesiones de Delphi:

Objeto del BDE	Clase de la VCL
Sesiones	*TSession*
Bases de datos	*TDatabase*
Cursores	*TBDEDataSet*

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 Delphi, aunque el programador puede acceder a los mismos en tiempo de ejecución. Es por esto que podemos postergar el estudio de casi todos estos objetos.

Tablas



Para comenzar el estudio de los conjuntos de datos utilizaremos las tablas: el componente *TTable*. Mediante este componente podemos conectarnos a tablas en cualquiera de los formatos reconocidos por el BDE. El componente *TTable* también permite conectarnos a una *vista* definida en una bases de datos SQL. Las vistas son tablas virtuales, definidas mediante una instrucción **select**, cuyos valores se extraen de otras tablas y vistas. Para más información, puede leer el capítulo 26, que trata sobre las consultas en SQL.

La configuración de una tabla es muy sencilla. Primero hay que asignar el valor de la propiedad *DatabaseName*. En esta propiedad se indica el nombre del alias del BDE donde reside la tabla. Este alias puede ser un alias persistente, como los que creamos con la utilidad de configuración del BDE, o un alias local a la aplicación, creado con el componente *TDatabase*. Esta última técnica se estudia en el capítulo 30, que trata sobre los componentes de bases de datos y sesiones. Es posible también, si la tabla está en formato Paradox o dBase, asignar el nombre del directorio a esta propiedad. No es, sin embargo, una técnica recomendable pues hace más difícil cambiar dinámicamente la ubicación de las tablas. El siguiente ejemplo muestra cómo asignar un nombre de directorio extraído del registro de Windows a la propiedad *DatabaseName* de una tabla:

```
uses
   Registry;   // Unidad necesaria para trabajar con el registro

resourcestring
   SRaizRegistro = 'Software\MiEmpresa\MiAplicacion';

procedure TForm1.FormCreate(Sender: TObject);
begin
   with TRegIniFile.Create(SRaizRegistro) do
   try
      Table1.DatabaseName := ReadString('BaseDatos', 'Dir', '');
      // …
      // Código necesario para terminar de configurar la tabla
      // …
```

```
      finally
        Free;
      end;
    end;
```

Una vez que tenemos asignado el nombre del alias o del directorio, podemos especificar el nombre de la tabla dentro de esa base de datos mediante la propiedad *TableName*. Las tablas de Paradox y dBase se almacenan en ficheros, por lo que los nombres de estas tablas llevan casi siempre su extensión, *db* ó *dbf*. Sin embargo, es preferible no utilizar extensiones para este tipo de tablas. La razón es que si no utilizamos extensiones podemos cambiar la base de datos asociada al alias, quizás con la configuración del BDE, y nuestra aplicación podrá trabajar también con tablas en formato SQL. Esta técnica se ilustra en la aplicación *mastapp*, que se instala en el directorio de demostraciones de Delphi.

Si no utilizamos extensiones con tablas locales, tenemos una propiedad y un parámetro del BDE para decidir el formato de la tabla. Si la propiedad *TableType* de la tabla vale *ttDefault*, que es el valor por omisión, la decisión se realiza de acuerdo al parámetro *DEFAULT DRIVER* del alias, que casi siempre es *PARADOX*. Pero también podemos asignar *ttDBase*, *ttParadox* ó *ttASCII* a *TableType*, para forzar la interpretación según el formato indicado:

```
Table1.DatabaseName := 'DBDEMOS';
// Esta tabla tiene extensión DBF
Table1.TableName := 'ANIMALS';
// Sin esta asignación, falla la apertura
Table1.TableType := ttDBase;
```

Para poder extraer, modificar o insertar datos dentro de la tabla, necesitamos que la tabla esté abierta o activa. Esto se controla mediante la propiedad *Active* de la clase. También tenemos los métodos *Open* y *Close*, que realizan asignaciones a *Active*; el uso de estos métodos hace más legible nuestros programas.

> En determinados sistemas cliente/servidor, como Oracle y MS SQL Server, los nombres de tablas pueden ir precedidos por el nombre del propietario de la tabla. Por ejemplo, *dbo.customer* significa la tabla *customer* creada por el usuario *dbo*, es decir, el propio creador de la base de datos.
>
> ¡No elimine el prefijo de usuario del nombre de la tabla, aunque el truco parezca funcionar! El BDE necesita toda esta información para localizar los índices asociados a la tabla. Sin estos índices, puede que la tabla no pueda actualizarse, o que ocurran problemas (en realidad *bugs*) al cambiar dinámicamente el criterio de ordenación. Esto es especialmente aplicable a MS SQL Server.

Active es una propiedad que está disponible en tiempo de diseño. Esto quiere decir que si le asignamos el valor *True* durante el diseño, la tabla se abrirá automáticamente

al cargarse desde el fichero *dfm*. También significa que mientras programamos la aplicación, podemos ver directamente los datos tal y como van a quedar en tiempo de ejecución; esta importante característica no está presente en ciertos sistemas RAD de cuyo nombre no quiero acordarme. No obstante, nunca está de más abrir explícitamente las tablas durante la inicialización del formulario o módulo de datos donde se ha definido. La propiedad *Active* puede, por accidente, quedarse en *False* por culpa de un error en tiempo de diseño, y de este modo garantizamos que en tiempo de ejecución las tablas estén abiertas. Además, aplicar el método *Open* sobre una tabla abierta no tiene efectos negativos, pues la llamada se ignora. En cuanto a cerrar la tabla, el destructor del componente llama automáticamente a *Close*, de modo que durante la destrucción del formulario o módulo donde se encuentra la tabla, ésta se cierra antes de ser destruida.

La propiedad *Exclusive* permite abrir una tabla en modo exclusivo, garantizando que solamente un usuario esté trabajando con la misma. Por supuesto, la apertura en este modo puede fallar, produciéndose una excepción. *Exclusive*, sin embargo, sólo funciona con Paradox y dBase. Del mismo modo, para estos formatos tenemos los siguientes métodos, que intentan aplicar un bloqueo global sobre la tabla, una vez que está abierta:

```
type
   TLockType = (ltReadLock, ltWriteLock);

procedure LockTable(LockType: TLockType);
procedure UnlockTable(LockType: TLockType);
```

> No cambie el valor de *Exclusive* en tiempo de diseño, pues impediría la apertura de la tabla al depurar el programa. Si desea hacer pruebas en este sentido, debe salir del entorno de desarrollo antes.

Otra propiedad relacionada con el modo de apertura de una tabla es *ReadOnly*, que permite abrir la tabla en el modo sólo lectura.

Un poco antes he mostrado un ejemplo en el que la propiedad *DatabaseName* se asigna en tiempo de ejecución. ¿Cómo hacer entonces para poder visualizar los datos en tiempo de diseño, y no tener que programar a ciegas? Si hacemos *Active* igual a *True* en el formulario, cuando se inicie la aplicación tendremos que cerrar la tabla antes de modificar *DatabaseName*, y el usuario notará el parpadeo del monitor; sin contar que el directorio de pruebas que utilizamos durante el diseño puede no existir en tiempo de ejecución. La solución consiste en realizar el cambio de la propiedad *DatabaseName* durante la respuesta al evento *BeforeOpen*, que se activa justo antes de abrir la tabla. Este evento puede compartirse por todos los componentes cuya base de datos se determine dinámicamente:

```
procedure TForm1.TablasBeforeOpen(DataSet: TDataSet);
begin
   with DataSet as TTable do
      with TRegIniFile.Create(SRaizRegistro) do
      try
         DatabaseName := ReadString('BaseDatos', 'Dir', '');
      finally
         Free;
      end;
end;
```

La apertura de la tabla tiene lugar automáticamente después de terminar la ejecución de este método. Observe que el parámetro del evento es del tipo *TDataSet*. Este evento y otros similares serán estudiados en un capítulo posterior.

En lo que queda de este capítulo, las propiedades, métodos y eventos que vamos a estudiar serán comunes a todos los tipos de conjuntos de datos, aunque utilizaremos tablas en los ejemplos para concretar.

Conexión con componentes visuales

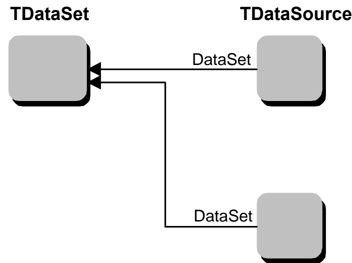
Un conjunto de datos, sea una tabla o una consulta, no puede visualizar directamente los datos con los que trabaja. Para comunicarse con los controles visuales, el conjunto de datos debe tener asociado un componente auxiliar, perteneciente a la clase *TDataSource*. Traduciré esta palabra como *fuente 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 "notificador". Los objetos que se conectan a este componente son avisados de los cambios de estado y de contenido del conjunto de datos controlado por *TDataSource*. Las dos propiedades principales de *TDataSource* son:

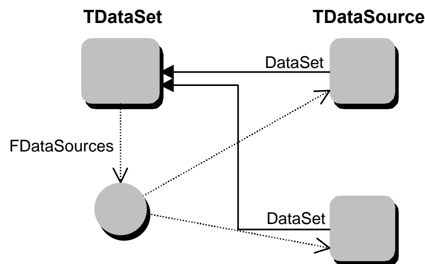
- *DataSet*: Es un puntero, de tipo *TDataSet*, al conjunto de datos que se controla.
- *AutoEdit*: Cuando es *True*, el valor por omisión, permite editar directamente sobre los controles de datos asociados, sin tener que activar explícitamente el modo de edición en la tabla. El modo de edición se explica más adelante.

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. Más adelante, dedicaremos un par de capítulos al estudio de estos controles.

Es posible acoplar más de una fuente de datos a un conjunto de datos. Estas fuentes de datos pueden incluso encontrarse en formularios o módulos de datos diferentes al del conjunto de datos. El propósito de esta técnica es establecer canales de notificación separados. Más adelante, en este mismo capítulo, veremos una aplicación en las relaciones *master/detail*. La técnica de utilizar varias fuentes de datos es posible gracias al mecanismo oculto que emplea Delphi. Cuando usted engancha un *data source* a un conjunto de datos esto es lo que ve:

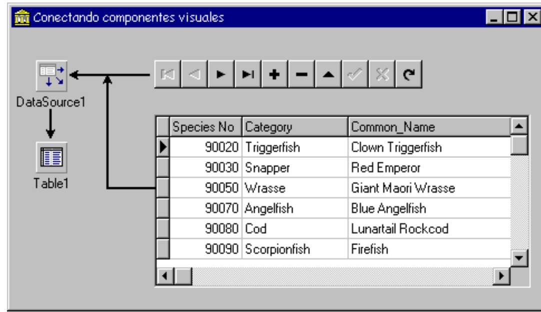


Sin embargo, existe un objeto oculto que pertenece al conjunto de datos, que es una lista de fuentes de datos y que completa el cuadro real:



Un ejemplo común de conexión de componentes es utilizar una *rejilla de datos* (*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
Table1: TTable	*DatabaseName*	El alias de la base de datos
	TableName	El nombre de la tabla
	Active	*True*
DataSource1: TDataSource	*DataSet*	*Table1*
DBGrid1: TDBGrid	*DataSource*	*DataSource1*
DBNavigator1: TDBNavigator	*DataSource*	*DataSource1*



Es tan frecuente ver a estos componentes juntos, que le recomiendo al lector que guarde la combinación como una plantilla de componentes, según explicamos en el capítulo 14.

Navegando por las filas

La mayor parte de las operaciones que se realizan sobre un conjunto de datos se aplican sobre la *fila activa* de éste, en particular aquellas operaciones que recuperan o modifican datos. Los valores correspondientes a las columnas de la fila activa se almacenan internamente en un *buffer*, del cual los campos extraen sus valores.

Al abrir un conjunto de datos, inicialmente se activa su primera fila. Qué fila es ésta depende de si el conjunto está ordenado o no. Si se trata de una tabla con un índice activo, o una consulta SQL con una cláusula de ordenación **order by**, el criterio de ordenación es, por supuesto, el indicado. Pero el orden de los registros no queda tan claro cuando abrimos una tabla sin índices activos. Para una tabla SQL el orden de las filas es aún más impredecible que para una tabla de Paradox o dBase. De hecho, el concepto de posición de registro no existe para las bases de datos SQL, debido a la forma en que generalmente se almacenan los registros. Este es el motivo por el cual la barra de desplazamiento vertical de las rejillas de datos de Delphi tienen sólo tres posiciones cuando se conectan a un conjunto de datos SQL: al principio de la tabla, al final o en el medio.

Los métodos de movimiento son los siguientes:

Método	Objetivo
First	Ir al primer registro
Prior	Ir al registro anterior
Next	Ir al registro siguiente
Last	Ir al último registro
MoveBy	Moverse la cantidad de filas indicada en el parámetro

Hay dos funciones que nos avisan cuando hemos llegado a los extremos de la tabla:

Función	Significado
BOF	¿Estamos en el principio de la tabla?
EOF	¿Estamos al final de a tabla?

Combinando estas funciones con los métodos de posicionamiento, podemos crear los siguientes algoritmos de recorrido de tablas:

Hacia delante

```
Table1.First;
while not Table1.EOF do
begin
   // Acción
   Table1.Next;
end;
```

Hacia atrás

```
Table1.Last;
while not Table1.BOF do
begin
   // Acción
   Table1.Prior;
end;
```

La combinación de las funciones *BOF* y *EOF* nos permite saber si un conjunto de datos está vacío o no; a partir de Delphi 3 se introduce la función *IsEmpty* que realiza esta tarea con más eficiencia:

```
procedure TForm1.Operaciones1Click(Sender: TObject);
begin
   Bajas1.Enabled := not Table1.IsEmpty;
   // En Delphi 2:
   // Bajas1.Enabled := not (Table1.BOF and Table1.EOF);
   Modificaciones1.Enabled := Bajas1.Enabled;
end;
```

Si queremos saber la cantidad de filas de un cursor, podemos utilizar el método *RecordCount*. Esta función, sin embargo, debe aplicarse con cautela, pues si estamos tratando con una consulta SQL, su ejecución puede forzar la evaluación completa de la misma, lo cual puede consumir bastante tiempo la primera vez. A partir de Delphi 3, la evaluación de la propiedad *RecordCount* se realiza en el servidor si el conjunto de datos es un *TTable*, ejecutándose la instrucción SQL siguiente:

```
select count(*)
from   TableName
```

Existe toda una variedad de métodos adicionales para cambiar la fila activa, relacionados con búsquedas por contenido, que se estudiarán en los capítulos que tratan sobre índices y métodos de búsqueda. Por el momento, solamente mencionaré uno más:

```
procedure TTable.GotoCurrent(OtraTable: TTable);
```

Este método se puede utilizar cuando dos componentes de tablas están trabajando sobre la misma tabla "física", y deseamos que una de ella tenga la misma fila activa que la otra. Puede que una de estas tablas tenga filtros y rangos activos, mientras que la otra no. Veremos una aplicación de *GotoCurrent* en un capítulo posterior, para realizar búsquedas sobre tablas de detalles.

Marcas de posición

Cuando cambiamos la fila activa de una tabla, es importante saber cómo regresar a nuestro lugar de origen. Delphi nos permite recordar una posición para volver más adelante a la misma mediante la técnica de *marcas de posición*. Este es un mecanismo implementado a nivel del BDE. La VCL nos ofrece un tipo de datos, *TBookmark*, que es simplemente un puntero, y tres métodos que lo utilizan:

```
function TDataSet.GetBookmark: TBookmark;
procedure TDataSet.GotoBookmark(B: TBookmark);
procedure TDataSet.FreeBookmark(B: TBookmark);
```

El algoritmo típico con marcas de posición se muestra a continuación. Tome nota del uso de la instrucción **try...finally** para garantizar el regreso y la destrucción de la marca:

```
var
   BM: TBookmark;
begin
   // Recordar la posición actual
   BM := Table1.GetBookmark;
   try
      // Mover la fila activa
   finally
      // Regresar a la posición inicial
      Table1.GotoBookmark(BM);
      // Liberar la memoria ocupada por la marca
      Table1.FreeBookmark(BM);
   end;
end;
```

A partir de Delphi 2, se simplifica el trabajo con las marcas de posición, al introducirse el nuevo 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 := Table1.Bookmark;
   try
      // Mover la fila activa
```

```
   finally
      // Regresar a la posición inicial
      Table1.Bookmark := BM;
   end;
end;
```

> *TBookmarkStr* está implementada como una cadena de caracteres larga, a la cual se le aplica una conversión de tipos estática. De esta manera, Delphi aprovecha el código generado automáticamente por el compilador para liberar la memoria asociada a la marca, evitándonos el uso de *FreeBookmark*.

Encapsulamiento de la iteración

Como los algoritmos de iteración o recorrido son tan frecuentes en la programación para bases de datos, es conveniente contar con algún tipo de recurso que nos ahorre teclear una y otra vez los detalles repetitivos de esta técnica. Podemos crear un procedimiento que, dado un conjunto de datos en general, lo recorra fila a fila y realice una acción. ¿Qué acción? Evidentemente, cuando programamos este procedimiento no podemos saber cuál; la solución consiste en pasar la acción como parámetro al procedimiento. El parámetro que indica la acción debe ser un puntero a una función o a un método. Es preferible utilizar un puntero a método, porque de este modo se puede aprovechar el estado del objeto asociado al puntero para controlar mejor las condiciones de recorrido. La declaración de nuestro procedimiento puede ser:

```
procedure RecorrerTabla(ADataSet: TDataSet; Action: TEventoAccion);
```

El tipo del evento, *TEventoAccion*, se debe haber definido antes del siguiente modo:

```
type
   TEventoAccion = procedure(DataSet: TDataSet; var Stop: Boolean)
      of object;
```

Recuerde que **of object** indica que *TEventoAccion* debe apuntar a un método, y no a un procedimiento declarado fuera de una clase.

En el evento pasaremos el conjunto de datos como primer parámetro. El segundo parámetro, de entrada y salida, permitirá que el que utiliza el procedimiento pueda terminar el recorrido antes de alcanzar el fin de tabla, asignando *True* a este parámetro. Finalmente, ésta es la implementación del procedimiento:

```
procedure RecorrerTabla(ADataSet: TDataSet; Action: TEventoAccion);
var
   BM: TBookmarkStr;
   Stop: Boolean;
begin
   Screen.Cursor := crHourglass;
```

```
        ADataSet.DisableControls;
        BM := ADataSet.Bookmark;
        try
           ADataSet.First;
           Stop := False;
           while not ADataSet.EOF and not Stop do
           begin
              if Assigned(Action) then
                 Action(ADataSet, Stop);
              ADataSet.Next;
           end;
        finally
           ADataSet.Bookmark := BM;
           ADataSet.EnableControls;
           Screen.Cursor := crDefault;
        end;
     end;
```

En este procedimiento se ha añadido el cambio de cursor durante la operación. El cursor *crHourglass* es el famoso reloj de arena que con tanta frecuencia, desgraciadamente, aparece en la pantalla de nuestros ordenadores. Además se han introducido un par de métodos nuevos: *DisableControls* y *EnableControls*. Estos métodos desactivan y reactivan el mecanismo interno de notificación a los controles de datos asociados. Si estos métodos no se utilizan en la iteración y la tabla tiene controles de datos asociados, cada vez que desplacemos la fila activa, los controles se redibujarán. Esto puede ser molesto para el usuario, y es sumamente ineficiente. Hay que garantizar, no obstante, que *EnableControls* vuelva a habilitar las notificaciones, por lo que este método se llama dentro de la cláusula **finally**.

NOTA IMPORTANTE

El hecho de que muchos ejemplos escritos en Delphi utilicen métodos de navegación como los anteriores, no le da patente de corso para abusar de los mismos en sus programas, especialmente si está trabajando con una base de datos cliente/servidor. Si usted tiene un bucle de navegación en cuyo interior no existe interacción alguna con el usuario, debe convertirlo lo antes posible en un procedimiento almacenado que se ejecute en el servidor. Así, los registros leídos no circularán por la red, sus programas se ejecutarán más rápido, ahorrarán energía eléctrica y salvarán la selva del Amazonas y la capa de ozono (¡!).

Una posible excepción es que los registros se encuentren en la memoria caché de la máquina, porque estemos trabajando con actualizaciones en caché. Y por supuesto, si estamos usando Paradox o dBase, no podemos utilizar procedimientos almacenados. Sin embargo, aquí puede sernos útil desarrollar aplicaciones distribuidas, como veremos más adelante en este libro.

La relación master/detail

Es bastante frecuente encontrar tablas dependientes entre sí mediante una relación *uno/muchos*: a una fila de la primera tabla corresponden cero, una o más filas de la segunda tabla. Esta es la relación que existe entre los clientes y sus pedidos, entre las cabeceras de pedidos y sus líneas de detalles, entre Enrique VIII y sus seis esposas... Al definir el esquema de un base de datos, estas relaciones se tienen en cuenta, por ejemplo, para crear restricciones de integridad referencial entre tablas.

Delphi permite establecer un tipo de vínculo entre dos tablas, la relación *master/detail*, con el que podemos representar este tipo de dependencias. En el vínculo intervienen dos tablas, a las que denominaremos la tabla *maestra* y la tabla *dependiente*. La tabla maestra puede ser, en realidad, cualquier tipo de conjunto de datos. La tabla dependiente, en cambio, debe ser un *TTable*. Existe una técnica para hacer que una consulta sea controlada desde otro conjunto de datos, que será estudiada en el capítulo 26.

Cada vez que se cambia la fila activa en la tabla maestra, se restringe el conjunto de trabajo de la tabla dependiente a las filas relacionadas. Si la tabla maestra es la tabla de clientes y la tabla dependiente es la de pedidos, la última tabla 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

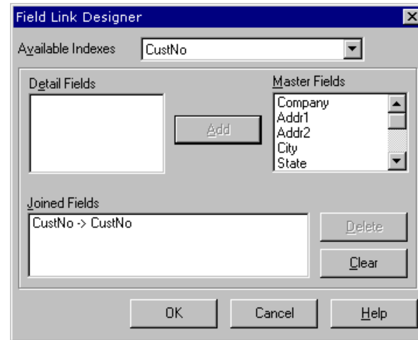
Clientes	
Código	**Nombre**
1221	Kauai Dive Shoppe
1231	Unisco
1351	Sight Diver

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 tablas solamente hay que hacer cambios en la tabla que va a funcionar como tabla dependiente. Las propiedades de la tabla dependiente que hay que modificar son las siguientes:

Propiedad	Propósito
MasterSource	Apunta a un *datasource* asociado a la tabla maestra
IndexName ó IndexFieldNames	Criterio de ordenación en la tabla dependiente
MasterFields	Los campos de la tabla maestra que forman la relación

Es necesario configurar una de las propiedades *IndexName* ó *IndexFieldNames*. Aunque estas propiedades se estudiarán en el capítulo sobre índices, nos basta ahora con saber que son modos alternativos de establecer un orden sobre una tabla. Este criterio de ordenación es el que aprovecha Delphi para restringir eficientemente el cursor sobre la tabla dependiente. En el ejemplo que mostramos antes, la tabla de pedidos debe estar ordenada por la columna *Cliente*.

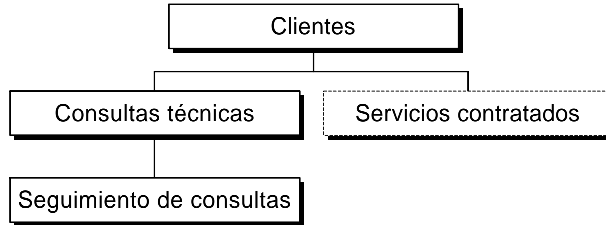


Sin embargo, no tenemos que modificar directamente estas propiedades en tiempo de diseño, pues el editor de la propiedad *MasterFields* se encarga automáticamente de ello. Este editor, conocido como el Editor de Enlaces (*Links Editor*) se ejecuta cuando realizamos una doble pulsación sobre la propiedad *MasterFields*, y su aspecto depende de si la tabla dependiente es una tabla local o SQL. Si la tabla está en formato Paradox o dBase, el diálogo tiene un combo, con la leyenda *AvailableIndexes* 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.

Pero si la tabla pertenece a una base de datos SQL, no aparece la lista de índices, y en el cuadro de lista de la izquierda aparecen todas las columnas de la tabla dependiente. Una tabla SQL no tiene, 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.

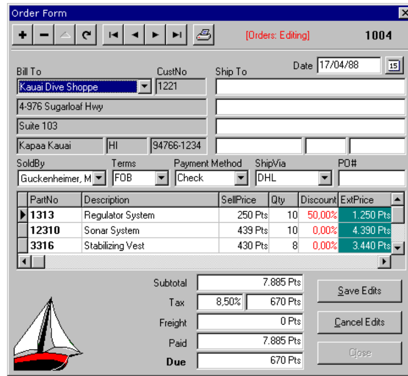
En cualquiera de los dos casos, las columnas de la lista de la derecha corresponden a la tabla maestra, y son las que se asignan realmente a la propiedad *MasterFields*. En el ejemplo anterior, *MasterFields* debe tener el valor *Código*.

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.

La relación *master/detail* no está limitada a representar relaciones uno/muchos, pues también puede utilizarse para la relación inversa. Podemos designar como tabla maestra la tabla de pedidos, y como tabla de detalles la tabla de clientes. En este ejemplo, 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 uno de los programas de ejemplo 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 correspondiente también a una tabla de detalles: las líneas correspondientes al pedido activo.

> Otra forma de representar relaciones uno/muchos es mediante las tablas anidadas de Oracle 8. Cuando estudiemos las extensiones de objetos de este producto, analizaremos el nuevo componente *TNestedTable*, que permite visualizar los datos de detalles desde Delphi 4. La nueva versión también permite tablas anidadas en conjuntos de datos clientes, que estudiaremos más adelante.

Navegación y relaciones master/detail

Supongamos que necesitamos conocer el total facturado por clientes que no viven en los Estados Unidos. Tenemos un formulario con un par de tablas, *tbClientes* y *tbPedidos*, en relación *master/detail*, y queremos aprovechar estos componentes para ejecutar esta operación. El país del cliente se almacena en la tabla de clientes, mientras que el total del pedido va en la tabla de pedidos. Y, muy importante para esta sección, las dos tablas están conectadas a sendas rejillas de datos. Estas serán las propiedades de la primera tabla:

	Propiedad	Valor
tbClientes	*DatabaseName*	*dbdemos*
	TableName	*customer.db*
	Active	*True*

A esta tabla se le asocia un componente *TDataSource*:

	Propiedad	Valor
dsClientes	*DataSet*	*tbClientes*

Ahora le toca el turno a la segunda tabla:

	Propiedad	Valor
tbPedidos	*DatabaseName*	*dbdemos*
	TableName	*orders.db*
	MasterSource	*dsClientes*
	IndexName	*CustNo*
	MasterFields	*CustNo*
	Active	*True*

La tabla tendrá su correspondiente *TDataSource*:

	Propiedad	Valor
dsPedidos	*DataSet*	*tbPedidos*

Añada, finalmente, un par de rejillas de datos (*TDBGrid*, en la página *Data Access*), y modifique sus propiedades *DataSource* para que apunten a los dos componentes correspondientes. Ponga entonces un botón en algún sitio del formulario para efectuar la suma de los pedidos. Y pruebe este algoritmo inicial, en respuesta al evento *OnClick* del botón:

```
// PRIMERA VARIANTE: ¡¡¡MUY INEFICIENTE!!!
procedure TForm1.Button1Click(Sender: TObject);
var
   Total: Currency;
   Tiempo: Cardinal;
begin
   Tiempo := GetTickCount;
   Total := 0;
   tbClientes.First;
   while not tbClientes.EOF do
   begin
      if tbClientes['COUNTRY'] <> 'US' then
      begin
         tbPedidos.First;
         while not tbPedidos.EOF do
         begin
            Total := Total + tbPedidos['ITEMSTOTAL'];
            tbPedidos.Next;
         end;
      end;
      tbClientes.Next;
   end;
   ShowMessage(Format('%m: %d', [Total, GetTickCount - Tiempo]));
end;
```

Bueno, he tenido que adelantarme un poco al orden de exposición del libro. 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 tbClientes['COUNTRY'] <> 'US' then
// …
Total := Total + tbPedidos['ITEMSTOTAL'];
```

Por el momento, debe saber que con esta técnica se obtiene el valor del campo como un tipo *Variant*, de modo que en dependencia del contexto en que se emplee el valor, Delphi realizará la conversión de tipos adecuada.

Habrá observado el comentario que encabeza el primer listado de esta sección. Si ha seguido el ejemplo y pulsado el botón, comprenderá por qué es ineficiente ese código. ¡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!!!
procedure TForm1.Button1Click(Sender: TObject);
var
   Total: Currency;
   Tiempo: Cardinal;
begin
   Tiempo := GetTickCount;
   Total := 0;
   tbClientes.DisableControls;       // ← NUEVO
   tbPedidos.DisableControls;        // ← NUEVO
   try
      tbClientes.First;
      while not tbClientes.EOF do
      begin
         if tbClientes['COUNTRY'] <> 'US' then
         begin
            tbPedidos.First;
            while not tbPedidos.EOF do
            begin
               Total := Total + tbPedidos['ITEMSTOTAL'];
               tbPedidos.Next;
            end;
         end;
         tbClientes.Next;
      end;
   finally
      tbPedidos.EnableControls;      // ← NUEVO
      tbClientes.EnableControls;     // ← NUEVO
   end;
   ShowMessage(Format('%m: %d', [Total, GetTickCount - Tiempo]));
end;
```

Ahora el algoritmo sí va rápido, ¡pero devuelve un resultado a todas luces incorrecto! Cuando llamamos a *DisableControls* estamos desconectando el mecanismo de notifica-

ción 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 la tabla de pedidos no modifica el conjunto de filas activas cada vez que se selecciona un cliente diferente.

¿Quiere una solución que funcione en cualquier versión de Delphi? Es muy sencilla: utilice dos componentes *TDataSource* acoplados a la tabla de clientes. Traiga un nuevo componente de este tipo, *DataSource1*, y cambie su propiedad *DataSet* al valor *tbClientes*. Entonces, haga que la propiedad *DataSource* de *DBGrid1* apunte a *DataSource1* en vez de a *dsClientes*. Por último, modifique el algoritmo de iteración del siguiente modo:

```
// TERCERA VARIANTE: ¡¡¡AL FIN BIEN!!!
procedure TForm1.Button1Click(Sender: TObject);
var
   Total: Currency;
   Tiempo: Cardinal;
begin
   Tiempo := GetTickCount;
   Total := 0;
   DataSource1.Enable := False;      // ← NUEVO
   tbPedidos.DisableControls;
   try
      tbClientes.First;
      while not tbClientes.EOF do
      begin
         if tbClientes['COUNTRY'] <> 'US' then
         begin
            tbPedidos.First;
            while not tbPedidos.EOF do
            begin
               Total := Total + tbPedidos['ITEMSTOTAL'];
               tbPedidos.Next;
            end;
         end;
         tbClientes.Next;
      end;
   finally
      tbPedidos.EnableControls;
      DataSource1.Enable := True;    // ← NUEVO
   end;
   ShowMessage(Format('%m: %d', [Total, GetTickCount - Tiempo]));
end;
```

Ahora, el flujo de notificaciones se corta al nivel de una fuente de datos particular, no al nivel general de la tabla. De esta forma, *tbClientes* sigue enviando notificaciones a sus dos fuentes de datos asociadas. La fuente de datos *dsClientes* propaga estas notificaciones a los objetos que hacen referencia a ella: en este caso, la tabla dependiente *dsPedidos*. Pero *DataSource1* es inhabilitada temporalmente para que la rejilla asociada no reciba notificaciones de cambio.

¿Quiere una solución que funciona solamente a partir de Delphi 4? En esta versión se introduce la propiedad *BlockReadSize*. Cuando el valor de la misma es mayor que cero, el conjunto de datos entra en un estado especial: la propiedad *State*, que veremos en la siguiente sección, toma el valor *dsBlockRead*. En este 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*.
- Al parecer, la modificación de esta propiedad en una tabla de detalles no funciona correctamente en la versión 4.0 de Delphi.

Teniendo en cuenta estas advertencias, nuestro algoritmo pudiera escribirse de esta forma alternativa en Delphi 4:

```
// CUARTA VARIANTE: ¡¡¡SOLO DELPHI 4!!!
procedure TForm1.Button1Click(Sender: TObject);
var
   Total: Currency;
   Tiempo: Cardinal;
begin
   Tiempo := GetTickCount;
   Total := 0;
   // Se llama a First antes de modificar BlockReadSize
   tbClientes.First;
   tbClientes.BlockReadSize := 10;   // ← NUEVO
   tbPedidos.DisableControls;        // Se mantiene
   try
      while not tbClientes.EOF do
      begin
         if tbClientes['COUNTRY'] <> 'US' then
         begin
            tbPedidos.First;
            while not tbPedidos.EOF do
            begin
               Total := Total + tbPedidos['ITEMSTOTAL'];
               tbPedidos.Next;
            end;
         end;
         tbClientes.Next;
      end;
   finally
      tbPedidos.EnableControls;
      tbClientes.BlockReadSize := 0; // ← NUEVO
   end;
   ShowMessage(Format('%m: %d', [Total, GetTickCount - Tiempo]));
end;
```

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.

El estado de un conjunto de datos

Una de las propiedades más importantes de los conjuntos de datos es *State*, cuya declaración es la siguiente:

```
type
   TDataSetState = (dsInactive, dsBrowse, dsEdit, dsInsert,
      dsSetKey, dsCalcFields, dsUpdateNew, dsUpdateOld, dsFilter);

   TDataSet = class(TComponent)
      // …
      property State: TDataSetState;
      // …
   end;
```

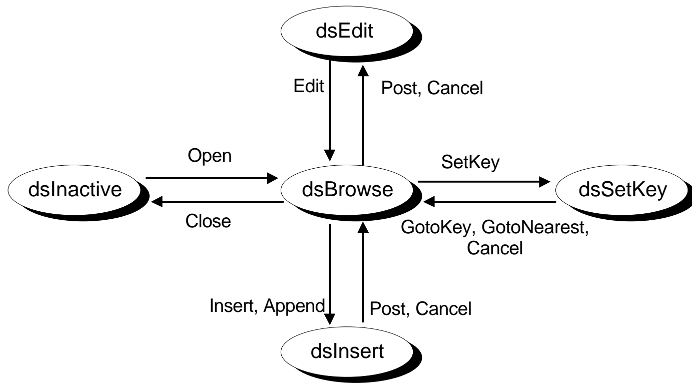
En Delphi 3 la definición de este tipo cambia un poco, aunque el cambio no debe afectar a ningún programa existente, pues solamente se han renombrado ciertos estados internos:

```
type
   TDataSetState = (dsInactive, dsBrowse, dsEdit, dsInsert,
      dsSetKey, dsCalcFields, dsFilter, dsNewValue, dsOldValue,
      dsCurValue);
```

El estado de una tabla determina qué operaciones se pueden realizar sobre la misma. Por ejemplo, en el estado de exploración, *dsBrowse*, no se pueden realizar asignaciones a campos. Algunas operaciones cambian su semántica de acuerdo al estado en que se encuentra la tabla. El método *Post*, por ejemplo, graba una nueva fila si el estado es *dsInsert* en el momento de su aplicación, pero modifica la fila activa si el estado es *dsEdit*.

La propiedad *State* es una propiedad de sólo lectura, por lo que no podemos cambiar de estado simplemente asignando un valor a ésta. Incluso hay estados a los cuales el programador no puede llegar explícitamente. Tal es el caso del estado *dsCalcFields*, al cual se pasa automáticamente cuando existen campos calculados en la tabla; estos campos se estudiarán en el capítulo siguiente. Las transiciones de estado que puede realizar el programador se logran mediante llamadas a métodos. El siguiente diagrama, todo un clásico de los libros de Delphi, muestra los diferentes estados de un

conjunto de datos a los que puede pasar explícitamente el programador, y las transiciones entre los mismos:



En el diagrama no se han representado los estados internos e inaccesibles. Los estados *dsUpdateNew* y *dsUpdateOld* (*dsOldValue*, *dsNewValue* y *dsCurValue*, en Delphi 3; Delphi 4 añade *dsBlockRead*) son estados utilizados internamente por Delphi, y el programador nunca encontrará que una rutina programada por él se está ejecutando con una tabla en uno de estos estados. En cambio, aunque el programador nunca coloca una tabla de forma explícita en los estados *dsCalcFields* y *dsFilter*, aprovecha estos estados durante la respuestas a un par de eventos, *OnCalcFields* y *OnFilterRecord*. El primero de estos eventos se utiliza para asignar valores a campos calculados; el segundo evento permite trabajar con un subconjunto de filas de una tabla, y lo estudiaremos en el capítulo sobre métodos de búsqueda.

La comprensión de los distintos estados de un conjunto de datos, los métodos y los eventos de transición son fundamentales para poder realizar actualizaciones en bases de datos. Más adelante, volveremos obligatoriamente sobre este tema.

Capítulo

16

Acceso a campos

Los componentes de acceso a campos son parte fundamental de la estructura de la VCL de Delphi. 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 a nosotros se nos olvida hacerlo. En este capítulo estudiaremos las clases de campos y sus propiedades, concentrándonos en los tipos de campos "simples", y en el uso del Diccionario de Datos para acelerar la configuración de campos en tiempo de diseño. En capítulos posteriores tendremos oportunidad de estudiar los campos BLOB y los correspondientes a las nuevas extensiones orientadas a objeto de Oracle 8.

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 a la tabla o consulta a la 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 tabla en un formulario, asociarle una fuente de datos y una rejilla, y echar a andar la aplicación resultante. ¿Para qué queremos campos entonces? Bueno, 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, Delphi detecta que el usuario no ha definido campos en tiempo de diseño, 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. Esta creación 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 *dfm*, 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. El formato del Editor es diferente en Delphi 1 respecto a las versiones posteriores. En Delphi 1, el Editor tiene botones incorporados para realizar los comandos más frecuentes. En Delphi 2 se decidió que de esta forma la ventana ocupaba demasiado espacio en pantalla, por lo cual se redujo al mínimo y se movieron los comandos al menú local del propio Editor de Campos. De este modo, para realizar casi cualquier acción en el Editor de Campos actual hay que pulsar el botón derecho del ratón y seleccionar el comando de menú adecuado. En común a todas las versiones, tenemos 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.

Añadir componentes de campos es muy fácil. Basta con pulsar, en Delphi 1, el botón *Add*, o el comando de menú *Add fields* del menú local, en caso contrario. En todas las versiones se presenta 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 se presenta por omisión con todos los campos seleccionados. 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 Delphi 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 estos campos que dejamos sin crear durante el diseño *no existen* en lo que concierne a Delphi.

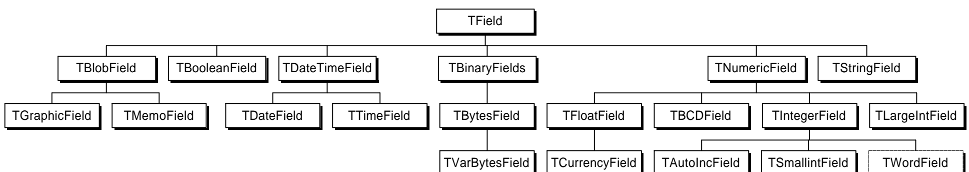
Add fields... Ctrl+A
New field... Ctrl+N
Add all fields Ctrl+F
Cut Ctrl+X
Copy Ctrl+C
Paste Ctrl+V
Delete Del
Select all Ctrl+L
Retrieve attributes Ctrl+R
Save attributes Ctrl+S
Save attributes as... Ctrl+E
Associate attributes... Ctrl+O
Unassociate attributes Ctrl+U

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, basta seleccionarlo en el Editor de Campos y pulsar la tecla SUPR.

> El comando *Add all fields*, del menú local del Editor de Campos, es una novedad de Delphi 4 para acelerar la configuración de campos.

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 esta jerarquía:

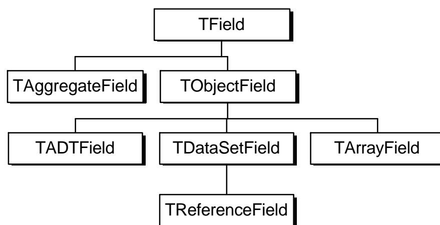


De todos estos tipos, *TField*, *TNumericField* y *TBinaryField* (nuevo en Delphi 4) nunca se utilizan directamente para crear instancias de objetos de campos; su papel es servir de ancestro a campos de tipo especializado. La correspondencia entre los tipos de las variables de campos y los tipos de las columnas es la siguiente:

Tipo de campo	dBase	Paradox	InterBase
TStringField	char	alpha	char, varchar
TIntegerField		longint	int, long
TAutoIncField		autoinc	
TWordField			
TSmallintField	number	shortint	short
TBCDField		bcd	
TFloatField	float, number	number	float, double
TCurrencyField		money	
TBooleanField	logical	logical	
TDateField	date	date	
TTimeField		time	
TDateTimeField		timestamp	date
TBlobField	ole, binary	fmtmemo, ole, binary	blob
TGraphicField		graphic	
TMemoField	memo	memo	text blob
TBytesField		bytes	
TVarBytesField			

Algunos tipos de campos se asocian con las clases de campos de Delphi en dependencia de su tamaño y precisión. Tal es el caso de los tipos **number** de dBase, y de **decimal** y **numeric** de InterBase.

A esta jerarquía de clases, Delphi 4 añade un par de ramas:



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, la versión actual de Delphi y del BDE no permiten tratarlos como tales directamente.

Nombre del campo y etiqueta de visualización

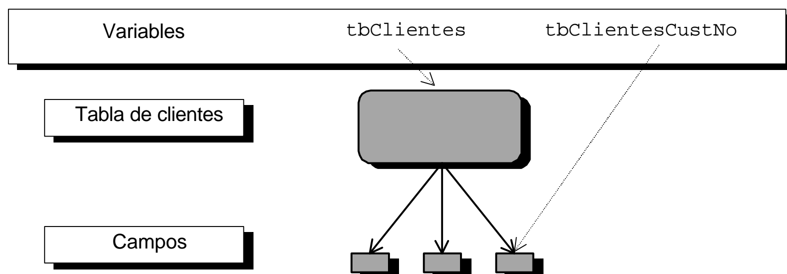
Existen tres propiedades de los campos que muchas veces son confundidas entre sí por el programador. Son las propiedades *Name*, *FieldName* y *DisplayLabel*. La primera es, como sucede con casi todos los componentes, el nombre de la variable de campo, o sea, del puntero al objeto. *FieldName* es el 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, es decir, *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 *dfm* crece desmesuradamente. Tomemos por ejemplo una aplicación pequeña que trabaje con diez tablas, y supongamos que cada tabla tiene diez campos; estas 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 *dfm* 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[I: Integer]: TField;
```

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. A esto volveremos a referirnos.

Si a la función *FieldByName* le pasamos un nombre inexistente de campo, se produce 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, Delphi construye 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('%d-%s',
   [tbClientesCodigo.Value,       // Un valor entero
    tbClientesNombre.Value]));    // Una cadena de caracteres
```

Si la referencia al campo es del 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(tbClientes.FieldByName('Codigo').AsInteger)
   + '-' + tbClientes.FieldByName('Nombre').AsString);
```

Las propiedades mencionadas intentan siempre hacer la conversión del valor almacenado realmente al tipo especificado. Por ejemplo, en el caso anterior hubiéramos podido utilizar también la propiedad *AsString* aplicada al campo *Codigo*.

Ahora bien, existe un camino alternativo para manipular los datos de un campo: la propiedad *FieldValues* de *TDataSet*, una novedad de Delphi 2. La declaración de esta propiedad es la siguiente:

```
property FieldValues[const FieldName: string]: Variant; default;
```

Como se puede ver, esta es la propiedad vectorial por omisión de los conjuntos de datos, por lo cual pueden aplicarse los corchetes directamente a una variable de tabla, como si ésta fuera un vector. Además, como la propiedad devuelve valores variantes, no es necesario preocuparse demasiado por el tipo del campo, pues la conversión transcurre automáticamente:

```
ShowMessage(tbClientes['Codigo'] + '-' + tbClientes['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 := tbClientes['Codigo;Nombre'];
   ShowMessage(V[0] + '-' + V[1]);
end;
```

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 28 sobre actualizaciones se tratan estos temas con mayor detalle; un poco más adelante, 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; las 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**
dd-mm-yy	04-07-64
dddd, d "de" mmmm "de" yyyy	sábado, 26 de enero de 1974
hh:mm	14:05
h:mm am/pm	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.

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, es llamado 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 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');
var
   I: Integer;
begin
   if Sender.AsFloat > 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;
```

Si a algún usuario se le ocurriera 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
         AnsiUpperBuff(@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.

Validación a nivel de campos

Los componentes de acceso a campos ofrecen otros dos eventos: *OnValidate* y *OnChange*. Este último se dispara cuando cambia el contenido del campo, y puede utilizarse para coordinar actualizaciones entre columnas. El más usado, sin embargo, es *OnValidate*, que se emplea para verificar que los valores asignados a un campo sean correctos.

¿Cuándo se dispara este evento? La respuesta es importante: no debe importarnos cuándo. En realidad, el evento se dispara cuando se va a transferir la información del campo al *buffer* del registro activo. Pero esta operación se realiza en unas cuantas situaciones diferentes, y es difícil rastrear todos estos casos. No hay que preocuparse de este asunto, pues la VCL de Delphi se encarga de disparar el evento en el momento adecuado. Por el contrario, debemos estar preparados para dar la respuesta adecuada cuando el hecho suceda. Cuando el sabio señala a la Luna, el tonto solamente ve el dedo.

Sin embargo, es bueno aclarar que *OnValidate* solamente se produce 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. Pero si intentamos crear nuevos registros con valores incorrectos, o modificar alguno de los registros incorrectos, se señalará el error.

El siguiente método, programado como respuesta a un evento *OnValidate*, verifica que un nombre propio debe contener solamente caracteres alfabéticos (no somos robots):

```
procedure TmodDatos.VerificarNombrePropio(Sender: TField);
var
   S: string;
   I: Integer;
begin
   S := Sender.AsString;
   for I := Length(S) downto 0 do
      if (S[I] <> ' ') and not IsCharAlpha(S[I]) then
         DatabaseError('Carácter no permitido en un 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 inválido en un 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 Delphi, no por el BDE.

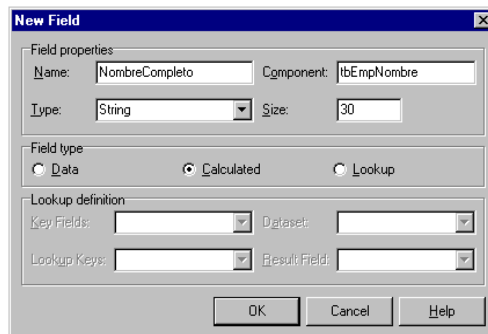
Propiedades de validación

La intercepción del evento *OnValidate* es la forma más general de verificar el contenido de un campo. Pero la mayor parte de las validaciones pueden efectuarse modificando valores de propiedades. Por ejemplo, para comprobar que a una columna no se le asigne un valor nulo se emplea la propiedad *Required* del campo correspondiente. Si el campo es numérico y queremos limitar el rango de valores aceptables, podemos utilizar las propiedades *MinValue* y *MaxValue*. En el caso de los campos alfanuméricos, el formato de la cadena se puede limitar mediante la propiedad *EditMask*, que hemos mencionado anteriormente.

> He dejado para más adelante, en este mismo capítulo, el uso de las *restricciones* (*constraints*) y su configuración mediante el Diccionario de Datos.

Campos calculados

Una potente característica de Delphi es la de permitir la definición de *campos calculados*, que no corresponden a campos definidos "físicamente" sobre la tabla o consulta base, 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. Esto, si no estamos en Delphi 1; en esa versión, había que pulsar el botón *Define* contenido en esa ventana. El cuadro de diálogo que aparece entonces es diferente en formato para estas versiones, pero básicamente nos sirve para suministrar la misma información: 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. En ambos casos, también hay que aclarar que el campo es *Calculated*.

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; esto último, sin embargo, no lo controla Delphi. 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.db* que podemos encontrar en el alias *dbdemos* de Delphi, podemos definir un par de campos calculados con esta estructura:

Campo	Tipo	Tamaño
NombreCompleto	*String*	30
Antigüedad	*Integer*	

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 *tbEmpleados*:

```
procedure TmodDatos.tbEmpleadosCalcFields(Sender: TDataSet);
begin
   tbEmpleados['NombreCompleto'] := tbEmpleados['LastName']
      + ', ' + tbEmpleados['FirstName'];
   if not tbEmpleadosHireData.IsNull then
      tbEmpleados['Antiguedad'] := Date - tbEmpleados['HireDate'];
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 el método *IsNull*. Por el contrario, he asumido que el nombre y los apellidos no pueden ser nulos, por lo cual no me he tomado la molestia de comprobar este detalle.

Campos de búsqueda

Es muy frecuente encontrar tablas relacionadas entre sí mediante una relación uno/muchos; ya hemos visto que esta relación se puede expresar en Delphi mediante una relación *master/detail* entre tablas. A veces es conveniente, sin embargo, considerar la inversa de esta relación. Por ejemplo, un cliente "posee" un conjunto de pedidos ¿Debemos por esto trabajar los pedidos como detalles de un cliente? Podemos hacerlo de esta manera, pero lo usual es que las altas de pedidos se realicen sobre una tabla sin restricciones. Desde el punto de vista de la tabla de pedidos, su relación con la tabla de clientes es una *relación de referencia*.

Teniendo el código de cliente que se almacena en la tabla de pedidos, es deseable poder obtener el nombre de la empresa representada por ese código. Se pueden crear campos calculados que realicen manualmente la búsqueda por medio de índices en la tabla a la cual se hace referencia. En el capítulo sobre índices veremos cómo se puede implementar esto en Delphi 1, y en el capítulo sobre técnicas de búsqueda veremos cómo hacerlo a partir de Delphi 2. Sin embargo, pocas veces es necesario

llegar a estos extremos, pues desde la versión 2 de Delphi disponemos de los denominados *campos de búsqueda* (o, en inglés, *lookup fields*), para los cuales el sistema ejecuta automáticamente el algoritmo de traducción de referencias.

Los campos de búsqueda se crean por medio del comando de menú *New field* del menú local del Editor de Campos; se trata del mismo cuadro de diálogo que crea campos calculados. Pongamos por caso que queremos crear un campo que nos dé el nombre del cliente asociado a un pedido. Utilizaremos ahora las tablas de pedidos (*tbPedidos*, asociada a *orders.db*) y de clientes (*tbClientes*, asociada a *customer.db*). 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
Cliente	*String*	30

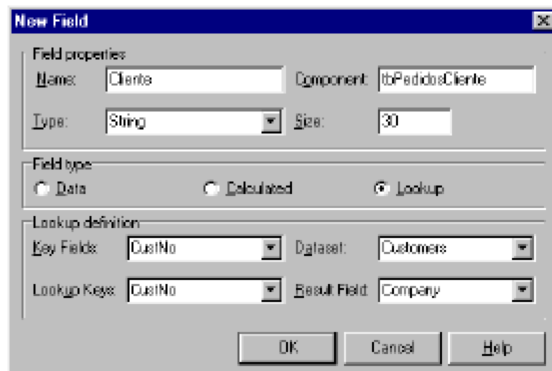
Después, tenemos que indicar el tipo de campo como *Lookup*; de este modo, se activan los controles de la parte inferior del diálogo. Estos son los valores que ha que suministrar, y su significado:

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 La tabla en la cual hay que buscar la referencia. En este caso, *tbClientes*.

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. Teclee *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
Lookup	Siempre igual a *True* para estos campos
KeyFields	Campos de la tabla en los que se basa la búsqueda (*Key fields*)
LookupDataset	Tabla de búsqueda (*Dataset*).
LookupKeyFields	Campos de la tabla de búsqueda que deben corresponder al valor de los *KeyFields* (*Lookup keys*)
LookupResultField	Campo de la tabla de búsqueda cuyo valor se toma (*Result field*)

La caché de búsqueda

A partir de Delphi 3 se han añadido propiedades y métodos para hacer más eficiente el uso de campos de búsqueda. Consideremos, por un momento, un campo que debe almacenar formas de pago. Casi siempre estas formas de pago están limitadas a un conjunto de valores determinados. Si utilizamos un conjunto especificado directamente dentro del código de la aplicación se afecta la extensibilidad de la misma, pues para añadir un nuevo valor hay que recompilar la aplicación. Muchos programadores optan por colocar los valores en una tabla y utilizar un campo de búsqueda para representar las formas de pago; en la columna correspondiente se almacena ahora el código asociado a la forma de pago. Desgraciadamente, este estilo de programación era algo ineficiente en Delphi 2, sobre todo en entornos cliente/servidor.

Se puede y debe utilizar memoria caché para los campos de búsqueda si se dan las siguientes condiciones:

- La tabla de referencia contiene relativamente pocos valores.
- Es poco probable que cambie la tabla de referencia.

Para activar la caché de un campo de búsqueda se utiliza la propiedad *LookupCache*, de tipo *Boolean*: asignándole *True*, los valores de referencia se almacenan en la propiedad *LookupList*, de la cual Delphi los extrae una vez inicializada automáticamente. Si ocurre algún cambio en la tabla de referencia mientras se ejecuta la aplicación, basta con llamar al método *RefreshLookupList*, sobre el componente de campo, para releer los valores de la memoria caché.

¿Son peligrosos los campos de búsqueda en la programación cliente/servidor? No, en general. La alternativa a ellos es el uso de consultas basadas en encuentros (*joins*), que tienen el inconveniente de no ser actualizables intrínsecamente y el de ser verdaderamente peligrosas para la navegación. El problema con los campos de búsqueda surge cuando se visualizan en un control *TDBLookupComboBox*, que permite la búsqueda incremental insensible a mayúsculas y minúsculas. La búsqueda se realiza con el método *Locate*, que desempeña atrozmente este cometido en particular. Para más información, lea el capítulo 27, sobre la comunicación cliente/servidor.

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.db*, para la cual utilizaremos un componente de tabla *tbDetalles*. Esta tabla 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 la tabla *parts.db* como tabla de referencia para los artículos, puede crearse un campo calculado *Precio* que devuelva el precio de venta del artículo en cuestión, extrayéndolo de la columna *ListPrice* de *parts.db*. 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.tbDetallesCalcFields(Sender: TDataSet);
begin
   tbDetalles['SubTotal'] :=
      tbDetalles['Qty'] *                    // Campo base
      tbDetalles['Precio'] *                 // Campo de búsqueda
      (1 - tbDetalles['Discount']/100);      // Campo base
end;
```

Lo importante es que, cuando se ejecuta este método, ya Delphi ha evaluado los campos de búsqueda, y tenemos un valor utilizable en el campo *Precio*.

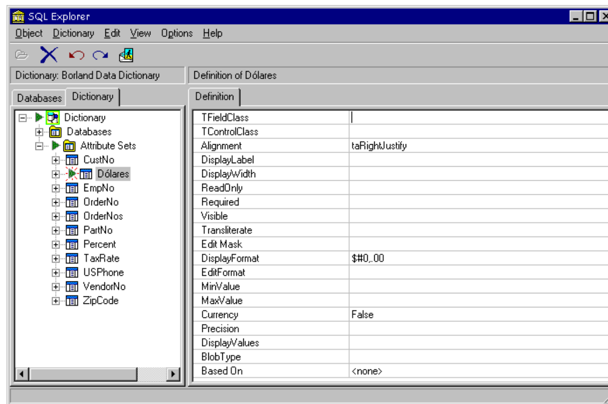
Otra información que es importante conocer acerca del orden de evaluación, es que Delphi evalúa los campos calculados antes de haber fijado el rango de filas de las tablas de detalles asociadas. Si la tabla *tbDetalles* está asociada a una tabla con cabeceras de pedidos, *tbPedidos*, por una relación *master/detail*, no es posible definir un campo calculado en la tabla de pedidos que sume los subtotales de cada línea asociada, extrayendo los valores de *tbDetalles*. Es necesario utilizar una tercera tabla auxi-

liar, que se refiera a la tabla de líneas de detalles, para buscar en esta última las líneas necesarias y realizar la suma.

El Diccionario de Datos

El Diccionario de Datos es una ayuda para el diseño que se administra desde la herramienta *Database Explorer*. El Diccionario nos ayuda a:

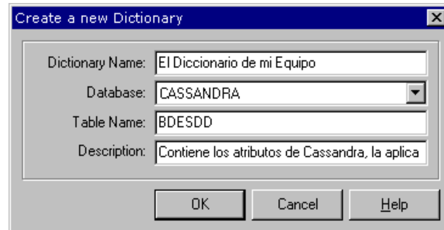
- Definir conjuntos de propiedades o *conjuntos de atributos* (*attribute sets*), que pueden después asociarse manual o automáticamente a los componentes de acceso a campos que se crean en Delphi. Por ejemplo, si nuestra aplicación trabaja con varios campos de porcentajes, puede sernos útil definir que los campos de este tipo se muestren con la propiedad *DisplayFormat* igual a *'%#0'*, y que sus valores oscilen entre 0 y 100.
- Propagar a las aplicaciones clientes restricciones establecidas en el servidor, ya sea a nivel de campos o de tablas, valores por omisión, definiciones de dominios, etcétera.
- Organizar de forma centralizada los criterios de formato y validación entre varios programadores de un mismo equipo, y de una aplicación a otra.



Delphi, al instalarse, crea un Diccionario de Datos por omisión, que se almacena como una tabla Paradox, cuyo nombre por omisión es *bdesdd* y que reside en el alias predefinido *DefaultDD*. Recuerde que esta herramienta es una utilidad que funciona en tiempo de diseño, de modo que la tabla anterior no tiene (no *debe*, más bien) que estar presente junto a nuestro producto final.

Ahora bien, podemos trabajar con otro diccionario, que puede estar almacenado en cualquier otro formato de los reconocidos por Delphi. Este diccionario puede incluso almacenarse en un servidor SQL y ser compartido por todos los miembros de

un equipo de programación. Mediante el comando de menú *Dictionary|New* uno de los programadores del equipo, digamos que usted, crea una nueva tabla para un nuevo diccionario, indicando el alias donde residirá, el nombre de la tabla y una descripción para el diccionario:



Después, mediante el comando de menú *Dictionary|Register* cualquier miembro del equipo, digamos que yo, puede registrar ese diccionario desde *otra* máquina, para utilizarlo con *su* Delphi. También es útil el comando *Dictionary|Select*, para activar alguno de los diccionarios registrados en determinado ordenador.

Conjuntos de atributos

Son dos los objetos almacenados en el Diccionario de Datos: los conjuntos de atributos e información sobre bases de datos completas. Un conjunto de atributos indica valores para propiedades comunes de campos. Existen dos formas de crear esta definición: guardando los atributos asociados a un campo determinado, o introduciéndolos directamente en el Diccionario de Datos. Para salvar las modificaciones realizadas a un campo desde Delphi, hay que seleccionarlo en el Editor de Campos, pulsar el botón derecho del ratón y ejecutar el comando *Save attributes as*, que nos pedirá un nombre para el conjunto de atributos.

La otra vía es crear directamente el conjunto de atributos en el propio Diccionario. Digamos que nuestra aplicación debe trabajar con precios en dólares, mientras que nuestra moneda local es diferente. Nos situamos entonces sobre el nodo *Attributes set* y ejecutamos *Object|New*. Renombramos el nuevo nodo como *Dollar*, y modificamos las siguientes propiedades, en el panel de la derecha:

Propiedad	Valor
Currency	*False*
DisplayFormat	*$#0,.00*

¿Qué se puede hacer con este conjunto de atributos, una vez creado?. Asociarlo a campos, por supuesto. Nuestra hipotética aplicación maneja una tabla *Articulos* con un campo *PrecioDolares*, que ha sido definido con el tipo *money* de Paradox o de MS

SQL Server. Delphi, por omisión, trae un campo de tipo *TCurrencyField*, cuya propiedad *Currency* aparece como *True*. El resultado: nuestros dólares se transforman mágicamente en pesetas (y pierden todo su valor). Pero seleccionamos el menú local del campo, y ejecutamos el comando *Associate attributes*, seleccionando el conjunto de atributos *Dollar* definido hace poco. Delphi lee los valores de *Currency* y *DisplayFormat* desde el Diccionario de Datos y los copia en las propiedades del campo deseado. Y todo vuelve a la normalidad.

Dos de los atributos más importantes que se pueden definir en el Diccionario son *TFieldClass* y *TControlClass*. Mediante el primero podemos establecer explícitamente qué tipo de objeto de acceso queremos asociar a determinado campo; esto es útil sobre todo con campos de tipo BLOB. *TControlClass*, por su parte, determina qué tipo de control debe crearse cuando se arrastra y suelta un componente de campo sobre un formulario en tiempo de diseño.

> Si un conjunto de atributos se modifica después de haber sido asociado a un campo, los cambios no se propagarán automáticamente a las propiedades de dicho campo. Habrá entonces que ejecutar el comando del menú local *Retrieve attributes*.

Importando bases de datos

Pero si nos limitamos a las técnicas descritas en la sección anterior, tendremos que configurar atributos campo por campo. Una alternativa consiste en *importar* el esquema de la base de datos dentro del Diccionario, mediante el comando de menú *Dictionary|Import from database*. Aparece un cuadro de diálogo para que seleccionemos uno de los alias disponibles; nos debemos armar de paciencia, porque la operación puede tardar un poco.

> Muchas aplicaciones trabajan con un alias de sesión, definido mediante un componente *TDatabase* que se sitúa dentro de un módulo de datos, en vez de utilizar alias persistentes. Si ejecutamos Database Explorer como una aplicación independiente no podremos importar esa base de datos, al no estar disponible el alias en cuestión. La solución es invocar al Database Explorer desde Delphi, con la aplicación cargada y la base de datos conectada. Todos los alias de sesión activos en la aplicación podrán utilizarse entonces desde esta utilidad.

Cuando ha finalizado la importación, aparece un nuevo nodo para nuestra base de datos bajo el nodo *Databases*. Este nodo contiene todas las tablas y los campos existentes en la base de datos, y a cada campo se le asocia automáticamente un conjunto de atributos. Por omisión, el Diccionario crea un conjunto de atributos por cada

campo que tenga propiedades dignas de mención: una restricción (espera a la próxima sección), un valor por omisión, etc. Esto es demasiado, por supuesto. Después de la importación, debemos sacar factor común de los conjuntos de atributos similares y asociarlos adecuadamente a los campos. La labor se facilita en InterBase si hemos creado *dominios*, como explicaremos en el capítulo 21. Supongamos que definimos el dominio *Dollar* en la base de datos mediante la siguiente instrucción:

```
create domain Dollar as
       numeric(15, 2) default 0 not null;
```

A partir de esta definición podremos definir columnas de tablas cuyo tipo de datos sea *Dollar*. Entonces el Diccionario de Datos, al importar la base de datos, creará automáticamente un conjunto de atributos denominado *Dollar*, y asociará correctamente los campos que pertenecen a ese conjunto.

¿Para qué todo este trabajo? Ahora, cuando traigamos una tabla a la aplicación y utilicemos el comando *Add fields* para crear objetos persistentes de campos, Delphi podrá asignar de forma automática los conjuntos de atributos a estos campos.

Evaluando restricciones en el cliente

Los sistemas de bases de datos cliente/servidor, y algunas bases de datos de escritorio, permiten definir restricciones en las tablas que, normalmente, son verificadas por el propio servidor de datos. Por ejemplo, que el nombre del cliente no puede estar vacío, que la fecha de venta debe ser igual o anterior a la fecha de envío de un pedido, que un descuento debe ser mayor que cero, pero menor que cien...

Un día de verano, un usuario de nuestra aplicación (¡ah, los usuarios!) se enfrasca en rellenar un pedido, y vende un raro disco de Hendrix con un descuento del 125%. Al enviar los datos al servidor, éste detecta el error y notifica a la aplicación acerca del mismo. Un registro pasó a través de la red, un error nos fue devuelto; al parecer, poca cosa. Pero multiplique este tráfico por cincuenta puestos de venta, y lo poco se convierte en mucho. Además, ¿por qué hacer esperar tanto al usuario para preguntarle si es tonto, o si lo ha hecho a posta? Este tipo de validación sencilla puede ser ejecutada perfectamente por nuestra aplicación, pues solamente afecta a columnas del registro activo. Otra cosa muy diferente sería, claro está, intentar verificar por duplicado en el cliente una restricción de unicidad.

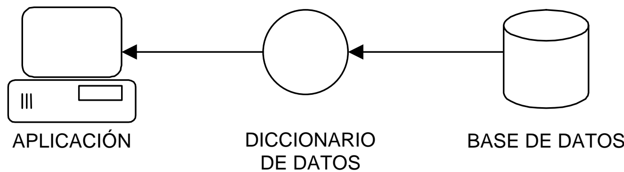
En Delphi 3 se introdujeron cuatro nuevas propiedades, de tipo **string**, para los componentes de acceso a campos:

Propiedad	Significado
DefaultExpression	Valor por omisión del campo
CustomConstraint	Restricciones importadas desde el servidor
ImportedConstraint	Restricciones adicionales impuestas en el cliente
ConstraintErrorMessage	Mensaje de error cuando se violan restricciones

DefaultExpression es una expresión SQL que sirve para inicializar un campo durante las inserciones. No puede contener nombres de campos. Antes de Delphi 3, la inicialización de los valores de los campos debía realizarse programando una respuesta al evento *OnNewRecord* del conjunto de datos al que pertenecía. Si un campo debe inicializarse con un valor constante, es más cómodo utilizar *DefaultExpression*.

> Hay un pequeño *bug* en Delphi: cuando se mezclan inicializaciones con el evento *OnNewRecord* y con las propiedades *DefaultExpression*, se producen comportamientos anómalos. Evite las mezclas, que no son buenas para la salud.

Podemos asignar directamente un valor constante en *DefaultExpression* ... pero si hemos asociado un conjunto de atributos al campo, Delphi puede leer automáticamente el valor por omisión asociado y asignarlo. Este conjunto de atributos puede haber sido configurado de forma manual, pero también puede haberse creado al importar la base de datos dentro del Diccionario. En este último caso, el Diccionario de Datos ha actuado como eslabón intermedio en la propagación de reglas de empresa desde el servidor al cliente:



Lo mismo se aplica a la propiedad *ImportedConstraint*. Esta propiedad recibe su valor desde el Diccionario de Datos, y debe contener una expresión SQL evaluable en SQL Local; léase, en el dialecto de Paradox y dBase. ¿Por qué permitir que esta propiedad pueda modificarse en tiempo de diseño? Precisamente, porque la expresión importada puede ser incorrecta en el dialecto local. En ese caso, podemos eliminar toda o parte de la expresión. Normalmente, el Diccionario de Datos extrae las restricciones para los conjuntos de atributos de las cláusulas **check** de SQL definidas a nivel de columna.

Si, por el contrario, lo que se desea es añadir nuevas restricciones, debemos asignarlas en la propiedad *CustomConstraint*. Se puede utilizar cualquier expresión lógica de SQL Local, y para representar el valor actual del campo hay que utilizar un identificador

cualquiera que no sea utilizado por SQL. Por ejemplo, esta puede ser una expresión aplicada sobre un campo de tipo cadena de caracteres:

```
x <> '' and x not like '% %'
```

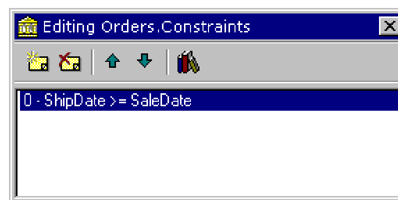
Esta expresión verifica que el campo no esté vacío y que no contenga espacios en blanco.

Cuando se viola cualquiera de las restricciones anteriores, *ImportedConstraint* ó *CustomConstraint*, por omisión se muestra una excepción con el mensaje *"Record or field constraint failed"*, más la expresión que ha fallado en la segunda línea del mensaje. Si queremos mostrar un mensaje personalizado, debemos asignarlo a la propiedad *ConstraintErrorMessage*.

Sin embargo, no todas las restricciones **check** se definen a nivel de columna, sino que algunas se crean a nivel de tablas, casi siempre cuando involucran a dos o más campos a la vez. Por ejemplo:

```
create table Pedidos (
        /* Restricción a nivel de columna */
        FormaPago varchar(10)
                check (FormaPago in ('EFECTIVO', 'TARJETA')),
        /* Restricción a nivel de tabla */
        FechaVenta date not null,
        FechaEnvio date,
        check (FechaVenta <= FechaEnvio),
        /* … */
);
```

Las restricciones a nivel de tabla se propagan a una propiedad de *TTable* denominada *Constraints*, que contiene tanto las restricciones importadas como alguna restricción personalizada añadida por el programador.



La imagen anterior muestra el editor de la propiedad *Constraints* en Delphi 4. El botón de la derecha permite leer las restricciones encontradas durante la importación de la base de datos al Diccionario; esta vez, Delphi no lo hace por sí mismo. Mediante el botón de la derecha se añaden restricciones personalizadas a la tabla. Cada restricción a nivel de tabla, venga de donde venga, tiene su propia propiedad *ConstraintErrorMessage*.

Información sobre campos

Delphi hace distinción entre la información sobre los campos físicos de una tabla y los componentes de acceso a los mismos. La información sobre campos se encuentra en la propiedad *FieldDefs*, y puede utilizarse, además, para la creación dinámica de tablas. Esta propiedad pertenece a la clase *TFieldDefs*, que es básicamente una lista de objetos de tipo *TFieldDef*. Las propiedades principales de esta clase son *Count*, que es la cantidad de campos, e *Items*, que es una propiedad vectorial para obtener cada definición individual. En *TFieldDefs* tenemos el método *Update*, que lee la lista de definiciones de campos del conjunto de datos, incluso si éste se encuentra cerrado:

```
with TTable.Create(nil) do
try
   DatabaseName := 'DBDEMOS';
   TableName := 'employee.db';
   FieldDefs.Update;       // Lee las definiciones de campos
   // …
finally
   Free;
end;
```

Una vez que tenemos la lista de definiciones en memoria, podemos acceder a cada definición por medio de la propiedad *Items*. Estas son las propiedades principales de la clase *TFieldDef*, a la cual pertenecen los elementos individuales de la colección:

Propiedad	Significado
Name	El nombre del campo
DataType	El tipo del campo
Size	Tamaño del campo, si es aplicable
Required	¿Admite valores nulos?
FieldClass	Referencia a la clase correspondiente derivada de *TField*

El siguiente método coloca en una lista de cadenas, que puede ser la de un *TListBox* o un *TMemo*, la definición de los campos de una tabla, en formato parecido al de SQL:

```
procedure LeerDefinicion(const ADB, ATable: string;
   Lista: TStrings);
var
   I: Integer;
   S: string;
begin
   with TTable.Create(nil) do
   try
      DatabaseName := ADB;
      TableName := ATable;
      FieldDefs.Update;
      Lista.Clear;
      Lista.Add('create table ' + ATable + '(');
```

```
      for I := 0 to FieldDefs.Count - 1 do
         with FieldDefs.Items[I] do
         begin
            S := GetEnumName(TypeInfo(TFieldType), Ord(DataType));
            S := '    ' + Name + ' ' + Copy(S, 3, Length(S) - 2);
            if Size <> 0 then
              S := S + '(' + IntToStr(Size) + ')';
            if Required then
              AppendStr(S, ' not null');
            if I < FieldDefs.Count - 1 then
              AppendStr(S, ',');
            Lista.Add(S);
         end;
      Lista.Add(')');
   finally
      Free;
   end;
end;
```

Hay un par de trucos en el código anterior. Uno de estos trucos es la conversión de enumerativo a cadena, utilizando la información de tipos de Delphi; esto ya lo vimos en el capítulo 13, sobre técnicas de gestión de ventanas. Cuando tenemos el nombre del tipo de datos, utilizamos la función *Copy* para eliminar los dos primeros caracteres: los nombres de los tipos de datos definidos en *TFieldType* comienzan todos con el prefijo *ft*: *ftString*, *ftFloat*, etc. Este procedimiento puede llamarse después del siguiente modo:

```
procedure TForm1.MostrarInfoClick(Sender: TObject);
begin
   LeerDefinicion('DBDEMOS', 'EMPLOYEE', ListBox1.Items);
end;
```

Creación de tablas

Aunque personalmente prefiero crear tablas mediante instrucciones SQL, es posible utilizar propiedades y métodos del componente *TTable* para esta operación. La razón de mi preferencia es que Delphi no ofrece mecanismos para la creación directa de restricciones de rango, de integridad referencial y valores por omisión para columnas; para esto, tenemos que utilizar llamadas directas al BDE, si tenemos que tratar con tablas Paradox, o utilizar SQL en las bases de datos que lo permiten.

De cualquier forma, las tablas simples se crean fácilmente, y la clave la tienen las propiedades *FieldDefs* e *IndexDefs*. Esta última propiedad se estudiará en el capítulo sobre índices, por lo cual aquí solamente veremos cómo crear tablas sin índices. La idea es llenar la propiedad *FieldDefs* mediante llamadas al método *Add*, y llamar al final al método *CreateTable*. Por ejemplo:

```
procedure TForm1.CrearTabla;
begin
   with TTable.Create(nil) do
   try
      DatabaseName := 'DBDEMOS';
      TableName := 'Whiskeys';
      TableType := ttParadox;
      FieldDefs.Add('Código', ftAutoInc, 0, False);
      FieldDefs.Add('Marca', ftString, 20, True);
      FieldDefs.Add('Precio', ftCurrency, 0, True);
      FieldDefs.Add('Puntuación', ftSmallInt, 0, False);
      CreateTable;
   finally
      Free;
   end;
end;
```

El alias que asociamos a *DatabaseName* determina el formato de la tabla. Si el alias es estándar, tenemos que utilizar la propiedad *TableType* para diferenciar las tablas Paradox de las tablas dBase.

Sin embargo, nos falta saber cómo crear índices y claves primarias utilizando esta técnica. Necesitamos manejar la propiedad *IndexDefs*, que es en muchos sentidos similar a *FieldDefs*. En el capítulo sobre índices explicaremos el uso de dicha propiedad. Por ahora, adelantaremos la instrucción necesaria para crear un índice primario para la tabla anterior, que debe añadirse antes de la ejecución de *CreateTable*:

```
      // …
      IndexDefs.Add('', 'Código', [ixPrimary]);
      // …
```

Hasta Delphi 3 las definiciones de campos no eran visibles en tiempo de diseño, pero Delphi 4 ha movido la declaración de la propiedad *FieldDefs* a la sección **published**. De esta forma, cuando queremos crear una tabla en tiempo de ejecución podemos ahorrarnos las llamadas al método *Add* de *FieldDefs*, pues esta propiedad ya viene configurada desde el tiempo de diseño. Hay dos formas de rellenar *FieldDefs*: entrar a saco en el editor de la propiedad, o leer las definiciones desde una tabla existente. Para esto último, ejecute el comando *Update table definition*, desde el menú de contexto de la tabla, que adicionalmente asigna *True* a su propiedad *StoreDefs*.

Delphi 4 añade una propiedad *ChildDefs* a los objetos de clase *TFieldDef*, para tener en cuenta a los campos ADT de Oracle 8.

Y como todo lo que empieza tiene que acabar, es bueno saber cómo eliminar y renombrar tablas. El método *DeleteTable* permite borrar una tabla. La tabla debe estar cerrada, y tenemos que especificar el nombre del alias, el de la tabla, y el tipo de tabla, de ser necesario. Otro método relacionado es *RenameTable*:

```
procedure TTable.RenameTable(const NuevoNombre: string);
```

Este método solamente puede utilizarse con tablas Paradox y dBase, y permite renombrar en una sola operación todos los ficheros asociados a la tabla: el principal, los índices, los de campos memos, etc.

> Las técnicas de creación de tablas desde Delphi no resuelven un problema fundamental: la creación de restricciones, principalmente las de integridad referencial, y de valores por omisión. Esta carencia no es grave en el caso de los sistemas SQL, pues la forma preferida de crear tablas en estos sistemas es mediante instrucciones SQL. Pero sí es un problema para Paradox y dBase. Lógicamente, el BDE permite este tipo de operaciones mediante llamadas al API de bajo nivel, pero estas son demasiado complicadas para analizarlas aquí. En el CD que acompaña al libro he incluido un componente que sirve para este propósito. De todos modos, soy partidario de evitar la creación dinámica de tablas en la medida de lo posible.

Capítulo

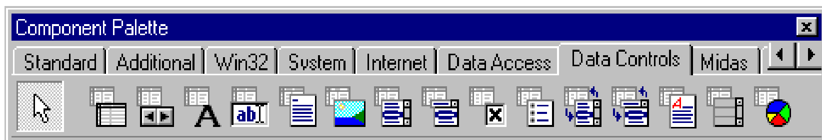
17

Controles de datos y fuentes 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 de Delphi 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[16] 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:

[16] Existen.

- 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 asociados 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 15 sobre conjuntos de datos explicamos 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.

Los controles de datos de Delphi son los siguientes:

Nombre de la clase	Explicación
Controles orientados a campos	
TDBLabel	Textos no modificables (no consumen recursos)
TDBEdit	Cuadros de edición
TDBMemo	Textos sin formato con múltiples líneas
TDBImage	Imágenes BMP y WMF
TDBListBox	Cuadros de lista (contenido fijo)
TDBComboBox	Cuadros de combinación (contenido fijo)
TDBCheckBox	Casillas de verificación (dos estados)
TDBRadioGroup	Grupos de botones (varios valores fijos)
TDBLookupListBox	Valores extraídos desde otra tabla asociada
TDBLookupComboBox	Valores extraídos desde otra tabla asociada
TDBRichEdit	Textos en formato RTF (Delphi 3,4)

Nombre de la clase	Explicación
Controles orientados a conjuntos de datos	
TDBGrid	Rejillas de datos para la exploración
TDBNavigator	Control de navegación y estado
TDBCtrlGrid	Rejilla que permite incluir controles (Delphi 2,3,4)

En este capítulo nos ocuparemos principalmente de los controles orientados a campos. El resto de los controles serán estudiados en el siguiente capítulo.

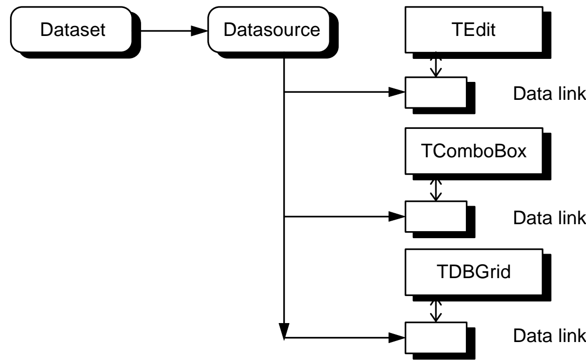
> He omitido intencionalmente el control *TDBChart* de la lista anterior, por trabajar con un sistema distinto a los demás. Este componente se estudiará más adelante.

Los enlaces de datos

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 por el programador de Delphi: *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 incluido en Delphi 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 Delphi 1, el Experto de Datos (*Database form expert*) para crear un formulario 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. El otro inconveniente es que siempre genera un nuevo componente *TTable* ó *TQuery*, no permitiendo utilizar componentes existentes de estos tipos. Esto es un problema a partir de Delphi 2, donde lo usual es definir primero los conjuntos de datos en un módulo aparte, para poder programar reglas de empresa.

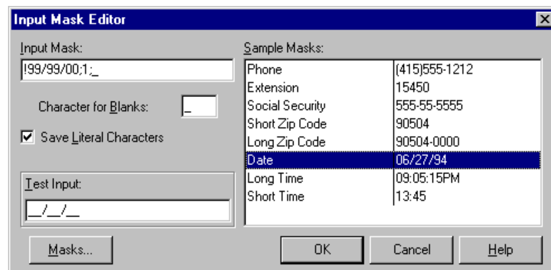
A partir de Delphi 2 podemos arrastrar campos desde el Editor de Campos sobre un formulario. Cuando hacemos esto, se crea automáticamente un control de datos asociado al campo. Junto con el control, se crea también una etiqueta, de clase *TLabel*, con el título extraído de la propiedad *DisplayLabel* 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 *TDBLookupComboBox*. Si es un memo o un campo gráfico, se crea un *TDBMemo* o un *TDBImage*. Si el campo es lógico, se crea un *TDBCheckBox*. Estas reglas implícitas, sin embargo,

pueden modificarse por medio del Diccionario de Datos. 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. 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 multilíneas. 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, Delphi 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 seleccio-

nado. 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
            Self.SelectNext(TWinControl(Sender), True, True);
end;
```

> Hasta la versión 3, Delphi arrastró un pequeño problema en relación con los cuadros de edición asociados a un campo numérico: la propiedad *MaxLength* siempre se hacía 0 cuando se crea el control. Aunque asignásemos algo a esta propiedad en tiempo de diseño, siempre se perdía en tiempo de ejecución. Delphi 4 ha corregido el error.

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 nuevo componente *TDBRichEdit*, de Delphi 3 y 4.

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
Alignment	La alineación del texto dentro del control.
Lines	El contenido del control, como una lista de cadenas de caracteres
ScrollBars	Determina qué barras de desplazamiento se muestran.
WantTabs	Si está activa, el editor interpreta las tabulaciones como tales; en caso contrario, sirven para seleccionar el próximo control.
WordWrap	Las líneas pueden dividirse si sobrepasan el extremo derecho del editor.

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.

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 desciende 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
csSimple	La lista siempre está desplegada. Se pueden teclear valores que no se encuentran en la lista.
csDropDown	La lista está inicialmente recogida. Se pueden teclear valores que no se encuentran en la lista.
csDropDownList	La lista está inicialmente recogida. No se pueden teclear valores que no se encuentren en la lista.
csOwnerDrawFixed	El contenido de la lista lo dibuja el programador. Líneas de igual altura.
csOwnerDrawVariable	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 *Antártida*, 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 Altura: 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 una tabla de países:

```
procedure TForm1.FormCreate(Sender: TObject);
var
   Bmp: TBitmap;
begin
   with TTable.Create(nil) do
   try
      DatabaseName := 'Prueba';
      TableName := 'Paises.Db';
      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, Canvas do
   begin
      FillRect(Rect);
      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. Aunque la propiedad se encuentra en la ayuda de Delphi, si se busca directamente, no aparece en la lista de propiedades de la referencia de

estos componentes. *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 obstante, 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
DataSource	La fuente de datos original. Es la que se modifica.
DataField	El campo original. Es el campo que contiene la referencia.
ListSource	La fuente de datos a la que se hace referencia.

Propiedad	Significado
KeyField	El campo al que se hace referencia.
ListField	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.

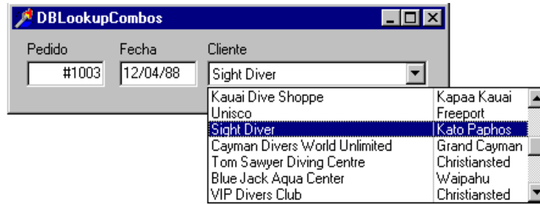
Los combos de búsquedas funcionan con el estilo equivalente al *csDropDownList* de los controles *TDBComboBox*. Esto quiere decir que no se pueden introducir valores que no se encuentren en la tabla de referencia. Pero también significa que podemos utilizar las búsquedas incrementales por teclado. Y 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. ¿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.

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 *TDBCheckBox*. 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;Bai';
DBCheckBox1.ValueUnchecked := 'No;Ez';
```

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 metaficheros, 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:

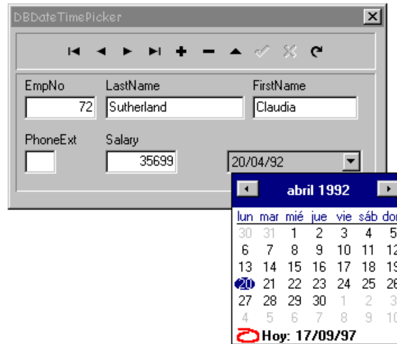
Propiedad	Significado
Center	La imagen aparece centrada o en la esquina superior izquierda
Stretch	Cuando es verdadera, la imagen se expande o contrae para adaptarse al área del control.
QuickDraw	Si es verdadera, se utiliza la paleta global. El dibujo es más rápido, pero puede perderse calidad en ciertas 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

Todas las versiones de Delphi nos ofrecen un componente *TCalendar*, que permite la edición y visualización de fechas como en un calendario. Este control se encuentra en la página *Samples* 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 y no tenemos tiempo y paciencia para crear un nuevo componente? La solución consiste en utilizar los eventos del componente *TDataSource* para convertir a *TCalendar*, pobre Cenicienta, en una hermosa princesa, capaz de comunicarse con todo tipo de bases de datos. La figura muestra la aplicación de esta técnica al componente *TDateTimePicker* de Delphi 3 y 4; aquí hemos utilizado *TCalendar* para que los programadores de Delphi 2 puedan seguir el ejemplo.



Los eventos del componente *TDataSource* son estos tres:

Evento	Se dispara...
OnStateChange	Cada vez que cambia el estado de la tabla asociada
OnDataChange	Cuando cambia el contenido del registro actual
OnUpdateData	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*:

```
procedure TForm1.DataSource1StateChange(Sender: TObject);
const
   Ventana: TForm = nil;
   Lista: TListBox = nil;
begin
   if Ventana = nil then
   begin
      Ventana := TForm.Create(Self);
      Lista := TListBox.Create(Ventana);
      Lista.Align := alClient;
      Lista.Parent := Ventana;
      Ventana.Show;
   end;
   Lista.Items.Add(GetEnumName(TypeInfo(TDataSetState),
      Ord(Table1.State)));
end;
```

Si queremos que este código compile, necesitamos incluir las unidades *StdCtrls* y *TypInfo* en alguna de las cláusulas **uses** de la unidad actual. La primera unidad contiene la declaración del tipo *TListBox*, mientras que la segunda declara las funciones *GetEnumName* y *TypeInfo*.

¿Y qué hay de la promesa de visualizar campos de fechas en un *TCalendar*? Es fácil: cree otra ventana típica, con la tabla, la fuente de datos, una barra de navegación y unos cuantos *TDBEdit*. Utilice, por ejemplo, la tabla *employee.db* del alias *dbdemos*. 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 *TCalendar*. Finalmente seleccione la fuente de datos y cree un manejador para el evento *OnDataChange*:

```
procedure TForm1.DataSource1DataChange(Sender: TObject;
   Field: TField);
begin
   Calendar1.CalendarDate := 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 **nil**, mientras que en el segundo apunta al campo que ha sido modificado. Le propongo al lector 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'] := Calendar1.CalendarDate;
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.Calendar1Change(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 relee 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, 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;
      Calendar1.CalendarDate := Table1['HireDate'];
   finally
      FCambiando := False;
   end;
end;

procedure TForm1.Calendar1Change(Sender: TObject);
begin
   if not FCambiando then
   try
      FCambiando := True;
      DataSource1.Edit;
   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.Calendar1Exit(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 NombreFichero: string);
procedure TBlobField.SaveToFile(const NombreFichero: 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:

tbImagenes	La tabla de imágenes.
tbImagenesFoto	El campo correspondiente a la columna *Foto*.
OpenDialog1	Cuadro de apertura de ficheros, con un filtro adecuado para cargar ficheros gráficos.
bnCargar	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 OpenDialog1.Execute then
   begin
      if not (tbImagenes.State in dsEditModes) then tbImagenes.Edit;
      tbImagenesFoto.LoadFromFile(OpenDialog1.FileName);
   end;
end;
```

Observe que este método no depende en absoluto 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 Delphi, 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.Create(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(Sender: TObject);
begin
   if OpenDialog1.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 la unidad *JPEG* en la cláusula **uses** de la unidad.

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;
   Graphic: TGraphic;
begin
   if tbImagenesFoto.IsNull then
      Image1.Picture.Graphic := nil
   else
   begin
      BS := TBlobStream.Create(tbImagenesFoto, bmRead);
      try
         Graphic := TJPEGImage.Create;
         try
            Graphic.LoadFromStream(BS);
            Image1.Picture.Graphic := Graphic;
         finally
            Graphic.Free;
         end;
      finally
         BS.Free;
      end;
   end;
end;
```

La clase *TJPEGImage* está definida en la unidad *JPEG*, y desciende 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.

Capítulo

18

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 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 *autoedición* 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 29, sobre eventos de transición de estados.

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. En Delphi 3 incluso pueden 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 a partir de Delphi 2, y sus valores por omisión:

Opción	PO	Significado
dgEditing	Sí	Permite la edición de datos sobre la rejilla
dgAlwaysShowEditor	No	Activa siempre el editor de celdas
dgTitles	Sí	Muestra los títulos de las columnas
dgIndicator	Sí	La primera columna muestra el estado de la tabla
dgColumnResize	Sí	Cambiar el tamaño y posición de las columnas
dgColLines	Sí	Dibuja líneas entre las columnas
dgRowLines	Sí	Dibuja líneas entre las filas
dgTabs	Sí	Utilizar tabulaciones para moverse entre columnas
dgRowSelect	No	Seleccionar filas completas, en vez de celdas
dgAlwaysShowSelection	No	Dejar siempre visible la selección
dgConfirmDelete	Sí	Permite confirmar los borrados
dgCancelOnExit	Sí	Cancela inserciones vacías al perder el foco
dgMultiSelect	No	Permite seleccionar varias filas a la vez.

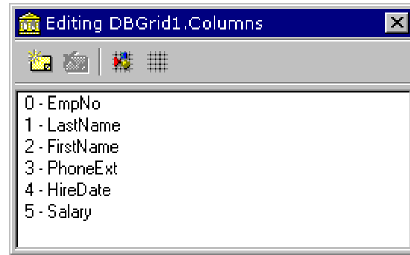
En Delphi 1 no se dispone de la opción *dgMultiSelect*. Curiosamente, la documentación impresa de Delphi 2 no incluía ayuda sobre cómo trabajar con la misma. Más adelante veremos ejemplos acerca de esta opción.

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.

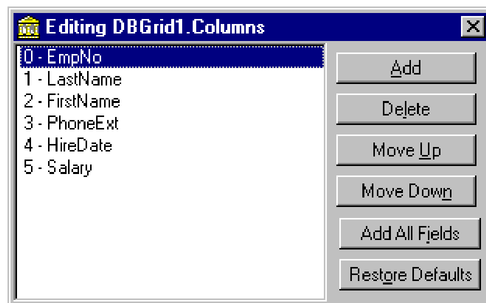
Columnas a la medida

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 Delphi 1, las tareas mencionadas se llevaban a cabo modificando propiedades de los componentes de campos de la tabla asociada a la rejilla. Por ejemplo, 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 si queríamos ocultar una columna, debíamos utilizar la propiedad *Visible* del componente de campo.

Esto ocasionaba bastantes problemas; el problema más grave en Delphi 1 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. A partir de Delphi 2 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 apariencia. 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*, no disponible en Delphi 1, 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. El editor de propiedades es diferente en Delphi 2, 3 y 4. En el primero, aparece un cuadro de diálogo modal y en las otras dos versiones se utiliza el editor genérico de colecciones, que es no modal. Este es, por ejemplo, el editor de columnas de Delphi 3:



La propiedades que nos interesan de las columnas son:

	Propiedad	Significado
	Alignment	Alineación de la columna
	ButtonStyle	Permite desplegar una lista desde una celda, o mostrar un botón de edición.
	Color	Color de fondo de la columna.
	DropDownRows	Número de filas desplegables.
	FieldName	El nombre del campo que se visualiza.
	Font	El tipo de letra de la columna.
	PickList	Lista opcional de valores a desplegar.
	ReadOnly	Desactiva la edición en la columna.
	Width	El ancho de la columna, en píxeles.
Title	*Alignment*	Alineación del título de la columna.
	Caption	Texto de la cabecera de columna.
	Color	Color de fondo del título de columna.
	Font	Tipo de letra del título de la columna.

A esta lista, Delphi 4 añade la propiedad *Expanded*, que 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. Más adelante, en el capítulo 25, que está dedicada a Oracle, volveremos a este asunto.

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.FieldCount - 1 do
      with DBGrid1.Columns[I] do
         if Field.IsIndexField then Font.Color := clBlue;
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 - 13;
end;
```

El valor que se resta en la fórmula se ha elegido por prueba y error, y corresponde al ancho de la columna de indicadores y a 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\' +
      Self.Name + '.' + DBGrid1.Name) do
   try
      for I := 0 to DBGrid1.Columns.Count - 1 do
         with DBGrid1.Columns.Items[I] do
            Width := ReadInteger('Width', FieldName, Width);
```

```
      finally
         Free;
      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\' +
      Self.Name + '.' + DBGrid1.Name) do
   try
      for I := 0 to DBGrid1.Columns.Count - 1 do
         with DBGrid1.Columns.Items[I] do
            WriteInteger('Width', FieldName, Width);
   finally
      Free;
   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

Como hemos visto, a partir de Delphi 2, las rejillas de datos permiten que una columna despliegue una lista de valores para que el usuario seleccione uno de ellos. Esto puede suceder en dos contextos diferentes:

- Cuando el campo visualizado en una columna es un campo de búsqueda (*lookup field*).
- Cuando 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 *TDBLookupCom-*

boBox, 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.

Listas desplegables en rejillas

PartNo	VendorNo	Vendor	Description	Cost	ListPrice
900	3820	Techniques	Dive kayak	$1.356,75	$3.999,95
912	3820	J.W. Luscher Mfg.	Underwater Diver Vehicle	$504,00	$1.680,00
1313	3511	Scuba Professional	Regulator System	$117,50	$250,00
1314	5641	Divers' Supply Sho / Techniques	Second Stage Regulator	$124,10	$365,00
1316	3511	Perry Scuba	Regulator System	$119,35	$341,00
1320	3511	Beauchat, Inc. / Amor Aqua	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.

> En Delphi 4 los puntos suspensivos aparecen también con los campos *TReferenceField* y *TDataSetField*, de Oracle 8. Cuando se pulsa el botón, aparece una rejilla emergente con los valores anidados dentro del campo. El nuevo método *ShowPopupEditor* permite invocar al editor desde el programa.

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?

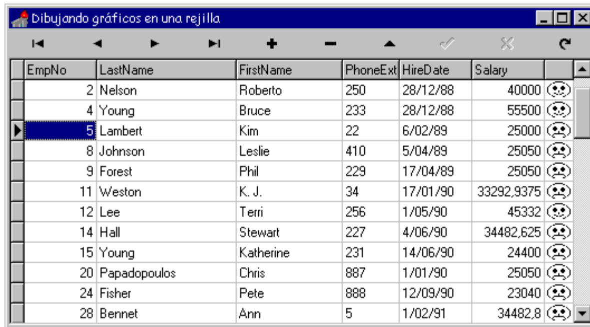
La clave para evitar este dolor de cabeza es el método *DefaultDrawColumnCell*, perteneciente a las rejillas de datos. Este método realiza el dibujo por omisión de las celdas, y puede ser llamado en el interior de la respuesta a *OnDrawColumnCell*. Los parámetros de este método son exactamente los mismos 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
   with Sender as TDBGrid do
   begin
      if tbParts['OnOrder'] >= tbParts['OnHand'] then
         Canvas.Font.Color := clRed;
```

```
            DefaultDrawColumnCell(Rect, DataCol, Column, State);
        end;
    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 tbEmpleados['Salary'] >= 45000 then
            Canvas.StretchDraw(Rect, CaraAlegre.Picture.Graphic)
        else
            Canvas.StretchDraw(Rect, CaraTriste.Picture.Graphic)
end;
```



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. El siguiente procedimiento auxiliar se ocupa de los detalles del dibujo, y asume que se le suministra el *Canvas* de una rejilla, dos mapas de bits de iguales dimensiones con una casilla de verificación marcada y sin marcar, el rectángulo que delimita la celda y el valor lógico a representar:

```
procedure DrawCheckBox(ACanvas: TCanvas;
    bmpChecked, bmpUncheked: TBitmap;
    const Rect: TRect; Value: Boolean);
var
    X, Y: Integer;
```

```
begin
   ASSERT(bmpChecked.Width = bmpUnchecked.Width);
   ASSERT(bmpChecked.Height = bmpUnchecked.Height);
   ACanvas.Pen.Color := clGray;
   ACanvas.MoveTo(Rect.Left, Rect.Bottom);
   ACanvas.LineTo(Rect.Right, Rect.Bottom);
   ACanvas.LineTo(Rect.Right, Rect.Top);
   ACanvas.Brush.Color := clSilver;
   ACanvas.FillRect(Rect);
   X := (Rect.Left + Rect.Right - bmpChecked.Width) div 2;
   Y := (Rect.Top + Rect.Bottom - bmpChecked.Height) div 2;
   if Value then
      ACanvas.Draw(X, Y, bmpChecked)
   else
      ACanvas.Draw(X, Y, bmpUnchecked);
end;
```

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);
begin
   if CompareText(Column.FieldName, 'ACTIVO') = 0 then
      DrawCheckBox(DBGrid1.Canvas,
         imgChecked.Picture.Bitmap, imgUnchecked.Picture.Bitmap,
         Rect, Table1['ACTIVO'])
   else
      DBGrid1.DefaultDrawColumnCell(Rect, DataCol, Column, State);
end;
```

Recuerde que mientras más complicado sea el dibujo, más tardará en redibujarse la rejilla.

Más eventos de rejillas

En Delphi 3 se añadieron un par de nuevos eventos a las rejillas de datos. Estos son *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
DatabaseName	*IBLOCAL*
TableName	*EMPLOYEE*
Active	*True*

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
      Table1.IndexFieldNames := Column.FieldName;
   except
   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 2 y las de versiones posteriores consiste en el comportamiento de la barra de desplazamiento vertical que tienen asociadas. En Delphi 1 y 2, 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 3 corrigió esta situación para las tablas locales aunque, por supuesto, las cosas siguen funcionando igual para las bases de datos SQL.

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*, que no aparecía documentada en los manuales de Delphi 2[17]. Esta propiedad es del tipo *TBookmarkList*, una lista de marcas de posición, cuyos principales métodos y propiedades son los siguientes:

Propiedades/Métodos	Propósito
Count	Cantidad de filas seleccionadas.
Items[*I*: *Integer*]	Lista de posiciones seleccionadas.
CurrentRowSelected	¿Está seleccionada la fila activa?
procedure *Clear*;	Eliminar la selección.
procedure *Delete*;	Borrar las filas seleccionadas.
procedure *Refresh*;	Actualizar la selección, eliminando filas borradas.

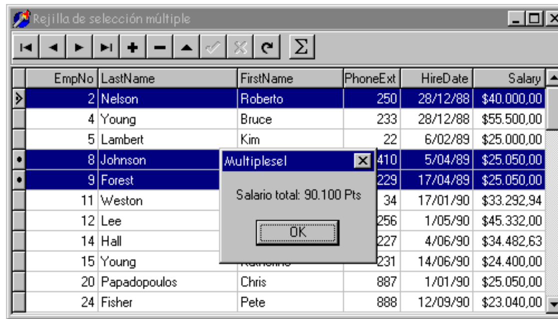
El siguiente método muestra cómo sumar los salarios de los empleados seleccionados en una rejilla con selección múltiple:

```
procedure TForm1.bnSumarClick(Sender: TObject);
var
   BM: TBookmarkStr;
   Total: Double;
   I: Integer;
begin
   tbEmpleados.DisableControls;
   BM := tbEmpleados.Bookmark;
   try
      Total := 0;
      for I := 0 to DBGrid1.SelectedRows.Count - 1 do
      begin
         tbEmpleados.Bookmark := DBGrid1.SelectedRows.Items[I];
         Total := Total + tbEmpleados['Salary'];
      end;
```

[17] Esta melodía ya nos es familiar.

```
    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 noveau* 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, y perpetrar el correspondiente atentado contra la estética.

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 una razón poderosa para utilizar siempre que sea posible 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 desagradable 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 de Delphi 3 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 están definidas, en Delphi 2, en el fichero de recursos *dbconsts.res*. Es posible editar este fichero para cambiar estos valores. En Delphi 3 y 4, se definen en la unidad *dbconsts.pas*, utilizando 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 aprovechar los métodos de clase mostrados en el capítulo de técnicas de gestión de ventanas 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
      Form2.Mostrar(0);       // 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 el método *BtnClick* va a disparar también el evento asociado a *OnClick* de la barra de navegación.

Delphi 3 introdujo un nuevo evento para las barras de navegación, *BeforeAction*, que es disparado cuando se pulsa un botón, 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.DBNavigator1BeforeAction(Sender: TObject);
begin
   (Sender as TDBNavigator).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:

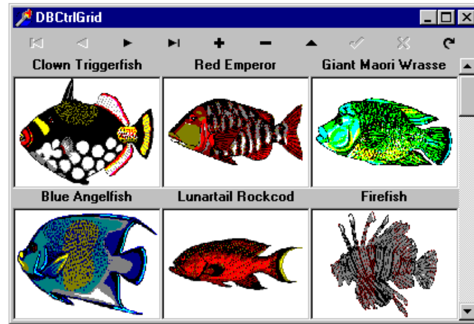
Propiedad	Significado
Orientation	Orientación del desplazamiento de paneles.
ColCount	Número de columnas.
RowCount	Número de filas.
PanelBorder	¿Tiene borde el panel?[18]
PanelWidth	Ancho de cada panel, en píxeles.
PanelHeight	Altura de cada panel, en píxeles.

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, combos, casillas de verificación, memos, imágenes o cualquier control que nos dicte nuestra imaginación. La excepción, en Delphi 3 y 4, son los *TDBRadioGroup* y los *TDBRichEdit*.

En Delphi 2, sin embargo, tampoco se podían colocar directamente controles *DBMemo* y *DBImage* en una rejilla. La causa inmediata era que estos controles carecían de la opción *csReplicatable* 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.

---

[18] ¿Y sueñan los androides con ovejas eléctricas?



La implementación de componentes duplicables en Delphi 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.

Capítulo

19

Indices

UNO DE LOS ASPECTOS BÁSICOS de la implementación de todo sistema de bases de datos es la definición y uso de índices. Los índices están presentes en casi cualquier área de la creación y explotación de una base de datos:

- Mediante los índices se pueden realizar búsquedas rápidas.
- Las búsquedas rápidas se aprovechan en la implementación de las restricciones de integridad referencial.
- Si los índices se implementan mediante árboles balanceados (*b-trees*) o alguna técnica equivalente, que es lo más frecuente, nos sirven también para realizar diferentes ordenaciones lógicas sobre los registros de las tablas.
- Sobre un índice que permite una ordenación se pueden definir eficientemente rangos, que son restricciones acerca de los registros visibles en un momento determinado.
- Gracias a las propiedades anteriores, los índices se utilizan para optimizar las operaciones SQL tales como las selecciones por un valor y los encuentros entre tablas.

En el tercer punto de la lista anterior, he mencionado indirectamente la posibilidad de índices que no estén implementados mediante árboles balanceados. Esto es perfectamente posible, siendo el caso más típico los índices basados en técnicas de *hash*, o desmenuzamiento de la clave. Estos índices, sin embargo, no permiten la ordenación de los datos básicos. Por esto, casi todos los índices de los sistemas de bases de datos más populares están basados en árboles balanceados y permiten tanto la búsqueda rápida como la ordenación.

Indices en Paradox

Paradox permite trabajar con un índice primario por cada tabla y con varios índices secundarios. Todos estos índices pueden ser mantenidos automáticamente por el sistema, aunque existe la posibilidad de crear índices secundarios no mantenidos.

Cada índice se almacena en un fichero aparte, con el mismo nombre que la tabla con los datos pero con diferente extensión.

Paradox permite el uso directo de índices con claves compuestas, formadas por varios campos. Otras opciones posibles de un índice Paradox es ignorar las mayúsculas y minúsculas en los campos de cadenas (la opción por omisión), la restricción de claves únicas y la posibilidad de ordenación en sentido descendente.

El índice primario es un índice con claves únicas que se utiliza para la definición y mantenimiento de la clave primaria de una tabla. Puede haber solamente un índice primario por cada tabla. En Paradox, este índice primario no tiene nombre propio. Si se define una clave primaria para una tabla, los campos de esta clave deben ser los primeros campos de la definición de la tabla. Así, por ejemplo, si la tabla *items.db* de la base de datos *dbdemos* de Delphi tiene una clave primaria formada por los campos *OrderNo* e *ItemNo*, estos campos deben ser, respectivamente, el primer y el segundo campo de la tabla. Si una tabla tiene un índice primario es imposible, al menos desde Delphi, afectar la posición de inserción de los registros. Estos índices primarios son utilizados también por el mecanismo de integridad referencial de Paradox.

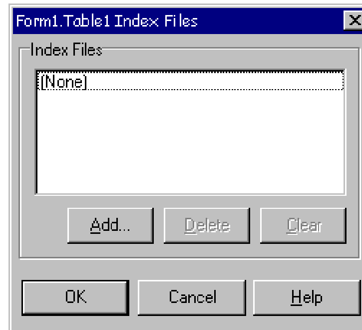
Indices en dBase

En dBase, a partir de la versión cuatro, se utilizan ficheros con extensión *mdx* para los índices. Cada uno de estos ficheros contiene en realidad un grupo de índices; la palabra inglesa que se utiliza para referirse a cada uno de los índices presentes en este superfichero es *tag*. De estos ficheros *mdx* hay uno especial: aquel cuyo nombre coincide con el del fichero de datos de extensión *dbf*. A este fichero de índices se le conoce como el *índice de producción*: para la tabla *animals.dbf* el índice de producción es el fichero *animals.mdx*. El índice de producción se abre automáticamente al abrirse el fichero de datos, y los índices pertenecientes al mismo, en consecuencia, se mantienen actualizados sin intervención del programador. Estos son, además, los índices que serán visibles inicialmente al programador de Delphi.

Pero también es posible la existencia de otros ficheros *mdx* asociados a una tabla. Estos índices secundarios no se actualizan automáticamente. La razón fundamental de esta restricción consiste en que al abrir el fichero principal *dbf* no existe forma de saber, sin intervención del programador, si tiene algún fichero *mdx* secundario asociado. Al menos, en Delphi 1.

A partir de Delphi 2 el problema se resuelve mediante la nueva propiedad *IndexFiles* de las tablas. Esta propiedad, de tipo *TStrings*, almacena la lista de nombres de ficheros *mdx* y *ndx* (¡sí, los de dBase III!) que deben abrirse junto a la tabla para su mantenimiento. Normalmente, esta propiedad debe asignarse en tiempo de diseño, pero es posible añadir y eliminar dinámicamente elementos en esta propiedad. El mismo

resultado se obtiene mediante los métodos *OpenIndexFile* y *CloseIndexFile*, de la clase *TTable*. Cualquier componente de un índice secundario abierto puede seleccionarse, en la propiedad *IndexName*, para utilizarlo como índice activo de la tabla.



Un problema relacionado es cómo utilizar un fichero de dBase IV cuando no tenemos el índice de producción correspondiente. El hecho es que el fichero de extensión *dbf* contiene una marca para indicar la presencia del correspondiente fichero *mdx*. Si la marca está puesta y no aparece el índice, no se puede abrir el fichero. Sin embargo, eliminar esta marca es relativamente fácil; se encuentra en el byte 29 de la cabecera del fichero, y el siguiente código muestra como se puede eliminar:

```
procedure EliminarMDX(const FicheroDBF: string);
var
   F: file of Byte;
   Cero: Byte;
begin
   Cero := 0;
   AssignFile(F, FicheroDBF);
   Reset(F);
   try
      Seek(F, 28);
      Write(F, Cero);
   finally
      CloseFile(F);
   end;
end;
```

No, no es un error: buscamos el byte 28 con el procedimiento *Seek* porque las posiciones dentro del fichero se cuentan a partir de cero.

En las versiones anteriores a Visual dBase 7, no existe el concepto de índice primario. Está de más decir que tampoco hay un soporte automático para restricciones de integridad referencial. No se soportan directamente los índices con claves compuestas. Hay que utilizar, en cambio, un retorcido mecanismo conocido como *índices de expresiones*; los programadores de xBase ensalzan este sistema, pero no hay que hacerles mucho caso, porque a casi todo se acostumbra uno. El lograr un índice insensible a mayúsculas y minúsculas también requiere el uso de expresiones. Veremos que

la presencia de un índice basado en expresiones nos obliga a utilizar técnicas sado-masoquistas en Delphi para poder trabajar con el mismo; aunque al final logramos el objetivo, estamos limitando las posibilidades de portabilidad de la aplicación.

Indices en InterBase

Los índices en InterBase muestran el trabajo típico con índices en una base de datos SQL. InterBase permite la creación explícita de índices, con claves simples y compuestas, únicas o no. Pero también se crean índices automáticamente para mantener las restricciones de integridad referencial. Por ejemplo, la definición de una clave primaria para una tabla provoca la creación y mantenimiento automático de un índice único, con el nombre *rdb$primaryN*. La *N* al final del nombre representa un valor entero generado por el sistema, que garantice la unicidad del nombre del índice. Del mismo modo, para las claves foráneas (*foreign keys*) se mantienen índices con el nombre *rdb$foreignN*. Estos índices son sumamente útiles al establecer, más adelante, relaciones *master/detail* entre las tablas.

Los índices de InterBase son sensibles a mayúsculas y minúsculas, aunque el orden alfabético depende del conjunto de caracteres y el orden definido al crear la base de datos.

Indices en MS SQL Server

Además de las opciones habituales aplicables a índices, MS SQL Server permite definir *índices agrupados* (*clustered indexes*). En un índice agrupado, las filas de la tabla quedan ordenadas físicamente, no sólo de forma lógica, de acuerdo al orden definido por el índice. Esto hace que las búsquedas sean muy rápidas, sobre todo cuando no buscamos un valor aislado, sino todo un rango de valores. Está claro, sin embargo, que solamente puede haber un índice agrupado por tabla.

Las claves primarias generan automáticamente un índice para imponer la restricción de unicidad; el nombre de este índice comienza con *PK_*, le sigue el nombre de la tabla, y un número de identificación. Desafortunadamente, las restricciones de integridad referencial no generan índices, como en Paradox e InterBase, sino que hay que crearlos explícitamente. Un programador acostumbrado a otro servidor SQL puede encontrarse una desagradable sorpresa al asumir la existencia automática de estos índices.

En dependencia de una opción que solamente puede especificarse durante la instalación del servidor, los índices de MS SQL Server pueden ser sensibles o no a mayúsculas y minúsculas.

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 números 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.

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[19] 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.

Por último, tenemos la posibilidad de utilizar *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 de la base de datos, manteniendo un solo índice sobre el número de pedido para ambas tablas. Este índice, incluso, puede 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.

---

[19] Si me he equivocado, perdonadme: yo no estudié Geografía Española en la escuela.

Hasta la versión 8, las comparaciones entre cadenas de caracteres en Oracle son sensibles a mayúsculas y minúsculas. Lo mismo sucede con los índices.

Con qué índices podemos contar

El primer paso para trabajar con índices en Delphi es conocer qué índices tenemos asociados a una tabla determinada. Si lo único que nos importa es la lista de nombres de índices, podemos utilizar el método *GetIndexNames*, aplicable a la clase *TTable*. El prototipo de este método es:

```
procedure TTable.GetIndexNames(Lista: TStrings);
```

Este método añade a la lista pasada como parámetro los nombres de los índices que están definidos para esta tabla. Es importante advertir que la acción realizada por *GetIndexNames* es añadir nuevas cadenas a una lista existente, por lo que es necesario limpiar primero la lista, con el método *Clear*, si queremos obtener un resultado satisfactorio. Para que *GetIndexNames* funcione no hace falta que la tabla esté abierta; un objeto de tipo *TTable* necesita solamente tener asignado valores a las propiedades *DatabaseName* y *TableName* para poder preguntar por los nombres de sus índices. Otra particularidad de este método es que no devuelve en la lista el índice primario de las tablas Paradox.

Ahora bien, si necesitamos más información sobre los índices, es en la propiedad *IndexDefs* de la tabla donde debemos buscarla. Esta propiedad, que devuelve un objeto de tipo *TIndexDefs*, es similar en cierto sentido a la propiedad *FieldDefs*, que ya hemos analizado. Podemos utilizar *IndexDefs* incluso con la tabla cerrada, pero para eso tenemos primeramente que aplicar el método *Update* al objeto retornado por esta propiedad. La clase *TIndexDefs* es en esencia una lista de definiciones de índices; cada definición de índice es, a su vez, un objeto de clase *TIndexDef*, en singular. Para acceder a las definiciones individuales se utilizan estas dos subpropiedades de *TIndexDefs*:

```
property Count: Integer;
property Items[Indice: Integer]: TIndexDef;
```

Por su parte, estas son las propiedades de un objeto de clase *TIndexDef*:

Propiedad	Significado
Name	Es el nombre del índice; si éste es el índice primario de una tabla Paradox, la propiedad contiene una cadena vacía.
Fields	Lista de campos de la clave, separados por puntos y comas. Si es un índice de expresiones de dBase, la propiedad contiene una cadena vacía.

Propiedad	Significado
Expression	Expresión que define un índice de expresiones de dBase.
Options	Conjunto de opciones del índice.

La propiedad *Options* es un conjunto al cual pueden pertenecer los siguientes valores:

Opción	Significado
ixPrimary	El índice es el primario de una tabla Paradox.
ixUnique	No se permiten claves duplicadas en el índice.
ixDescending	El criterio de ordenación es descendente.
ixExpression	El índice es un índice de expresiones de dBase.
ixCaseInsensitive	Se ignora la diferencia entre mayúsculas y minúsculas. No aplicable en tablas dBase.

Si no existiera el método *GetIndexNames* se pudiera simular con el siguiente procedimiento:

```
procedure LeerNombresDeIndices(Tabla: TTable; Lista: TStrings);
var
   I: Integer;
begin
   with Tabla.IndexDefs do
   begin
      Update;
      Lista.Clear;     // Una mejora al algoritmo
      for I := 0 to Count - 1 do
         if Items[I].Name <> '' then
            Lista.Add(Items[I].Name);
   end;
end;
```

Pero si lo que deseamos es obtener una lista de las definiciones de índices, similar a la lista desplegable de la propiedad *IndexFieldNames* que estudiaremos en breve, debemos utilizar un procedimiento parecido a este otro:

```
procedure LeerDefiniciones(Tabla: TTable; Lista: TStrings);
var
   I: Integer;
begin
   with Tabla.IndexDefs do
   begin
      Update;
      Lista.Clear;
      for I := 0 to Count - 1 do
         if ixExpression in Items[I].Options then
            Lista.Add(Items[I].Expression)
         else
            Lista.Add(Items[I].Fields);
   end;
end;
```

Especificando el índice activo

Ahora que podemos conocer qué índices tiene una tabla, nos interesa empezar a trabajar con los mismos. Una de las operaciones básicas con índices en Delphi es especificar el *índice activo*. El índice activo determina el orden lógico de los registros de la tabla. Es necesario tener un índice activo para poder realizar las siguientes operaciones:

- Búsquedas rápidas con *FindKey*, *FindNearest* y otros métodos relacionados.
- Activación de rangos.
- Uso de componentes *TDBLookupComboBox* y *TDBLookupListBox* con la tabla asignada indirectamente en su propiedad *ListSource*.
- Tablas en relación *master/detail*. La tabla de detalles debe indicar un índice activo. Esto lo veremos más adelante.

Aunque normalmente en una aplicación de entrada de datos el criterio de ordenación de una tabla es el mismo para toda la aplicación, el cambio de índice activo es una operación posible y completamente dinámica. A pesar de que los manuales de Delphi aconsejan cerrar la tabla antes de cambiar de índice, esta operación puede realizarse con la tabla abierta sin ningún tipo de problemas, en cualquier momento y con un costo despreciable.

Existen dos formas de especificar un índice para ordenar una tabla, y ambas son mutuamente excluyentes. Se puede indicar el nombre de un índice existente en la propiedad *IndexName*, o se pueden especificar los campos que forman el índice en la propiedad *IndexFieldNames*. Si utilizamos *IndexName* debemos tener en cuenta que los índices primarios de Paradox no aparecen en la lista desplegable del editor de la propiedad. Si queremos activar este índice debemos asignar una cadena vacía a la propiedad: este es, por otra parte, su valor por omisión. La propiedad *IndexFieldNames*, en cambio, es más fácil de utilizar, pues en la lista desplegable de su editor de propiedad aparecen los campos que forman la clave del índice; generalmente, esto más informativo que el simple nombre asignado al índice.

En el siguiente ejemplo muestro cómo cambiar en tiempo de ejecución el orden de una tabla utilizando un combo con la lista de criterios posibles de ordenación. Para el ejemplo se necesita, por supuesto, una tabla (*Table1*) y un cuadro de combinación (*ComboBox1*) cuya propiedad *Style* sea igual a *csDropDownList*. Es necesario definir manejadores para el evento *OnCreate* de la tabla y para el evento *OnChange* del combo:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
   // Llamar a la función del epígrafe anterior
   LeerDefiniciones(Table1, ComboBox1.Items);
   // Seleccionar el primer elemento del combo
   ComboBox1.ItemIndex := 0;
```

```
      // Simular el cambio de selección en el combo
      ComboBox1Change(ComboBox1);
   end;

   procedure TForm1.ComboBox1Change(Sender: TObject);
   begin
      Table1.IndexName := Table1.IndexDefs[ComboBox1.ItemIndex].Name;
   end;
```

Observe que, en vez de asignar directamente a *IndexFieldNames* los campos almacenados en el texto del combo, se busca el nombre del índice correspondiente en *IndexDefs*. El motivo son los índices de expresiones de dBase; recuerde que la función *LeerDefiniciones* fue programada para mostrar la expresión que define al índice en estos casos.

Aunque algunos sistemas SQL permiten definir índices descendentes, como InterBase, la activación de los mismos mediante la propiedad *IndexName* no funciona correctamente, pues se utiliza siempre el orden ascendente. La única forma de lograr que los datos aparezcan ordenados descendentemente es utilizar una consulta *TQuery*. Sin embargo, las consultas presentan determinados inconvenientes, sobre los que trataremos más adelante en el capítulo 27, sobre comunicación cliente/servidor.

Especificando un orden en tablas SQL

Si estamos trabajando con una tabla perteneciente a una base de datos orientada a registros (una forma elegante de decir Paradox ó dBase), solamente podemos utilizar los criterios de ordenación pertenecientes a índices existentes. Si quiero ordenar una tabla de empleados por sus salarios y no existe el índice correspondiente mis deseos quedarán insatisfechos. Sin embargo, si la base de datos es una base de datos SQL, puedo ordenar por la columna o combinación de columnas que se me antoje; es

responsabilidad del servidor la ordenación según el método más eficiente. Y para esto se utiliza la propiedad *IndexFieldNames*. En esta propiedad se puede asignar el nombre de una columna de la tabla, o una lista de columnas separadas por puntos y comas. Por supuesto, las columnas deben pertenecer a tipos de datos que permitan la comparación. El orden utilizado es el ascendente.

Se puede realizar una demostración bastante espectacular de esta técnica si estamos visualizando una tabla de InterBase o cualquier otro sistema SQL en una rejilla. En el formulario creamos un menú emergente, de tipo *TPopupMenu*, con una opción *Ordenar*. En el evento *OnPopup* programamos lo siguiente:

```
procedure TForm1.PopupMenu1Popup(Sender: TObject);
begin
   Ordenar1.Checked :=
      Pos(';' + DBGrid1.SelectedField.FieldName + ';',
         ';' + Table1.IndexFieldNames + ';') <> 0;
end;
```

De esta forma el comando de menú aparece inicialmente marcado si el nombre del campo correspondiente a la columna seleccionada se encuentra ya en el criterio de ordenación. Para añadir o eliminar el campo del criterio de ordenación, creamos un manejador para el evento *OnClick* del comando:

```
procedure TForm1.Ordenar1Click(Sender: TObject);
var
   Criterio, Campo: string;
   Posicion: Integer;
begin
   Criterio := ';' + Table1.IndexFieldNames + ';';
   Campo := ';' + DBGrid1.SelectedField.FieldName + ';';
   Posicion := Pos(Campo, Criterio);
   if Posicion <> 0 then
   begin
      Delete(Criterio, Posicion, Length(Campo) - 1);
      Table1.IndexFieldNames := Copy(Criterio, 2,
         Length(Criterio) - 2);
   end
   else if Criterio = ';;' then
      Table1.IndexFieldNames := DBGrid1.SelectedField.FieldName
   else
      Table1.IndexFieldNames := Table1.IndexFieldNames
         + ';' +DBGrid1.SelectedField.FieldName;
end;
```

Le propongo al lector que mejore esta técnica añadiendo al menú emergente la posición del campo en la lista de columnas, si es que está ordenado.

Búsqueda basada en índices

En Delphi es muy fácil realizar una búsqueda rápida sobre el índice activo. Al estar los índices principales de todos los formatos de bases de datos soportados por Delphi implementados mediante árboles balanceados o técnicas equivalentes, es posible realizar dos tipos de búsqueda: la búsqueda exacta de un valor y la búsqueda inexacta en la cual, si no encontramos el valor deseado, nos posicionamos en la fila correspondiente al valor más cercano en la secuencia ascendente o descendente de claves del índice. Estas dos operaciones, a su vez, se pueden realizar en Delphi utilizando directamente ciertos métodos simples o descomponiendo estos métodos en secuencias de llamadas más elementales; por supuesto, esta segunda técnica es más complicada. Comenzaremos por la forma más sencilla y directa, para luego explicar el por qué de la existencia de la segunda.

Para realizar una búsqueda exacta sobre un índice activo se utiliza el método *FindKey*. Su prototipo es:

```
function TTable.FindKey(const Valores: array of const): Boolean;
```

El parámetro *Valores* corresponde a la clave que deseamos buscar. Como la clave puede ser compuesta y estar formada por campos de distintos tipos, se utiliza un vector de valores de tipo arbitrario para contenerla. Ya hemos visto los parámetros de tipo **array of const** en el capítulo 12, que trata sobre los tipos de datos de Delphi.

Si *FindKey* encuentra la clave especificada, la tabla a la cual se le aplica el método cambia su fila activa a la fila que contiene el valor. En ese caso, *FindKey* devuelve *True*. Si la clave no se encuentra, devuelve *False* y no se cambia la fila activa.

Por el contrario, el método *FindNearest* no devuelve ningún valor, porque siempre encuentra una fila:

```
procedure TTable.FindNearest(const Valores: array of const);
```

El propósito de *FindNearest* es encontrar el registro cuyo valor de la clave coincide con el valor pasado como parámetro. Sin embargo, si la clave no se encuentra en la tabla, el cursor se mueve a la primera fila cuyo valor es superior a la clave especificada. De este modo, una inserción en ese punto dejaría espacio para un registro cuya clave fuera la suministrada al método.

La aplicación más evidente del método *FindNearest* es la búsqueda incremental. Supongamos que tenemos una tabla ordenada por una clave alfanumérica y que la estamos visualizando en una rejilla. Ahora colocamos en el formulario un cuadro de edición, de tipo *TEdit*, y hacemos lo siguiente en respuesta al evento *OnChange*:

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
   Table1.FindNearest([Edit1.Text]);
end;
```

Se puede evitar el uso del cuadro de edición si la rejilla está en modo de sólo lectura:

Propiedad	Valor
ReadOnly	*True*
Options	Eliminar *dgEditing*

Necesitamos también declarar una variable de cadena para almacenar la clave de búsqueda actual; ésta se declara en la sección **private** de la declaración de la clase del formulario:

```
private
   Clave: string;  // Se inicializa automáticamente a cadena vacía
```

Después hay que interceptar el evento *OnKeyPress* de la rejilla:

```
procedure TForm1.DBGrid1KeyPress(Sender: TObject; var Key: Char);
begin
   if Key > ' ' then
      AppendStr(Clave, Key)
   else if Key = #8 then
      Delete(Clave, Length(Clave), 1)
   else
      Exit;
   Key := #0;
   Table1.FindNearest([Clave]);
end;
```

El inconveniente principal de esta técnica es que no sabemos en un momento determinado qué clave hemos tecleado exactamente, pero se puede utilizar un texto estático, *TLabel*, para paliar este problema.

Código	Apellidos	Nombre	Extensión	Contrato	Salario
34	Baldwin	Janet	2	21/03/91	$23.300,00
105	Bender	Oliver H.	255	8/10/92	$36.799,00
28	Bennet	Ann	5	1/02/91	$34.482,80
83	Bishop	Dana	290	1/06/92	$45.000,00
109	Brown	Kelly	202	4/02/93	$27.000,00
71	Burbank	Jennifer M.	289	15/04/92	$45.332,00
107	Cook	Kevin	894	1/02/93	$35.500,00
29	De Souza	Roger	288	18/02/91	$25.500,00
121	Ferrari	Roberto	1	12/07/93	$40.500,00
24	Fisher	Pete	888	12/09/90	$23.040,00
9	Forest	Phil	229	17/04/89	$25.050,00
134	Glon	Jacques		23/08/93	$24.855,00
138	Green	T.J.	218	1/11/93	$36.000,00

Implementación de referencias mediante FindKey

Como sabemos, los campos de referencia tienen una implementación directa y sencilla a partir de Delphi 2, pero no contamos con este recurso en Delphi 1. Una de las aplicaciones de los métodos de búsquedas sobre el índice activo es precisamente suplir la falta de este tipo de campos en la versión de 16 bits.

Volvemos al ejemplo de los pedidos de la base de datos *dbdemos*. La tabla *items* representa las cantidades vendidas de un producto en determinado pedido. En cada fila de esta tabla se almacena únicamente el código del producto, *PartNo*, por lo cual para conocer el precio por unidad del mismo hay que realizar una búsqueda en la tabla de artículos, *parts*. Se puede crear un campo calculado, llamémosle *PrecioUnitario*, sobre la tabla *items*, de tipo *Float*, y calcular el valor del mismo durante el evento *OnCalcFields* del siguiente modo:

```
procedure TForm1.tbItemsCalcFields(Sender: TDataSet);
begin
   if tbParts.FindKey([tbItemsPartNo.Value]) then
      tbItemsPrecioUnitario.Value := tbPartsListPrice.Value;
end;
```

En Delphi 2, el nuevo campo hubiera sido de tipo *Currency*, pero desgraciadamente este tipo no existe en la versión anterior. Hemos supuesto que están creados ya los campos para cada una de las dos tablas utilizadas.

Este código tiene un par de detalles importantes para asimilar. En primer lugar, la tabla *parts* tiene que tener al índice primario, definido sobre el código de producto, como índice activo. En segundo lugar, este algoritmo cambia la posición de la fila activa de la tabla de artículos. Es por ello que no debe utilizarse esta misma tabla para visualizar los artículos, pues el usuario verá como el cursor de la misma se mueve desenfrenadamente de vez en cuando. En realidad, el algoritmo utilizado por Delphi para los campos de referencia utiliza la función *Lookup*, de más fácil manejo, que veremos un poco más adelante.

Búsquedas utilizando SetKey

Ahora vamos a descomponer la acción de los métodos *FindKey* y *FindNearest* en llamadas a métodos de más bajo nivel. El método fundamental de esta técnica es *SetKey*:

```
procedure TTable.SetKey;
```

El objetivo de este método es muy simple: colocar a la tabla en el estado especial *dsSetKey*. En este estado se permiten asignaciones a los campos de la tabla; estas asignaciones se interpretan como asignaciones a una especie de *buffer* de búsqueda. Por

supuesto, estas asignaciones deben realizarse sobre los campos que componen la clave del índice activo. Una vez que los valores deseados se encuentran en el *buffer* de búsqueda, podemos utilizar uno de los métodos siguientes para localizar un registro:

```
function TTable.GotoKey: Boolean;
procedure TTable.GotoNearest;
```

La correspondencia de estos métodos con *FindKey* y *FindNearest* es evidente. *GotoKey* intentará localizar una fila que corresponda exactamente a la clave especificada, y *GotoNearest* localizará siempre la más cercana. Si deseamos salir del estado *dsSetKey* sin realizar búsqueda alguna, podemos utilizar el método *Cancel* para regresar al estado normal: el estado *dsBrowse*.

Experimentando con SetKey

Para comprender mejor el uso de *SetKey* le propongo un pequeño experimento, no muy útil como técnica de interfaz, pero que aclara el sentido de este método. Para el experimento necesitamos un formulario con una rejilla de exploración. Supongamos además que la tabla visualizada es la tabla de empleados del alias *dbdemos*, la tabla *employee.db*. Los objetos protagonistas son:

Form1: *TForm*	El formulario principal
Table1: *TTable*	La tabla de empleados
DatabaseName *TableName* *IndexName*	*dbdemos* *employee.db* *ByName*
DataSource1: *TDataSource*	Fuente de datos para Table1
DataSet	*Table1*
DBGrid1: *TDbGrid*	La rejilla de exploración
DataSource	*DataSource1*
Button1: *TButton*	Un botón para invocar un diálogo de búsqueda

El índice *ByName* utilizado en la tabla *employee* está definido sobre las columnas *LastName*, el apellido, y *FirstName*, el nombre del empleado. El botón que hemos colocado permitirá invocar a un cuadro de búsqueda. En este segundo formulario colocamos los siguientes objetos:

Form2: *TForm*	El formulario de búsqueda
DbEdit1: *TDbEdit*	Para leer el nombre del empleado a buscar
DataSource *DataField*	*Form1.DataSource1* *FirstName*
DbEdit2: *TDbEdit*	Para leer el apellido del empleado

DataSource *DataField*	*Form1.DataSource1* *LastName*
Button1: *TBitBtn*	Botón para aceptar
Kind	*bkOk*
Button2: *TBitBtn*	Botón para cancelar
Kind	*bkCancel*

Observe que los cuadros de edición están trabajando directamente con la misma tabla que la rejilla del primer formulario; ésta es la parte fundamental del experimento. En Delphi 1 esto cuesta un poco más, porque las asignaciones a las propiedades *DataSource* del segundo formulario hay que hacerlas entonces en tiempo de ejecución. A partir de Delphi 2 basta con arrastrar los campos desde el Editor de Campos de la tabla del formulario principal.

El único código que necesitamos en este ejemplo se produce en respuesta a la pulsación del botón del primer formulario:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   Table1.SetKey;
   if Form2.ShowModal = mrOk then
      Table1.GotoNearest
   else
      Table1.Cancel;
end;
```

Cuando se pulsa el botón la tabla se pone en el estado *dsSetKey* y se invoca al cuadro de búsqueda que hemos creado. Si mueve este cuadro y observa la rejilla verá como desaparecen las filas de la misma temporalmente. Ahora, cualquier cosa que tecleemos en los cuadros de edición del diálogo será considerada como una asignación a un campo de la tabla, aunque en realidad se trata de una asignación a un *buffer* de búsqueda: mientras tecleamos podemos ver también el efecto sobre el contenido de la rejilla. Cuando por fin cerramos el diálogo cancelamos la búsqueda o la disparamos, en dependencia del botón utilizado para terminar el diálogo, y entonces la tabla regresa a la normalidad. Repito, no es una técnica para incluir directamente en un programa, pero es interesante para comprender el mecanismo de búsquedas mediante el índice activo.

¿Por qué existe SetKey?

Además de cualquier motivo filosófico o espiritual que puedan alegar ciertos libros de Delphi que circulan por ahí, *SetKey* existe porque existen los índices de expresiones de dBase; de no ser por esta razón, *SetKey* pudiera quedar como un método interno de la implementación de la VCL. En este tipo de índices, *FindKey* y *FindNearest* se niegan rotundamente a trabajar: el primer paso de estos métodos es "repartir" los

valores de lista pasada como parámetro entre los campos que forman el índice. Pero en un índice de expresiones no es sencillo determinar cuáles son los campos que forman el índice; en realidad es algo que no se intenta en absoluto.

Por ejemplo, tomemos una tabla que tenga los campos *Nombre* y *Apellidos*, y cuyo índice activo sea un índice basado en la expresión *Nombre* + *Apellidos*. Hay dos campos involucrados en el índice, y un programador inocente puede verse tentado a programar algo así:

```
Table1.FindKey(['Howard', 'Lovecraft']); // No funciona
```

¿Cuál es el valor que debe tomarse como nombre y cuál debe tomarse como apellido? Delphi no lo sabe. Y tanto *FindKey* como *FindNearest* están programados para lanzar una excepción si el índice activo es un índice de expresiones. La técnica correcta para realizar una búsqueda sobre este tipo de índices es la siguiente:

```
Table1.SetKey;
Table1['Nombre'] := 'Howard';
Table1['Apellidos'] := 'Lovecraft';
Table1.GotoNearest;
```

Observe que el último método de la secuencia es *GotoNearest*, en vez de *GotoKey*. La causa es la misma: para determinar si encontramos la fila buscada tendríamos que ser capaces de descomponer la expresión del índice en campos, y esto puede no ser posible, en el caso general.

Existe una variante de *SetKey*, el método *EditKey*, que es útil solamente cuando el índice activo, o el criterio de ordenación, incluye varias columnas. *EditKey* coloca la tabla en el estado *dsSetKey*, pero no borra las asignaciones anteriores en el *buffer* de búsqueda. De este modo, después de efectuar la búsqueda del ejemplo anterior podemos ejecutar las siguientes instrucciones para buscar a un tal Howard Duncan:

```
Table1.EditKey;
Table1['Apellidos'] := 'Duncan';
Table1.GotoNearest;
```

Rangos: desde el Alfa a la Omega

Un rango en Delphi es una restricción de las filas visibles de una tabla, mediante la cual se muestran solamente las filas en las cuales los valores de ciertas columnas se encuentran entre dos valores dados. La implementación de este recurso se basa en los índices, por lo cual para poder establecer un rango sobre una o varias columnas debe existir un índice sobre las mismas y estar activo. No podemos definir rangos simultáneamente sobre dos índices diferentes; si tenemos un índice sobre nombres de empleados y otro sobre salarios, no podemos utilizar rangos para pedir las perso-

nas cuyos nombres comienzan con la letra A y que ganen entre tres y cinco millones de pesetas anuales. Eso sí, podemos establecer una de estas restricciones y luego utilizar alguna otra técnica, como los filtros que veremos más adelante, para volver a limitar el resultado.

El índice sobre el cual se define un rango puede ser un índice compuesto, definido sobre varias columnas. Sin embargo, la semántica de la aplicación de rangos sobre índices compuestos es bastante confusa e induce fácilmente a errores. Lo digo por experiencia personal, pues en la edición anterior de este libro di una explicación equivocada de cómo funciona esta operación. El ejemplo concreto que utilicé fue éste: tenemos una tabla *Table1* que está ordenada por un índice compuesto por el nombre y el apellido. ¿Qué hace la siguiente instrucción?

```
Table1.SetRange(['M', 'F'], ['Mzzzz', 'Fzzzz']);
```

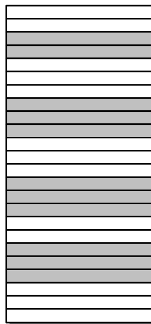
Al parecer, debe limitar el conjunto de registros activos a aquellas personas cuyo nombre comienza con "M" *y* su apellido con "F". ¡No! Ahora aparecen, además de *María Filiberta*, casi todas las personas cuyo nombre comienza con "M" ... sin importar el apellido, al parecer. El problema es que uno espera los registros que satisfacen la condición:

```
'M' <= Nombre and Nombre <= 'Mzzzz' and
'F' <= Apellidos and Apellidos <= 'Fzzzz'
```

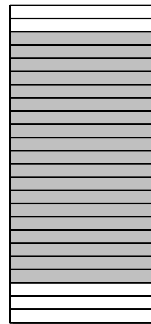
Pero lo que Delphi genera es lo siguiente:

```
(('M' = Nombre and 'F' <= Apellidos) or 'M' < Nombre) and
((Nombre = 'Mzzzz' and Apellidos <= 'Fzzzz') or Nombre <= 'Mzzzz')
```

Es decir, el criterio sobre el apellido solamente se aplica *en los extremos* del rango definido por el criterio establecido sobre el nombre. La siguiente imagen muestra la situación:



Esto es lo que esperamos...



.. y esto es lo que obtenemos...

> Los filtros, que serán estudiados en el siguiente capítulo, nos permiten restringir el conjunto de registros activos utilizando criterios independientes para cada campo.

Lo mismo que sucede con las búsquedas sobre índices activos, sucede con los rangos: existen métodos de alto nivel y de bajo nivel, y la razón es la misma. La forma más fácil de establecer restricciones de rango es utilizar el método *SetRange*, que ya hemos visto en acción.

```
procedure TTable.SetRange(const ValsMin, ValsMax: array of const);
```

La aplicación de este método provoca la actualización inmediata de la tabla. El método inverso a la aplicación de un rango es la cancelación del mismo mediante una llamada a *CancelRange*:

```
procedure TTable.CancelRange;
```

En alguna que otra ocasión puede ser útil la propiedad *KeyExclusive*. Si esta propiedad tiene el valor *True*, los extremos del rango son descartados; normalmente la propiedad vale *False*.

El ejemplo de rangos de casi todos los libros

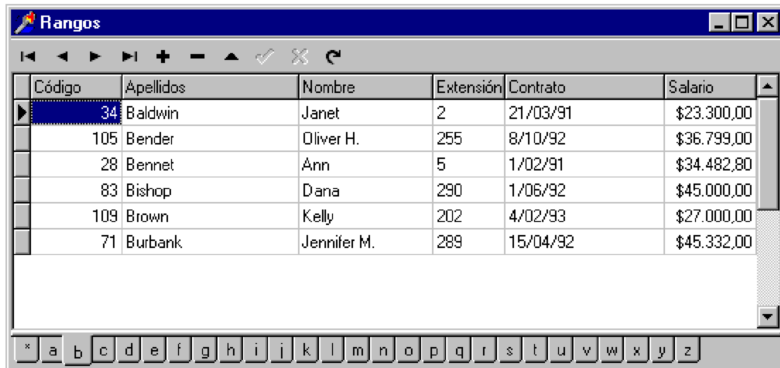
Si el índice activo de una tabla está definido sobre una columna alfanumérica, se puede realizar una sencilla demostración de la técnica de rangos. Reconozco, no obstante, que no es un ejemplo muy original, pues casi todos los libros de Delphi traen alguna variante del mismo, pero no he encontrado todavía algo mejor.

Para esta aplicación utilizaremos un formulario típico de exploración con una rejilla, mediante el cual visualizaremos una tabla ordenada por alguna columna de tipo cadena. He elegido la tabla de clientes de *dbdemos*, *customers.db*, que tiene un índice *ByCompany* para ordenar por nombre de compañía:

Form1: *TForm*	El formulario principal
Table1: *TTable*	La tabla de clientes
DatabaseName *TableName* *IndexName*	*dbdemos* *customer.db* *ByCompany*
DataSource1: *TDataSource*	Fuente de datos para Table1
DataSet	*Table1*
DBGrid1: *TDbGrid*	La rejilla de exploración
DataSource	*DataSource1*

Hasta aquí, lo típico. Ahora necesitamos añadir un nuevo control de tipo *TabControl*, de la página *Win32* de la Paleta, para situar un conjunto de pestañas debajo de la rejilla. En este componente modificamos la propiedad *Tabs*, que determina el conjunto de pestañas del control. Esta propiedad, que es una lista de cadenas de caracteres, debe tener 27 líneas. La primera línea debe contener un asterisco, o la palabra *Todos*, o lo que más le apetezca. Las restantes 26 deben corresponder a las 26 letras del alfabeto (no conozco a nadie que su nombre comience con Ñ, al menos en España): la primera será una A, la segunda una B, y así sucesivamente. Después asigne las siguientes propiedades:

Propiedad	Valor
Align	*alBottom*
TabPosition	*tpBottom*
TabWidth	*18*
Height	*24*



Cuando tenga lista la interfaz de usuario, realice una doble pulsación sobre el control para interceptar su evento *OnChange*:

```
procedure TForm1.TabControl1Change(Sender: TObject);
var
   Letra: string;
begin
   Letra := TabControl1.Tabs[TabControl1.TabIndex];
   if TabControl1.TabIndex = 0 then
   // Si es la pestaña con el asterisco …
   // … mostramos todas las filas.
      Table1.CancelRange
   else
   // Activamos el rango correspondiente
      Table1.SetRange([Letra], [Letra + 'zzz']);
   // Actualizamos los controles asociados
   Table1.Refresh;
end;
```

Aunque la actualización de los controles asociados a la tabla ocurre automáticamente en la mayoría de las situaciones, hay casos en los cuales es imprescindible la llamada al método *Refresh*.

Más problemas con los índices de dBase

dBase nos sigue dando problemas con sus famosos índices de expresiones; nuevamente tenemos dificultades con la asignación automática de valores a campos en el método *SetRange*. Para solucionar este inconveniente, hay que descomponer la llamada a *SetRange* en las funciones de más bajo nivel *SetRangeStart*, *SetRangeEnd* y *ApplyRange*:

```
procedure TTable.SetRangeStart;
procedure TTable.SetRangeEnd;
procedure TTable.ApplyRange;
```

Los métodos *SetRangeStart* y *SetRangeEnd* indican que las asignaciones a campos que se produzcan a continuación especifican, en realidad, los valores mínimo y máximo del rango; el rango se activa al llamar al método *ApplyRange*. El ejemplo del epígrafe anterior puede escribirse de la siguiente forma mediante estos métodos:

```
Table1.SetRangeStart;
Table1['Company'] := Letra;
Table1.SetRangeEnd;
Table1['Company'] := Letra + 'zzz';
Table1.ApplyRange;
```

Como es lógico, si el índice activo para la tabla o el criterio de ordenación establecido en *IndexFieldNames* contemplan varias columnas, hay que realizar varias asignaciones después de *SetRangeStart* y *SetRangeEnd*, una por cada columna del índice o del criterio. Por ejemplo, si *Table1* se refiere a la tabla *employees.db*, y está activo el índice *ByName*, definido sobre los apellidos y el nombre de los empleados, es posible establecer un rango compuesto con los empleados del siguiente modo:

```
Table1.SetRangeStart;
Table1['LastName'] := 'A';
Table1['FirstName'] := 'A';
Table1.SetRangeEnd;
Table1['LastName'] := 'Azzz';
Table1['FirstName'] := 'Azzz';
Table1.ApplyRange;
```

El ejemplo anterior sería, precisamente, lo que tendríamos que hacer si la tabla en cuestión estuviese en formato *dbf* y tuviéramos como índice activo un índice de expresiones definido por la concatenación del apellido y del nombre. Recuerde, no obstante, que este rango compuesto no afecta de forma independiente a los dos campos involucrados, y que desde un punto de vista práctico tiene poca utilidad.

Del mismo modo que contamos con los métodos *SetKey* y *EditKey* para la búsqueda por índices, tenemos también los métodos *EditRangeStart* y *EditRangeEnd*, como variantes de *SetRangeStart* y *SetRangeEnd* cuando el índice está definido sobre varias columnas. Estos dos nuevos métodos permiten la modificación del *buffer* que contiene los valores extremos del rango sin borrar los valores almacenados con anterioridad. En el ejemplo anterior, si quisiéramos modificar el rango de modo que incluya solamente a los apellidos que comienzan con la letra B, dejando intacta la restricción de los nombres que empiezan con A, podríamos utilizar el siguiente código:

```
Table1.EditRangeStart;
Table1['LastName'] := 'B';
Table1.EditRangeEnd;
Table1['LastName'] := 'Bzzz';
Table1.ApplyRange;
```

Cómo crear un índice temporal

Para añadir un nuevo índice a los existentes desde un programa escrito en Delphi, hay que utilizar el método *AddIndex*, del componente *TTable*. Los parámetros de este método son similares a los del método *Add* de la clase *TIndexDefs*:

```
procedure TTable.AddIndex(const Nombre, Definicion: string;
   Opciones: TIndexOptions);
```

La *Definicion*, en la mayoría de los casos, es la lista de campos separados por puntos y comas. Pero si la tabla está en formato dBase, podemos utilizar la opción *ixExpression* en el conjunto de opciones, y especificar una expresión en el parámetro *Definicion*:

```
TablaDBase.AddIndex('PorNombre', 'NOMBRE+APELLIDOS',
   [ixExpression]);
```

Para eliminar el índice recién creado necesitamos el método *DeleteIndex*. Para que este procedimiento pueda cumplir su tarea es necesario que la tabla esté abierta en modo exclusivo. Esta es la declaración de *DeleteIndex*:

```
procedure TTable.DeleteIndex(const Nombre: string);
```

> Aunque *AddIndex* y *DeleteIndex* funcionan para tablas pertenecientes a bases de datos SQL, es preferible utilizar instrucciones SQL lanzadas desde componentes *TQuery* para crear índices desde un programa. Vea el capítulo 26 para más información.

Capítulo

20

Métodos de búsqueda

EN EL CAPÍTULO ANTERIOR estudiamos la búsqueda de valores utilizando índices y el uso de rangos como forma de restringir las filas accesibles de una tabla. En este capítulo estudiaremos los restantes métodos de búsqueda y filtrado que ofrece Delphi. La mayor parte de estos métodos han sido introducidos con Delphi 2, aunque la funcionalidad ya estaba disponible a nivel del BDE para 16 bits. Comenzaremos con los filtros, un método de especificación de subconjuntos de datos, y luego veremos métodos de búsqueda directa similares en cierta forma a *FindKey* y *FindNearest*, pero más generales.

Filtros

Los filtros nos permiten limitar las filas visibles de un conjunto de datos mediante una condición arbitraria establecida por el programador. De cierta manera, son similares a los rangos, pero ofrecen mayor flexibilidad y generalidad, pues no están limitados a condiciones sobre las columnas del índice activo. Cuando se aplican a tablas locales, son menos eficientes, pues necesitan ser evaluados para cada fila del conjunto de datos original. En cambio, rangos y filtros se implementan en cliente/servidor por medio de mecanismos similares, al menos cuando hablamos de filtros definidos por expresiones, y en el caso típico nos ofrecen la misma velocidad.

En Delphi 1, donde no existían filtros, había que utilizar consultas SQL para simular este recurso. En efecto, una tabla filtrada es equivalente a una consulta sobre la misma tabla con la condición del filtro situada en la cláusula **where**. No obstante, en la mayoría de los casos el uso de los filtros es preferible al uso de consultas, por su mayor dinamismo y por evitar los costos añadidos a la creación de cursores.

Es curioso, sin embargo, que el BDE de 16 bits de Delphi 1 permitía la implementación de filtros, a bajo nivel, por supuesto. De todos modos, si es necesario utilizar filtros en Delphi 1, recomiendo el uso de la biblioteca de controles InfoPower, de Woll2Woll, pues la programación de filtros con el BDE es bastante engorrosa.

Existen dos formas principales de establecer un filtro en Delphi. La más general consiste en utilizar el evento *OnFilterRecord*; de esta técnica hablaremos más adelante. La otra forma, más fácil, es hacer uso de la propiedad *Filter*. Estas son las propiedades relacionadas con el uso de filtros:

Propiedad	**Significado**
Filter: **string**	Contiene la condición lógica de filtrado
Filtered: *Boolean*	Indica si el filtro está "activo" o "latente"
FilterOptions: *TFilterOptions*	Opciones de filtrado; las posibles opciones son *foCaseInsensitive* y *foNoPartialCompare*.

La propiedad *Filtered*, de tipo lógico, determina si tiene lugar la selección según el filtro o no; más adelante veremos cómo podemos aprovechar el filtro incluso cuando no está activo.

La condición de selección se asigna, como una cadena de caracteres, en la propiedad *Filter*. La sintaxis de las expresiones que podemos asignar en esta propiedad es bastante sencilla; básicamente, se reduce a comparaciones entre campos y constantes, enlazadas por operadores *and*, *or* y *not*. Por ejemplo, las siguientes expresiones son válidas:

```
Pais = 'Siam'
(Provincia <> '') or (UltimaFactura > '4/07/96')
Salario >= 30000 and Salario <= 100000
```

Si el nombre del campo contiene espacios o caracteres especiales, hay que encerrar el nombre del campo entre corchetes:

```
[Año] = 1776
```

Cuando se trata de tablas de Paradox y dBase, siempre hay que comparar el valor del campo con una constante; no se pueden comparar los valores de dos campos entre sí. En contraste, esta limitación no existe cuando se utilizan tablas pertenecientes a bases de datos cliente/servidor.

Esto no lo dice la documentación...

De no ser por esos "detalles" que se le olvidan a los escritores de manuales, mal lo pasaríamos los escritores de libros. Con el tema de los filtros, al equipo de documentación de Delphi se le quedaron un par de trucos en el tintero. El primero de ellos tiene que ver con la posibilidad de utilizar algo parecido al operador **is null** de SQL en una expresión de filtro. Por ejemplo, si queremos filtrar de la tabla de clien-

tes, *customer.db*, solamente aquellas filas que tengan una segunda línea de dirección no nula, la columna *Addr2*, la expresión apropiada es:

```
Addr2 <> NULL
```

Con la constante *Null* solamente podemos comparar en busca de igualdades y desigualdades. Si, por el contrario, queremos los clientes con la segunda línea de dirección no nula, con toda naturalidad utilizamos esta otra expresión:

```
Addr2 = NULL
```

Recuerde que **null** no es exactamente lo mismo que una cadena vacía, aunque en Paradox se represente de esta manera.

El segundo de los trucos no documentados está relacionado con las búsquedas parciales. Por omisión, Delphi activa las búsquedas parciales dentro de las tablas cuando no se especifica la opción *foNoPartialCompare* dentro de la propiedad *FilterOptions*. Pero si asignamos la siguiente expresión a la propiedad *Filter* de la tabla de clientes, no logramos ninguna fila:

```
Company = 'A'
```

Esto fue lo que se les olvidó aclarar: hay que terminar la constante de cadena con un asterisco. La expresión correcta es la siguiente:

```
Company = 'A*'
```

De esta manera obtenemos todas las compañías cuyos nombres comienzan con la letra *A*. Sin embargo, por razones que explicaré en el capítulo 27, que se ocupa de la comunicación cliente/servidor, prefiero sustituir la expresión anterior por esta otra:

```
Company >= 'A' and Company < 'B'
```

Un ejemplo con filtros rápidos

Es fácil diseñar un mecanismo general para la aplicación de filtros por el usuario de una rejilla de datos. La clave del asunto consiste en restringir el conjunto de datos de acuerdo al valor de la celda seleccionada en la rejilla. Si tenemos seleccionada, 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 todos 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 una tabla simple; si quiere experimentar, le recomiendo conectar la rejilla a la tabla *customer* del alias *dbdemos*, 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 (=)	*miIgual*
Distinto de (<>)	*miDistinto*
Mayor o igual (>=)	*miMayorIgual*
Menor o igual (<=)	*miMenorIgual*
Activar filtro	*miActivarFiltro*
Eliminar filtro	*miEliminarFiltro*

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 TForm1.Filtrar(Sender: TObject);
const
   TiposConComillas: set of TFieldType
   = [ftString, ftDate, ftTime, ftDateTime];
var
   Campo, Operador, Valor: string;
   I: Integer;
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
   Campo := DBGrid1.SelectedField.FieldName;
   // Extraer y dar formato al valor seleccionado
   if DBGrid1.SelectedField.DataType in TiposConComillas then
      Valor := QuotedStr(DBGrid1.SelectedField.AsString)
   else
   begin
      Valor := DBGrid1.SelectedField.AsString;
      for I := 1 to Length(Valor) do
         if Valor[I] = DecimalSeparator then Valor[I] := '.';
   end;
   // Combinar la nueva condición con las anteriores
   if Table1.Filter = '' then
      Table1.Filter := Format('[%s] %s %s',
         [Campo, Operador, Valor])
   else
      Table1.Filter := Format('%s AND [%s] %s %s',
         [Table1.Filter, Campo, Operador, Valor]);
   // Activar directamente el filtro
   miActivarFiltro.Checked := True;
   Table1.Filtered := True;
   Table1.Refresh;
end;
```

El nombre del campo ha sido encerrado automáticamente entre corchetes, para evitar sorpresas con espacios, acentos y eñes. La parte más trabajosa del método es la que tiene que ver con el formato del valor. Si la columna *Company* tiene el valor *Marteens' Diving Academy*, un filtro por igualdad sobre este valor tendría el siguiente aspecto:

```
Company = 'Marteens'' Diving Academy'
```

Tome nota de los apóstrofes repetidos dentro de la constante de cadena; esta es una convención léxica de Pascal. Si, por el contrario, creamos un filtro sobre salarios, no son necesarios los apóstrofes para encerrar el valor:

```
Salary = 100000
```

La función *QuotedStr* nos ayuda a dar formato a una cadena de caracteres, y está definida en la unidad *SysUtils*. También hay que tener cuidado con el formato de los campos numéricos con decimales. Nuestro separador de decimales es la coma, mientras que Pascal espera un punto.

> En Delphi 4, todos estos problemas se resuelven fácilmente, pues se pueden encerrar entre comillas todos los valores constantes, incluidos los numéricos. De este modo, para todos los tipos de datos le daríamos formato a la constante mediante la función *QuotedStr*.

En el método que añade la nueva condición al filtro, se ha activado de paso el filtro, estableciendo la propiedad *Filtered* de la tabla a *True*; de esta forma se obtiene inmediatamente una idea de lo que estamos haciendo. Independientemente de lo anterior, es conveniente tener una opción para activar y desactivar manualmente el filtro, y de esto se encarga el comando de menú *Activar filtro*:

```
procedure TForm1.miActivarFiltroClick(Sender: TObject);
begin
   miActivarFiltro.Checked := not miActivarFiltro.Checked;
   // Activar o desactivar en dependencia …
   // … del estado de la opción del menú.
   Table1.Filtered := miActivarFiltro.Checked;
   Table1.Refresh;
end;
```

Por último, se requiere un comando para eliminar todas las condiciones establecidas:

```
procedure TForm1.miEliminarFiltroClick(Sender: TObject);
begin
   miActivarFiltro.Checked := False;
   Table1.Filtered := False;
   Table1.Filter := '';
   Table1.Refresh;
end;
```

El evento OnFilterRecord

Más posibilidades ofrece la intercepción del evento *OnFilterRecord*. Este es el evento típico que en su encabezamiento tiene un parámetro lógico pasado por referencia, para que aceptemos o rechacemos el registro actual:

```
procedure TForm1.Table1FilterRecord(Sender: TDataSet;
   var Accept: Boolean);
```

El parámetro *Accept* trae por omisión el valor *True*; solamente si queremos rechazar un registro necesitamos asignarle *False* a este parámetro. El algoritmo de decisión que empleemos, por otra parte, puede ser totalmente arbitrario; debemos recordar, sin

embargo, que este evento se disparará para cada fila de la tabla original, por lo cual el algoritmo de filtrado debe ser lo más breve y eficiente posible.

¿Cómo podemos aprovechar este evento? En primer lugar, se puede aprovechar este evento para utilizar relaciones que no sean simples comparaciones. Por ejemplo, quiero seleccionar solamente las compañías que pertenezcan al grupo multinacional Marteens (la fantasía no tributa todavía a Hacienda, ¿no?). Una posible solución es utilizar este manejador de eventos:

```
procedure TForm1.Table1FilterRecord(Sender: TDataSet;
   var Accept: Boolean);
begin
   Accept := Pos('MARTEENS', UpperCase(Sender['Company'])) <> 0;
end;
```

La función *UpperCase* lleva una cadena a mayúsculas y *Pos* busca un subcadena dentro de otra. Este otro ejemplo compara entre sí los prefijos de los números de teléfono y de fax:

```
procedure TForm1.Table1FilterRecord(Sender: TDataSet;
   var Accept: Boolean);
begin
   Accept := Copy(Sender['Phone'], 1, 3) <>
      Copy(Sender['Fax'], 1, 3);
   // ¡Estos tienen el teléfono y el fax en distintas ciudades!
end;
```

También se puede aprovechar el filtro para comparar entre sí dos campos de una tabla Paradox y dBase.

> Medite bien antes de decidirse a utilizar *OnFilterRecord*. Tenga por seguro que este filtro se aplica en el cliente, lo cual implica que la aplicación debe bajarse a través de la red incluso los registros que no satisfacen al filtro. Todo depende, sin embargo, del nivel de selectividad que esperamos de la expresión.

Localización y búsqueda

Si el uso de filtros es similar al uso de rangos, en cierto sentido, los métodos *Locate* y *Lookup* amplían las posibilidades de los métodos de búsqueda con índices. El primero de estos dos métodos localiza la primera fila de una tabla que tenga cierto valor almacenado en una columna, sin importar si existe o no un índice sobre dicha columna. La declaración de este método es:

```
function TDataSet.Locate(const Columnas: string;
   const Valores: Variant; Opciones: TLocateOptions): Boolean;
```

En el primer parámetro se pasan 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
loCaseInsensitive	Ignorar mayúsculas y minúsculas
loPartialKey	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 de la tabla. Si, por el contrario, no se localiza una fila con tales características, las función devuelve *False* y no se altera la posición del cursor sobre la tabla. El algoritmo de búsqueda implementado para *Locate* es capaz de aprovechar los índices existentes. Si el conjunto de columnas no puede aprovechar un índice, *Locate* realiza la búsqueda mediante filtros. Esto es lo que dice la documentación de Inprise/Borland; en el capítulo 27 veremos cómo se implementan realmente estas operaciones en bases de datos SQL.

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 tbClientes.Locate('CustNo', tbPedidos['CustNo'], []) then
   ShowMessage('Se nos ha extraviado 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 tbClientes.Locate('CustNo', 007, []) then
   ShowMessage('… o se lo ha tragado la tierra');
```

Se puede aprovechar este algoritmo para crear un campo calculado sobre la tabla de clientes, que diga si el cliente ha realizado compras o no. Suponiendo que el nuevo campo, *HaComprado*, es de tipo *Boolean*, necesitamos la siguiente respuesta al evento *OnCalcFields* de la tabla *TablaClientes*:

```
procedure TForm1.tbClientesCalcFields(DataSet: TDataSet);
begin
   tbClientes['HaComprado'] := tbPedidos.Locate(
      'CustNo', tbClientes['CustNo'], []);
end;
```

Consideremos ahora que queremos localizar un empleado, dados el nombre y el apellido. La instrucción necesaria es la siguiente:

```
tbEmpleados.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 buscar un valor, 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 tampoco se cambia la fila activa original de la tabla al terminar la ejecución del método.

Por ejemplo, la siguiente función localiza el nombre de un cliente, dado su código:

```
function TDataModule1.ObtenerNombre(Codigo: Integer): string;
begin
   Result := VarToStr(tbClientes.Lookup('Código', Codigo,
      'Compañía'));
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 := tbEmpleados.Lookup('Apellido;Nombre',
      VarArrayOf([Apellido, Nombre]), 'Código');
   if VarIsNull(V) then
      DatabaseError('Empleado no encontrado');
   Result := V;
end;
```

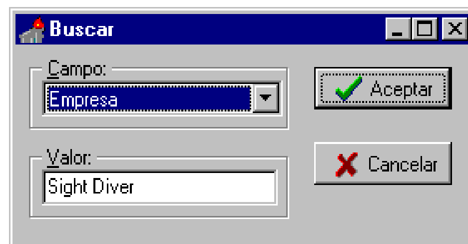
Por variar, he utilizado una excepción para indicar el fallo de la búsqueda; esto equivale a asumir que lo normal es que la función *ObtenerCodigo* deba encontrar el registro del empleado. Note nuevamente el uso de la función *VarArrayOf*, además del uso de *VarIsNull* para controlar la presencia del valor variante 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 := tbEmpleados.Lookup('Codigo', Codigo, 'Nombre;Apellido');
   if VarIsNull(V) then
      DatabaseError('Empleado no encontrado');
   Result := V[0] + ' ' + V[1];
end;
```

Un diálogo genérico de localización

Se puede programar un diálogo general de búsqueda que aproveche el método *Locate* para localizar la primera fila de una tabla que tenga determinado valor en determinada columna. Este diálogo genérico se programa una sola vez y puede utilizarse sobre cualquier tabla en la que se quiera hacer la búsqueda.



Necesitamos un formulario, al que denominaremos *TDlgBusqueda*, con un combo, llamémosle *cbColumnas*, de estilo *csDropDownList*, un cuadro de edición, de nombre *edValor*, y el par típico de botones para aceptar o cancelar el diálogo. Definiremos también, de forma manual, los siguientes métodos y atributos en la declaración de clase del formulario:

```
type
   TDlgBusqueda = class(TForm)
      // …
   private
      FTabla: TTabla;
      procedure AsignarTabla(Valor: TTable);
   protected
      property Tabla: TTabla read FTabla write AsignarTabla;
   public
      function Ejecutar(ATable: TTable): Boolean;
   end;
```

El atributo *FTabla* servirá para recordar la última tabla utilizada para la búsqueda. Para llenar el combo de columnas con los nombres de los campos de la tabla sobre la cual se realiza la búsqueda, se utiliza el método *AsignarTabla*:

```
procedure TDlgBusqueda.AsignarTabla(Valor: TTable);
var
   I: Integer;
begin
   if Valor <> FTabla then
   begin
      cbColumnas.Items.Clear;
      edValor.Text := '';
      for I := 0 to Valor.FieldCount - 1 do
         with Valor.Fields[I] do
            if (FieldKind = fkData)
               and not (DataType in ftNonTextTypes) then
               cbColumnas.Items.AddObject(
                  DisplayLabel, Valor.Fields[I]);
      cbColumnas.ItemIndex := 0;
      FTabla := Valor;
   end;
end;
```

Este método solamente modifica los valores del combo si la tabla parámetro es diferente de la última tabla asignada; así se ahorra tiempo en la preparación de la búsqueda. El algoritmo también verifica que los campos a añadir no sean campos calculados o de referencia, con los que el método *Locate* no trabaja. Por las mismas razones se excluyen los campos BLOB de la lista de columnas; la constante de conjunto *ftNonTextTypes* está definida en la unidad *DB*. Finalmente, se añaden las etiquetas de visualización, *DisplayLabel*, en vez de los nombres originales de campos. Para poder encontrar el campo original sin tener que efectuar una búsqueda de estas etiquetas, la inserción dentro del vector *Items* del combo se realiza mediante el método *AddObject* en vez de *Add*; de esta manera, asociamos a cada elemento del combo el puntero al campo correspondiente.

La búsqueda en sí se realiza en el método *Ejecutar*:

```
function TDlgBusqueda.Ejecutar(ATable: TTable): Boolean;
var
   Campo: TField;
```

```
begin
   Tabla := ATable;
   if ShowModal = mrOk then
   begin
      Campo := cbColumnas.Items.Objects[
         cbColumnas.ItemIndex]);
      Result := FTabla.Locate(Campo.FieldName, edValor.Text, []);
      if not Result then Application.MessageBox(
         'Valor no encontrado',
         'Error', MB_OK + MB_ICONSTOP);
   end
   else
      Result := False;
end;
```

Se asigna la tabla, para llenar si es preciso el combo con los nombres de columnas, y ejecutamos el diálogo, con *ShowModal*. Si el usuario pulsa el botón *Aceptar*, recuperamos primeramente el puntero al campo seleccionado mediante la propiedad *Objects* de la lista de elementos del cuadro de combinación. A partir del nombre de este campo y del valor tecleado en el cuadro de edición, se efectúa la búsqueda mediante el método *Locate*. No he permitido el uso de opciones de búsqueda para no complicar innecesariamente el código de este algoritmo, pero usted puede añadirlas sin mayor dificultad.

Hasta aquí lo relacionado con el diseño y programación del diálogo genérico de búsqueda. En cuanto al uso del mismo, es muy fácil. Suponga que estamos explorando una tabla, *Table1*, sobre una ventana con una rejilla. Colocamos un botón de búsqueda en algún sitio libre de la ventana y programamos la siguiente respuesta a su método *OnClick*:

```
procedure TForm1.bnBusquedaClick(Sender: TObject);
begin
   DlgBusqueda.Ejecutar(Table1);
end;
```

He supuesto que el formulario *DlgBusqueda* es creado automáticamente al crearse el proyecto.

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 reducir el conjunto de datos activo según la condición deseada. La novedad consiste en la posibilidad de, teniendo *Filtered* el valor *False*, 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.

Se puede adaptar el cuadro de diálogo del ejercicio anterior para poder también establecer filtros adecuados a este tipo de búsqueda. Esto lo haremos definiendo un nuevo método, *Buscar*, para establecer el filtro y buscar el primer registro:

```
function TDlgBusqueda.Buscar(ATable: TTable): Boolean;
var
   NumCampo: Integer;
begin
   AsignarTabla(ATable);
   if ShowModal = mrOk then
   begin
      NumCampo := Integer(cbColumnas.Items.Objects[
         cbColumnas.ItemIndex]);
      FTabla.Filter := Format('[%s] = %s',
         [FTabla.Fields[NumCampo].FieldName,
          QuotedStr(edValor.Text]);
      Result := FTabla.FindFirst;
      if not Result then
         Application.MessageBox('Valor no encontrado', 'Error',
            MB_ICONERROR + MB_OK);
   end
   else
      Result := False;
end;
```

Sí, aquí tenemos otra vez a nuestra vieja conocida: la función *QuotedStr*, que utilizamos en el ejemplo de filtros rápidos. La implementación del comando *Buscar siguiente* sería algo así:

```
function TDlgBusqueda.BuscarSiguiente: Boolean;
begin
   Result := FTabla.FindNext;
```

```
    if not Result then
      Application.MessageBox('Valor no encontrado', 'Error',
        MB_ICONERROR + MB_OK);
  end;
```

Por supuesto, para este tipo de búsqueda es preferible habilitar botones en el propio cuadro de diálogo, para lograr el mayor parecido posible con el cuadro estándar de búsqueda de Windows.

Filter By Example

Este es otro ejemplo de cómo diseñar un mecanismo de búsqueda genérico, que aproveche la técnica de los *filtros latentes*. Vamos a crear un prototipo de ventana que pueda ser aprovechada mediante la herencia visual. En esta ventana podremos situar componentes de edición, uno por cada columna por la que queramos buscar. El filtro estará determinado por los valores que introduzcamos en estos campos de edición, de forma similar a lo que ocurre en el lenguaje de consultas *Query By Example*.

Por ejemplo, supongamos que tenemos una tabla de clientes con campos para el código, el nombre y el teléfono. Entonces la ventana de búsqueda tendrá tres editores, uno por cada campo. Si el usuario teclea estos valores:

Código	>34
Nombre	Micro*
Teléfono	

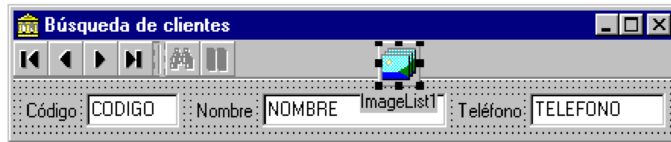
queremos que la expresión de filtro sea la siguiente:

```
Codigo > '34' and Nombre >= 'Micro' and Nombre < 'Micrp'
```

No se ha generado ninguna comparación para el teléfono, porque el usuario no ha tecleado nada en el campo correspondiente. Observe que he evitado el uso del asterisco al final de la constante, para que el filtro sea realmente eficiente.

Creamos una aplicación, y en su ventana principal situamos una rejilla conectada a una tabla arbitraria. Llamaremos *wndMain* a esta ventana principal. Creamos un nuevo formulario en la aplicación, con el nombre *wndSearch*, y lo guardamos en la unidad *Search*. Lo quitamos de la lista de creación automática, pues cuando lo necesitemos lo crearemos nosotros mismos. Le cambiamos la propiedad *BorderStyle* a *bsToolWindow*, y *FormStyle* al valor *fsStayOnTop*, de modo que siempre se encuentre en primer plano. Luego añadimos una barra de herramientas, con cuatro botones de navegación y, un poco separados, un par de botones, para aplicar el filtro de búsqueda (*bnApply*) y para eliminarlo (*bnClean*).

La idea es que, en los descendientes de este formulario, se añadan cuadros de búsquedas del tipo *TEdit*, es decir, comunes y corrientes. Para asociar un campo de la tabla en que se quiere buscar a cada editor, se asignará el nombre del campo en la propiedad *Text* del control. Por ejemplo, la siguiente figura muestra el aspecto de una ventana de búsqueda sobre la tabla de clientes en tiempo de diseño:



Pero no se preocupe por la propiedad *Text* de estos controles, pues durante la creación de la ventana se utilizará para asociar un campo al control, y después se borrará. Esta tarea es responsabilidad del siguiente método de clase público:

```
class function TwndSearch.Launch(AOwner: TForm;
   ATable: TTable): TwndSearch;
var
   I: Integer;
   C: TComponent;
begin
   LockWindowUpdate(Application.MainForm.ClientHandle);
   try
      for I := 0 to Screen.FormCount - 1 do
         if Screen.Forms[I].ClassType = Self then
         begin
            Result := TwndSearch(Screen.Forms[I]);
            Exit;
         end;
      Result := Create(AOwner);
      Result.FTable := ATable;
      for I := 0 to Result.ComponentCount - 1 do
      begin
         C := Result.Components[I];
         if C is TEdit then
         begin
            TEdit(C).Tag :=
               Integer(ATable.FindField(TEdit(C).Text));
            TEdit(C).Text := '';
         end;
      end;
      Result.bnApply.Enabled := False;
   finally
      LockWindowUpdate(0);
   end;
end;
```

Como se puede apreciar, se utiliza el valor guardado en el texto del control para buscar el puntero al campo, el cual se asigna entonces a la propiedad *Tag*. La respuesta a los cuatro botones de navegación es elemental:

```
procedure TwndSearch.bnFirstClick(Sender: TObject);
begin
   FTable.FindFirst;
end;

procedure TwndSearch.bnPriorClick(Sender: TObject);
begin
   FTable.FindPrior;
end;

procedure TwndSearch.bnNextClick(Sender: TObject);
begin
   FTable.FindNext;
end;

procedure TwndSearch.bnLastClick(Sender: TObject);
begin
   FTable.FindLast;
end;
```

También es predecible la respuesta al botón que elimina el filtro:

```
procedure TwndSearch.bnCleanClick(Sender: TObject);
begin
   FTable.Filter := '';
   bnClean.Enabled := False;
   bnApply.Enabled := True;
end;
```

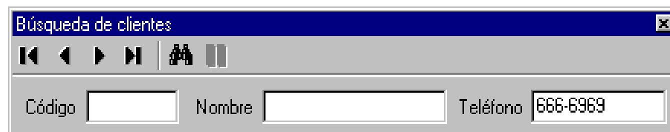
Donde realmente hay que teclear duro es en la respuesta al botón que activa el filtro:

```
procedure TwndSearch.bnApplyClick(Sender: TObject);
var
   F: string;
   I: Integer;
   C: TComponent;
begin
   F := '';
   for I := 0 to ComponentCount - 1 do
   begin
      C := Components[I];
      if C is TEdit then
         AddFilter(F, TEdit(C), TField(TEdit(C).Tag));
   end;
   FTable.Filter := F;
   FTable.FindFirst;
   bnApply.Enabled := False;
   bnClean.Enabled := F <> '';
end;
```

El método *AddFilter* debe haber sido definido en la parte privada de la declaración del formulario, y se encarga de dar el formato correcto al valor tecleado en cada control. Si el campo es de tipo cadena, debe encerrarse el valor entre apóstrofos; si es un número real, hay que sustituir nuestras comas decimales por los puntos americanos:

```
procedure TwndSearch.AddFilter(var F: string; E: TEdit;
   AField: TField);
const
   Ops: array [1..6] of string = ('<>', '<=', '>=', '<', '>', '=');
var
   I: Integer;
   S, S1, Op: string;
begin
   S := Trim(E.Text);
   if (S = '') or (AField = nil) then Exit;
   // Buscar el operador
   Op := '=';
   for I := 1 to 6 do
      if Pos(Ops[I], S) = 1 then
      begin
         Op := Ops[I];
         Delete(S, 1, Length(Op));
         S := TrimLeft(S);
         Break;
      end;
   // Formatear el valor resultante
   if (Op = '=') and (AField.DataType = ftString)
      and (Length(S) > 1) and (S[Length(S)] = '*') then
   begin
      Delete(S, Length(S), 1);
      S1 := S;
      Inc(S1[Length(S1)]);
      S := Format('[%s]>=%s and [%0:s]<%2:s',
         [AField.FieldName, QuotedStr(S), QuotedStr(S1)]);
   end
   else
      S := '[' + AField.FieldName + ']' + Op + QuotedStr(S);
   // Añadir al filtro existente
   if F <> '' then
      AppendStr(F, ' AND ');
   AppendStr(F, S);
end;
```

Ahora derivamos por herencia visual a partir de esta ventana un nuevo formulario, de nombre *TschClientes*, y añadimos los tres cuadros de edición para los tres campos que queremos que participen en la búsqueda. Recuerde quitarlo de la lista de creación automática. El aspecto en ejecución del diálogo de búsqueda de clientes es el siguiente:



Y la instrucción utilizada para lanzarlo, desde la ventana de exploración principal, es la que mostramos a continuación:

```
procedure TwndMain.Buscar1Click(Sender: TObject);
begin
   TschClientes.Launch(Self, modDatos.tbCli);
end;
```

Lo principal es que, teniendo esta plantilla, no hace falta escribir una línea de código en las ventanas de búsqueda que heredan de la misma. Se pueden diseñar estrategias de búsqueda aún más sofisticadas. Por ejemplo, se pueden habilitar teclas para que un campo de edición tome el valor real de la fila activa en la ventana. También se puede hacer que esta ventana se pueda aparcar (*dock*) en la barra de tareas de la ventana principal. Todo eso se lo dejo a su paciencia e imaginación.

Búsqueda en una tabla de detalles

Para terminar, supongamos que la tabla sobre la cual se busca es una tabla de detalles, es decir, una tabla que juega el rol de subordinada en una relación *master/detail*. Quizás deseemos buscar todas las ventas de cierto producto, para mostrar la factura correspondiente; el código del producto debe buscarse en la tabla de detalles, pero las filas visibles de esta tabla están limitadas por el rango implícito en su relación con la tabla de pedidos. No queda más remedio que buscar en otro objeto de tabla, que se refiera a la misma tabla física de pedidos, pero que no participe en una relación *master/detail*. Después tendremos que localizar el pedido correspondiente en la tabla de pedidos, para mostrar la factura completa. Ya sabemos la teoría necesaria para todo esto, y solamente tenemos que coordinar las acciones.

Necesitamos dos ventanas, una para mostrar los pedidos y las líneas de detalles, y otra para seleccionar el artículo a buscar. Esta última es la más sencilla, pues situaremos en la misma un par de botones (*Aceptar* y *Cancelar*), una rejilla sólo para lectura, una fuente de datos y una tabla, *tbArticulos*, que haga referencia a la tabla de artículos *parts.db* de la base de datos *dbdemos*. Llamaremos a este formulario *dlgSeleccion*.

En el formulario principal, llamémosle *wndPrincipal*, situaremos las siguientes tablas:

Tabla	Propósito
tbPedidos	Trabaja con la tabla *orders.db*, del alias *dbdemos*.
tbDetalles	Tabla *items.db*, en relación *master/detail* con la anterior.
tbBusqueda	Tabla *items.db*, pero sin relación *master/detail*.

También necesitaremos rejillas para mostrar las tablas, y un par de botones: *bnBuscar* y *bnSiguiente*, este último con la propiedad *Enabled* a *False*. La respuesta al evento *OnClick* del botón *bnBuscar* es la siguiente:

```
procedure TwndPrincipal.bnBuscarClick(Sender: TObject);
begin
   if dlgSeleccion.ShowModal = mrOk then
   begin
      tbBusqueda.Filter := 'PartNo = ' +
         VarToStr(dlgSeleccion.tbArticulos['PartNo']);
      tbBusqueda.FindFirst;
      bnSiguiente.Enabled := True;
      Sincronizar;
   end;
end;
```

En el método anterior, ejecutamos el diálogo de selección, inicializamos el filtro, buscamos la primera fila que satisface el criterio de búsqueda y sincronizamos la posición de las tablas visibles con el método *Sincronizar*. Este método lo definimos del siguiente modo:

```
procedure TwndPrincipal.Sincronizar;
begin
   if not tbBusqueda.Found then
      bnSiguiente.Enabled := False
   else
   begin
       tbPedidos.Locate('OrderNo', tbBusqueda['OrderNo'], []);
       tbDetalles.GotoCurrent(tbBusqueda);
   end;
end;
```

Se sabe si la última búsqueda tuvo éxito o no consultando la propiedad *Found* de la tabla de búsqueda. En caso afirmativo, se localiza el pedido correspondiente en la tabla de pedidos, y solamente entonces se procede a activar la fila encontrada de la tabla de detalles; observe el uso del método *GotoCurrent* para realizar la sincronización.

La respuesta al botón *bnSiguiente*, teniendo el método anterior programado, es trivial:

```
procedure TwndPrincipal.bnSiguienteClick(Sender: TObject);
begin
   tbBusqueda.FindNext;
   Sincronizar;
end;
```

4

Programación con SQL

- Breve introducción a SQL
- Consultas y modificaciones en SQL
- Procedimientos almacenados y triggers
- Microsoft SQL Server
- Oracle
- Usando SQL con Delphi
- Comunicación cliente/servidor

Parte

Capítulo

21

Breve introducción a SQL

CUANDO IBM DESARROLLÓ EL PRIMER PROTOTIPO DE base de datos relacional, el famoso System R, creó en paralelo un lenguaje de definición y manipulación de datos, llamado QUEL. La versión mejorada de este lenguaje que apareció un poco más tarde se denominó, un poco en broma, SEQUEL. Finalmente, las siglas se quedaron en SQL: *Structured Query Language*, o Lenguaje de Consultas Estructurado. Hay quien sigue pronunciando estas siglas en inglés como *sequel*, es decir, secuela.

La estructura de SQL

Las instrucciones de SQL se pueden agrupar en dos grandes categorías: las instrucciones que trabajan con la estructura de los datos y las instrucciones que trabajan con los datos en sí. Al primer grupo de instrucciones se le denomina también el *Lenguaje de Definición de Datos*, en inglés *Data Definition Language*, con las siglas DDL. Al segundo grupo se le denomina el *Lenguaje de Manipulación de Datos*, en inglés *Data Manipulation Language*, o DML. A veces las instrucciones que modifican el acceso de los usuarios a los objetos de la base de datos, y que en este libro se incluyen en el DDL, se consideran pertenecientes a un tercer conjunto: el *Lenguaje de Control de Datos*, *Data Control Language*, ó DCL.

En estos momentos existen estándares aceptables para estos tres componentes del lenguaje. El primer estándar, realizado por la institución norteamericana ANSI y luego adoptada por la internacional ISO, se terminó de elaborar en 1987. El segundo, que es el que está actualmente en vigor, es del año 1992. En estos momentos está a punto de ser aprobado un tercer estándar, que ya es conocido como SQL-3.

Las condiciones en que se elaboró el estándar de 1987 fueron especiales, pues el mercado de las bases de datos relacionales estaba dominado por unas cuantas compañías, que trataban de imponer sus respectivos dialectos del lenguaje. El acuerdo final dejó solamente las construcciones que eran comunes a todas las implementaciones; de este modo, nadie estaba obligado a reescribir su sistema para no quedarse sin la certificación. También se definieron diferentes niveles de conformidad para

facilitarles las cosas a los fabricantes; si en algún momento alguien le intenta vender un sistema SQL alabando su conformidad con el estándar, y descubre en letra pequeña la aclaración "compatible con el nivel de entrada (*entry-level*)", tenga por seguro que lo están camelando. Este estándar dejó fuera cosas tan necesarias como las definiciones de integridad referencial. Sin embargo, introdujo el denominado *lenguaje de módulos*, una especie de interfaz para desarrollar funciones en SQL que pudieran utilizarse en programas escritos en otros lenguajes.

El estándar del 92 se ocupó de la mayoría de las áreas que quedaron por cubrir en el 87. Sin embargo, no se hizo nada respecto a recursos tales como los procedimientos almacenados, los *triggers* o disparadores, y las excepciones, que permiten la especificación de reglas para mantener la integridad y consistencia desde un enfoque *imperativo*, en contraste con el enfoque *declarativo* utilizado por el DDL, el DCL y el DML. Dedicaremos un capítulo al estudio de estas construcciones del lenguaje. En este preciso momento, cada fabricante tiene su propio dialecto para definir procedimientos almacenados y disparadores. El estándar conocido como SQL-3 se encarga precisamente de unificar el uso de estas construcciones del lenguaje.

En este capítulo solamente nos ocuparemos de los lenguajes de definición y control de datos. El lenguaje de manipulación de datos se trata en el capítulo siguiente. Más adelante nos ocuparemos del lenguaje de *triggers* y procedimientos almacenados.

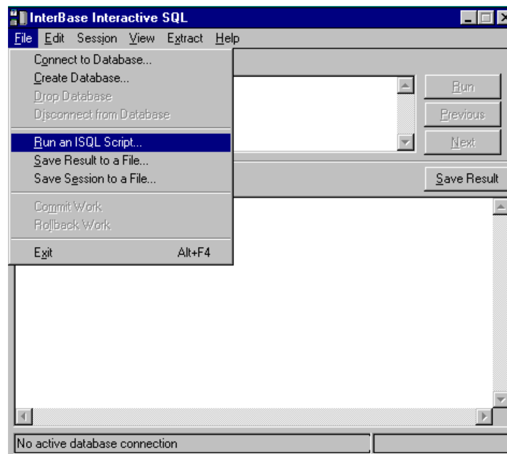
Para seguir los ejemplos de este libro…

Para poder mostrar todas las posibilidades del lenguaje SQL necesitamos un sistema de bases de datos potente, que admita la mayor parte posible de las construcciones sintácticas definidas por el lenguaje. Aunque Paradox y dBase pueden ser abordados mediante SQL, el intérprete ofrecido por el BDE tiene limitaciones, sobre todo en el lenguaje de definición de datos. Por lo tanto, necesitamos algún sistema cliente/servidor para seguir los ejemplos. ¿Cuál de los muchos disponibles? Para este primer capítulo y para el siguiente, casi da lo mismo el sistema elegido, pues las implementaciones de los sublenguajes DDL y DML son bastante similares en casi todos los dialectos existentes de SQL. Donde realmente necesitaremos aclarar con qué dialecto estaremos tratando en cada momento, será en el capítulo sobre *triggers* y procedimientos almacenados.

Así que si usted tiene un Oracle a mano, utilice Oracle; si tiene Informix, utilícelo. ¿Pero qué pasa si nunca ha trabajado con uno de esos sistemas? Pues que siempre tendremos a mano a InterBase, el sistema de bases de datos SQL de Borland. Si el lector tiene instalado, como supongo, cualquiera de las versiones de Delphi 1, o las versiones cliente/servidor o Developer de Delphi 2 (la versión Desktop no vale), tendrá también instalado el servidor local de InterBase. Si está utilizando Delphi 3 ó 4, debe realizar la instalación por separado. En cualquier caso, tomaré como punto de

partida a InterBase para los primeros tres capítulos. En estos capítulos iniciales, presentaré las características fundamentales del lenguaje SQL: el lenguaje de definición de datos en el primer capítulo, en el siguiente, el lenguaje de consultas y manipulación de datos, y para finalizar, las extensiones procedimentales para *triggers* y procedimientos almacenados. Luego dedicaré un par de capítulo a las particularidades de Oracle y MS SQL Server.

InterBase viene acompañado por la aplicación Windows ISQL. Con esta utilidad podemos crear y borrar bases de datos de InterBase, conectarnos a bases de datos existentes y ejecutar todo tipo de instrucciones SQL sobre ellas. Las instrucciones del lenguaje de manipulación de datos, y algunas del lenguaje de definición, pueden ejecutarse directamente tecleando en un cuadro de edición multilíneas y pulsando un botón. El resultado de la ejecución aparece en un control de visualización situado en la parte inferior de la ventana.



Para las instrucciones DDL más complejas y la gestión de procedimientos, disparadores, dominios y excepciones es preferible utilizar *scripts* SQL. Un *script* es un fichero, por lo general de extensión SQL, que contiene una lista de instrucciones arbitrarias de este lenguaje separadas entre sí por puntos y comas. Este fichero debe ejecutarse mediante el comando de menú *File|Run an ISQL Script*. Las instrucciones SQL se van ejecutando secuencialmente, según el orden en que se han escrito. Por omisión, los resultados de la ejecución del *script* también aparecen en el control de visualización de la ventana de Windows ISQL.

Por supuesto, también puede utilizarse la utilidad SQL Explorer del propio Delphi para ejecutar instrucciones individuales sobre cualquier base de datos a la que deseemos conectarnos. También podemos utilizar Database Desktop si queremos realizar pruebas con Paradox y dBase.

La creación y conexión a la base de datos

Estamos en InterBase, utilizando Windows ISQL. ¿Cómo nos conectamos a una base de datos de InterBase para comenzar a trabajar? Basta con activar el comando de menú *File|Connect to database*. Si ha instalado InterBase en un servidor remoto y tiene los SQL Links que vienen con Delphi cliente/servidor, puede elegir la posibilidad de conectarse a ese servidor remoto. En cualquier caso, puede elegir el servidor local. Cuando especificamos un servidor remoto, tenemos que teclear el nombre completo del fichero de base de datos en el servidor; no necesitamos tener acceso al fichero desde el sistema operativo, pues es el servidor de InterBase el que nos garantiza el acceso al mismo. Si estamos utilizando el servidor local, las cosas son más fáciles, pues contamos con un botón *Browse* para explorar el disco. Las bases de datos de InterBase se sitúan por lo general en un único fichero de extensión *gdb*; existe, no obstante, la posibilidad de distribuir información en ficheros secundarios, lo cual puede ser útil en servidores con varios discos duros. Los otros datos que tenemos que suministrar a la conexión son el nombre de usuario y la contraseña. El nombre de usuario inicial en InterBase es, por omisión, *SYSDBA*, y su contraseña es *masterkey*. Respete las mayúsculas y minúsculas, por favor. Una vez que acepte el cuadro de diálogo, se intentará la conexión. En cualquier momento de la sesión podemos saber a qué base de datos estamos conectados mirando la barra de estado de la ventana.

El mismo mecanismo puede utilizarse para crear una base de datos interactivamente. La diferencia está en que debemos utilizar el comando *File|Create database*. Sin embargo, necesitamos saber también cómo podemos crear una base de datos y establecer una conexión utilizando instrucciones SQL. La razón es que todo *script* SQL debe comenzar con una instrucción de creación de bases de datos o de conexión. La más sencilla de estas instrucciones es la de conexión:

```
connect "C:\Marteens\IntrBase\Examples\Prueba.GDB"
user "SYSDBA" password "masterkey";
```

Por supuesto, el fichero mencionado en la instrucción debe existir, y el nombre de usuario y contraseña deben ser válidos. Observe el punto y coma al final, para separar esta instrucción de la próxima en el *script*.

La instrucción necesaria para crear una base de datos tiene una sintaxis similar. El siguiente ejemplo muestra el ejemplo más común de creación:

```
create database "C:\Marteens\IntrBase\Examples\Prueba.GDB"
user "SYSDBA" password "masterkey"
page_size 2048;
```

InterBase, y casi todos los sistemas SQL, almacenan los registros de las tablas en bloques de longitud fija, conocidos como *páginas*. En la instrucción anterior estamos

redefiniendo el tamaño de las páginas de la base de datos. Por omisión, InterBase utiliza páginas de 1024 bytes. 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. 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.

Tipos de datos en SQL

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:

Tipo de dato	Tamaño	Observaciones
char(*n*), varchar(*n*)	*n* bytes	Longitud fija; longitud variable
integer, int	32 bits	
smallint	16 bits	
float	4 bytes	Equivale al tipo *Single* de Delphi
double precision	8 bytes	Equivale al *Double* de Delphi

Tipo de dato	Tamaño	Observaciones
numeric(*prec*, *esc*)	*prec*=1-15, *esc* <= *prec*	Variante "exacta" de *decimal*
decimal(*prec*, *esc*)	*prec*=1-15, *esc* <= *prec*	
date	64 bits	Almacena la fecha y la hora
blob	No hay límite	

La diferencia fundamental respecto al estándar SQL tiene que ver con el tipo **date**. SQL estándar ofrece los tipos **date**, **time** y **timestamp**, para representar fecha, hora y la combinación de fecha y hora. El tipo **date** de InterBase corresponde al tipo **timestamp** del SQL estándar.

El tipo de dato **blob** (*Binary Large OBject* = Objeto Binario Grande) se utiliza para almacenar información de longitud variable, generalmente de gran tamaño. En principio, a InterBase no le preocupa qué formato tiene la información almacenada. Pero para cada 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
```

Una de las peculiaridades de InterBase como gestor de bases 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. El estudio de este recurso, sin embargo, está más allá de los objetivos de este libro.

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
Clientes	Los clientes de nuestra empresa
Empleados	Los empleados que reciben los pedidos
Articulos	Las cosas que intentamos vender
Pedidos	Pedidos realizados
Detalles	Una fila por cada artículo vendido

Un ejemplo similar lo encontramos en la base de datos de demostración que trae Delphi, pero en formato Paradox. Utilizaremos esas tablas más adelante para demostrar algunas características de este formato de bases de datos.

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 profun-

didad 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 del 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 "Marteens Corporation"=
               (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. 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,
        NIF     varchar(9) not null,
        /* … */
        primary key (Codigo),
        unique (NIF)
);
```

Por cada clave primaria o alternativa definida, InterBase crea un índice único para mantener la restricción. Este índice se bautiza según el patrón *rdb$primaryN*, donde *N* es un número serial.

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 keys*). 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 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 (incluyendo la versión 7) 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 5 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 *rdb$foreignN*, donde *N* es un número serial.

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 Pascal. 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 otras tablas.

> Es aconsejable definir dominios en InterBase por una razón adicional: el Diccionario de Datos de Delphi los reconoce y asocia automáticamente a conjuntos de atributos (*attribute sets*). De esta forma, 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.

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;
```

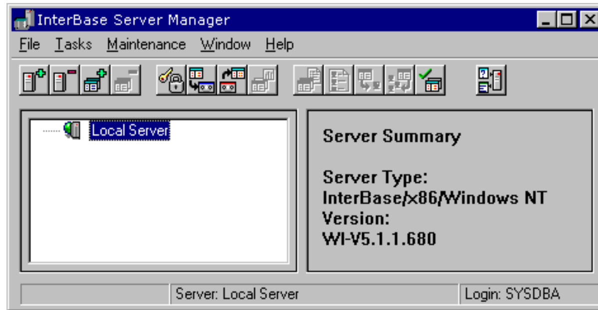
Creación de vistas

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*.

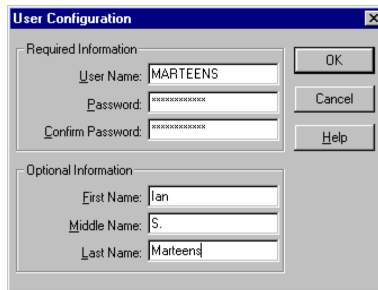
Como todavía no conocemos el lenguaje de consultas, que nos permite especificar las operaciones sobre tablas, postergaremos el estudio de las vistas para más adelante.

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. La gestión de los nombres de usuarios y sus contraseñas se realiza mediante la utilidad *Server Manager*.



Dentro de esta aplicación, hay que ejecutar el comando de menú *Tasks|User security*, para llegar al diálogo con el que podemos añadir, modificar o eliminar usuarios. La siguiente imagen muestra el diálogo de creación de nuevos usuarios:



El nombre del usuario *SYSDBA* no puede cambiarse, pero es casi una obligación cambiar su contraseña en cuanto termina 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.

El sistema de seguridad explicado tiene un par de aparentes "fallos". En primer lugar, 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.

En realidad, el fallo consiste en permitir que cualquier mequetrefe pueda acceder a nuestras apreciadas bases de datos. Así que, antes de planear la protección del sistema de gestión de base de datos (ya sea InterBase o cualquier otro), ocúpese de controlar el acceso al servidor de la gente indeseable.

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
select	Lectura de datos
update	Modificación de datos existentes
insert	Creación de nuevos registros
delete	Eliminación de registros
all	Los cuatro privilegios anteriores
execute	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 destinatarios 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. Estas instrucciones se estudiarán en el momento adecuado.

Roles

Los roles 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.

Primero necesitamos crear los roles 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, en vez de a los usuarios directamente, como hacíamos antes. Observe que InterBase sigue permitiendo ambos tipos de permisos:

```
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 roles, 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 rol de domador a otros usuarios. De este modo, siempre habrá quien se ocupe de los animales cuando me ausente del circo por vacaciones.

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         int 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         int not null,
       Apellidos      varchar(20) not null,
       Nombre         varchar(15) not null,
       FechaContrato  date default "Now",
       Salario        int,
       NombreCompleto computed by (Nombre || " " || Apellidos),

       primary key    (Codigo)
);

create table Articulos (
       Codigo         int not null,
       Descripcion    varchar(30) not null,
       Existencias    int default 0,
       Pedidos        int default 0,
       Costo          int,
       PVP            int,

       primary key    (Codigo)
);

create table Pedidos (
       Numero         int not null,
       RefCliente     int not null,
```

```
        RefEmpleado    int,
        FechaVenta     date default "Now",
        Total          int default 0,

        primary key    (Numero),
        foreign key    (RefCliente) references Clientes (Codigo)
                       on delete no action on update cascade
);

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 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 *****/
```

Capítulo

22

Consultas y modificaciones en SQL

DESDE 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 PCs 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 este mismo capítulo 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 subdirectorio *demos\data*, a partir del directorio de instalación de Delphi. Estas tablas también se encuentran en formato Paradox, en el mismo subdirectorio. Puede utilizar el programa *Database Desktop* para probar el uso de SQL sobre tablas Paradox y dBase. Sin embargo, trataré de no tocar las peculiaridades del Motor de SQL Local ahora, dejando esto para el capítulo 26, que explica cómo utilizar SQL desde Delphi.

La instrucción select: el lenguaje de consultas

A grandes rasgos, la estructura de la instrucción **select** es la siguiente:

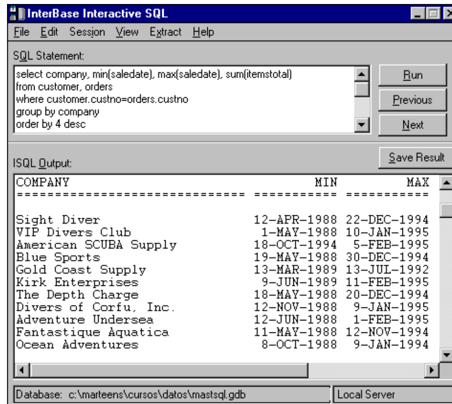
```
select [distinct] lista-de-expresiones
from lista-de-tablas
[where condición-de-selección]
[group by lista-de-columnas]
[having condición-de-selección-de-grupos]
[order by lista-de-columnas]
[union instrucción-de-selección]
```

¿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 dejar. 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".



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 Hawai. 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 Pascal. Sin embargo, 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 com-

portamiento 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 **like** '% Ocean'	'Pacific Ocean'	'Ocean Paradise'
Fruta **like** '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 tablas locales de Delphi acepta las funciones **upper**, **lower**, **trim** y **substring** de SQL estándar. 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 36 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 (¡aquellos 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
Macarrones a la boloñesa
Judías verdes con jamón
Crema de champiñones

Segundo plato
Escalope a la milanesa
Pollo a la parrilla
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 milanesa
Macarrones a la boloñesa	Pollo a la parrilla
Macarrones a la boloñesa	Chuletas de cordero
Judías verdes con jamón	Escalope a la milanesa
Judías verdes con jamón	Pollo a la parrilla
Judías verdes con jamón	Chuletas de cordero
Crema de champiñones	Escalope a la milanesa
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 un café solo 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**, hubiéramos obtenido un resultado de 473550 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
```

No se puede 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 *SalesDate* al resultado y modificar la cláusula de ordenación del siguiente modo:

```
select Company, ItemsTotal, SalesDate, 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
```

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.

Funciones de conjuntos

Existen cinco funciones de conjuntos en SQL, conocidas en inglés como *aggregate functions*. Estas funciones son:

Función	Significado
count	Cantidad de valores no nulos en el grupo
min	El valor mínimo de la columna dentro del grupo
max	El valor máximo de la columna dentro del grupo
sum	La suma de los valores de la columna dentro del grupo
avg	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.

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. Esto 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. Los desarrolladores que estén programando aplicaciones con Delphi 1 para Windows 3.1 y Paradox o dBase deben tener en cuenta esta restricción.

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 muestran 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

CustNo	*Company*
1510	Ocean Paradise
1666	Marteens' Diving Academy

Orders

OrderNo	*CustNo*
1025	1510
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 de Delphi, 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**.

> El encuentro externo por la izquierda puede escribirse en Oracle de esta forma alternativa:
>
> ```
> select *
> from   Customer, Orders
> where  Customer.CustNo (+) = Orders.CustNo
>   and  Orders.OrderNo is null
> ```

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
```

Esta segunda variante no estuvo disponible en el intérprete de SQL local hasta la versión 4 del BDE. 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
```

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 Hawai:

```
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.

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, ¿pudiéramos insertar dentro de la vista *Hawaianos* una empresa con sede social en la ciudad costera de Cantalapiedra[20]? Si permitiésemos esto, después de la inserción 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.

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 situa-

[20] 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 cantalapedrenses, pues me parece que viven en ciudad sin costas.

ciones como las descritas, en que un registro recién insertado o modificado desaparece misteriosamente por no pertenecer a la vista.

Capítulo

23

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 son bastante diferentes entre sí, 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 (asumiendo que tiene Delphi). 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 implementa 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 en sintaxis, 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 infor-

mación. 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 centralizamos 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 Delphi tenemos el componente *TStoredProc*, diseñado para la ejecución de estos procedimientos. En un capítulo posterior veremos cómo suministrar parámetros, ejecutar procedimientos y recibir información utilizando este componente.

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 *Minimo* y *Aumento* son variables o parámetros, puede ejecutarse la siguiente instrucción:

```
update Empleados
set    Salario = Salario * :Aumento
where  Salario < :Minimo;
```

  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  Codigo = 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 procedimientos 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 tiene 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; /* ¡¡¡ 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 este 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...do**, 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[21]. 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

[21] Realmente, es el BDE quien configura a InterBase de este modo.

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 un operador **first** para este propósito, 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 ^
```

Entonces podremos realizar consultas como la siguiente:

```
select *
from   MejoresClientes(10)
```

Triggers, o disparadores

Una de las posibilidades 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
NombreTrigger	El nombre que se le va a asignar al *trigger*
Tabla	El nombre de la tabla a la cual está asociado
active | **inactive**	Puede crearse inactivo, y activarse después
before | **after**	Se activa antes o después de la operación
delete | **insert** | **update**	Qué operación provoca el disparo del *trigger*
position	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 llamar a este algoritmo 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 implementado 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.

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 tienen problemas cuando se genera una clave primaria para una fila mediante un generador, pues el registro recién insertado "desaparece" según el punto de vista de la tabla. Para no tener que abandonar los generadores, la solución 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.ParaByName('COD').AsInteger;
end;
```

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.

Simulando la integridad referencial

Como hemos explicado, mediante los *triggers* y los procedimientos almacenados podemos expresar reglas de consistencia en forma *imperativa*, en contraste con las reglas *declarativas* que se enuncian al crear tablas: claves primarias, alternativas y externas, condiciones de verificación, etc. En general, es preferible utilizar una regla declarativa antes que su equivalente imperativo. Pero sucede que las posibilidades de las reglas declarativas son más limitadas que las posibilidades de las reglas imperativas.

En InterBase 4, por ejemplo, las restricciones de integridad referencial no admiten modificaciones ni borrados en las tablas maestras de una clave externa. Sin embargo, a veces es deseable permitir estas operaciones y propagar los cambios en cascada a las tablas dependientes.

Ilustraré la forma de lograr restricciones de integridad referencial con propagación de cambios mediante el ejemplo de la tabla de pedidos y líneas de detalles. Recordemos la definición de la tabla de pedidos, en el capítulo sobre el Lenguaje de Definición de Datos:

```
create table Pedidos (
       Numero         int not null,
       RefCliente     int not null,
       RefEmpleado    int,
       FechaVenta     date default "Now",
       Total          int default 0,
```

```
        primary key (Numero),
        foreign key (RefCliente) references Clientes (Codigo)
);
```

La definición de la tabla de detalles cambia ahora, sustituyéndose la cláusula **foreign key** que hacía referencia a la tabla de pedidos:

```
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 100),

        primary key (RefPedido, NumLinea),
        foreign key (RefArticulo) references Articulos (Codigo),
        /*
          Antes: foreign key(RefPedido) references Pedidos(Numero)
        */
        check (RefPedido in (select Numero from Pedidos))
);
```

La nueva cláusula **check** verifica en cada inserción y modificación que no se introduzca un número de pedido inexistente. El borrado en cascada se puede lograr de la siguiente manera:

```
create trigger BorrarDetallesEnCascada for Pedidos
        active after delete
        position 0
as
begin
        delete from Detalles
        where  RefPedido = old.Numero;
end ^
```

Un poco más larga es la implementación de actualizaciones en cascada.

```
create trigger ModificarDetallesEnCascada for Pedidos
        active after update
        position 0
as
begin
        if (old.Numero <> new.Numero) then
                update Detalles
                set    RefPedido = new.Numero
                where  RefPedido = old.Numero;
end ^
```

Por supuesto, los *triggers* hubieran sido mucho más complicados si hubiéramos mantenido la restricción **foreign key** en la declaración de la tabla de detalles, en particular, la propagación de modificaciones.

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 el 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 Marteens 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 alerters*) 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, en este estilo:

```
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
Events	Propiedad	Los nombres de eventos que nos interesan.
Registered	Propiedad	Debe ser *True* para notificar, en tiempo de diseño, nuestro interés en los eventos almacenados en la propiedad anterior.
Database	Propiedad	La base de datos a la cual nos conectaremos.

Nombre	Tipo	Propósito
RegisterEvents	Método	Notifica a la base de datos nuestro interés por los eventos de la propiedad *Events*.
UnRegisterEvents	Método	El método inverso del método anterior.
OnEventAlert	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 *OnEventAlert* similar a éste:

```
procedure TForm1.IBEventAlerter1EventAlert(Sender: TObject;
   EventName: string; EventCount: Longint;
   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.

Capítulo

24

Microsoft SQL Server

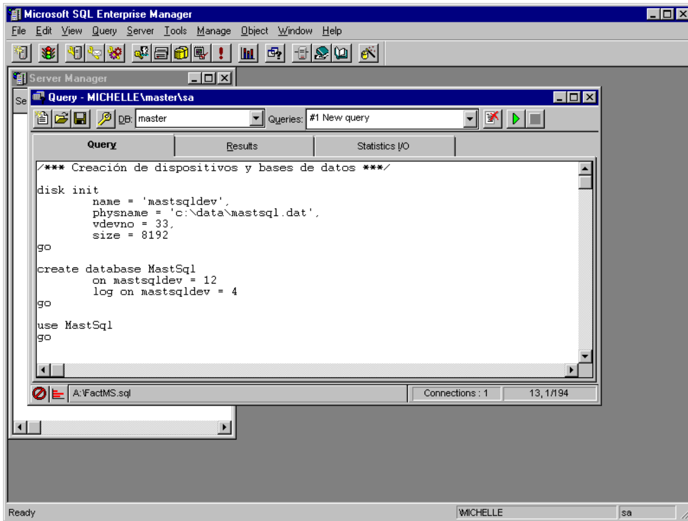
EN ESTE CAPÍTULO ANALIZAREMOS LAS CARACTERÍSTICAS generales de la implementación de SQL por parte de Microsoft SQL Server. Este sistema es uno de los más extendidos en el mundo de las redes basadas en Windows NT. No está, sin embargo, dentro de la lista de mis sistemas favoritos, por dos razones. La primera: una arquitectura bastante pobre, con tamaño fijo de página, bloqueos a nivel de página que disminuyen la capacidad de modificación concurrente, ficheros de datos y de transacciones de tamaño fijo... La segunda razón: MS SQL Server tiene uno de los dialectos de SQL más retorcidos y horribles del mundo de los servidores de datos relacionales, dudoso privilegio que comparte con Sybase.

Es muy probable que la primera razón deje de ser tal con la próxima aparición de la versión 7 de este sistema de bases de datos, que mejora bastante la arquitectura física utilizada. Este capítulo está basado fundamentalmente en la versión actual, la 6.5. En el momento de la escritura de este libro, la versión 7 estaba en fase beta. Trataré en lo posible de adelantar algunas características de la misma, pero advierto al lector de los peligros de deducir características de la versión final a partir de una beta.

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 dispositivos (más adelante veremos qué son), bases de datos, crear usuarios y administrar sus 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.

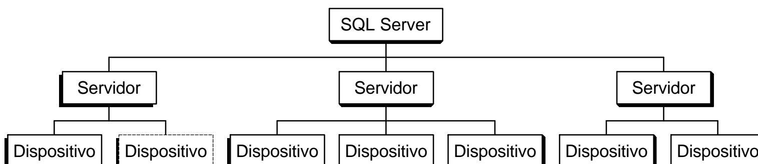


Hay que reconocer que las herramientas de administración de MS SQL Server clasifican entre las amigables con el usuario. Esto se acentúa en la versión 7, en la cual pegas una patada al servidor y saltan cinco o seis asistentes (*wizards*) para adivinar qué es lo que quieres realmente hacer.

Creación de bases de datos con MS SQL Server

Uno de los cambios que introduce la versión 7 de SQL Server es precisamente en la forma de crear bases de datos. En esta sección, sin embargo, describiré fundamentalmente la técnica empleada hasta la versión 6.5, pues pueden aparecer novedades en la versión final. Además, las bases de datos de Sybase se crean con mecanismos similares.

Para trabajar con SQL Server hay que comprender qué es un *dispositivo* (*device*). Muy fácil: es un fichero del sistema operativo. ¿Entonces por qué inventar un nombre diferente? La razón está en la prehistoria de este producto, a cargo de Sybase. A diferencia de SQL Server, el servidor de Sybase puede ejecutarse en distintos sistemas operativos, y puede que, en determinado sistema, un fichero físico no sea el soporte más adecuado para el almacenamiento de una base de datos.



Los dispositivos pertenecen al servidor en que se crean. Para crear un dispositivo hay que indicar el nombre del fichero, el tamaño inicial, un nombre lógico y un número entero único, entre 0 y 255, que servirá internamente como identificación. Pueden crearse dispositivos en cualquiera de los discos locales del servidor. La sintaxis completa de la instrucción de creación de dispositivos en SQL Server es la siguiente:

```
disk init
        name='nombre_lógico',
        physname='nombre_de_fichero',
        vdevno=número_de_dispositivo_virtual,
        size=número_de_bloques
        [, vstart=dirección_virtual]
```

Por ejemplo, la siguiente instrucción crea un dispositivo basado en el fichero *midisp.dat*, con 20 Mb de espacio:

```
disk init name='MiDisp',
        physname='c:\datos\midisp.dat',
        vdevno=33,
        size=10000
```

Hay que tener en cuenta que el tamaño de bloque es siempre el mismo: 2048 bytes. En SQL Server 7 este tamaño aumenta a 8192 bytes, pero siempre es constante. Otro problema relacionado con los dispositivos consiste en que no aumentan su tamaño por demanda. Si una base de datos situada en uno de estos dispositivos necesita más memoria, hay que ampliar la capacidad del dispositivo de forma manual. Por supuesto, siempre podemos contratar a un "administrador de bases de datos" que se ocupe del asunto, y de esa forma ayudamos a reducir las cifras del paro.

Ahora podemos crear las bases de datos, mediante la siguiente instrucción:

```
create database nombre_base_de_datos
        [on {default|dispositivo} [=tamaño]
                [, dispositivo [=tamaño]]...]
        [log on dispositivo [=tamaño]
                [, dispositivo [=tamaño]]...]
        [for load]
```

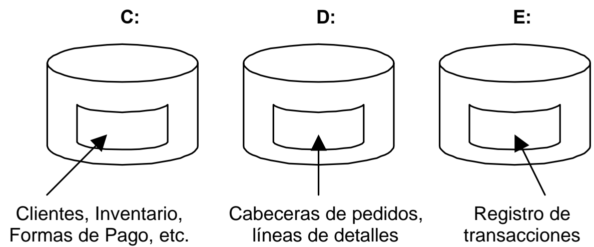
En este caso, el tamaño siempre se especifica en megabytes, como en la siguiente instrucción:

```
create database Facturas on MiDisp=20, log on MiLog=5
```

La base de datos *Facturas* tendrá sus datos en el dispositivo *MiDisp*, mientras que la información sobre transacciones pendientes se almacenará en el segmento *MiLog*. ¿Qué información es esta? La mayoría de los sistemas de gestión de bases de datos, con la excepción de InterBase, implementan la atomicidad de las transacciones llevando cuenta de los cambios efectuados durante una transacción en un *log file*, o *re-*

gistro de transacciones. Si cancelamos una transacción, la información almacenada en este fichero se utiliza para anular las inserciones, borrados y modificaciones realizadas. Es conveniente que el registro de transacciones y los datos residan en dispositivos diferentes, y de ser posible, que estén situados en discos diferentes. De este modo, se puede aprovechar la concurrencia ofrecida por el uso de distintos controladores físicos de discos, a nivel de hardware.

SQL Server permite incluso definir varios dispositivos para almacenar datos y para el registro de transacciones. Utilizando instrucciones como *sp_addsegment* y *sp_placeobject*, podemos almacenar distintas tablas en diferentes discos, como se muestra en el siguiente esquema:



Recuerde que el objetivo principal de la segmentación es el aumentar la eficiencia de la base de datos.

> La principal novedad de SQL Server 7 en cuanto a arquitectura es que desaparece el concepto de dispositivo. Ahora las bases de datos se definen sobre ficheros físicos. El fichero de datos principal tiene la extensión *mdf*, el fichero *log* tiene la extensión *ldf*, y los ficheros de datos secundarios, *ndf*. Los nuevos ficheros pueden crecer por demanda, como sucede en los sistemas de bases de datos "serios". Ya he mencionado, además, que el tamaño de página se aumenta a 8192, aunque este valor no puede modificarse.

Tipos de datos predefinidos

Los tipos de datos soportados por SQL Server son los siguientes:

Tipo de datos	Implementación de SQL Server
Binario	*binary[(n)], varbinary[(n)]*
Caracteres	*char[(n)], varchar[(n)]*
Fecha y hora	*datetime, smalldatetime*
Numérico exacto	*decimal[(p[, s])], numeric[(p[, s])]*
Numérico aproximado	*float[(n)], real*

Tipo de datos	Implementación de SQL Server
Enteros	*int, smallint, tinyint*
Moneda	*money, smallmoney*
Especiales	*bit, timestamp*
Texto e imágenes	*text, image*

Nos encontramos aquí con el mismo problema que en InterBase: el tipo *datetime* representa simultáneamente fechas y horas. Curiosamente, el tipo *timestamp* no se refiere a fechas, sino que es un entero de 8 bytes que se actualiza automáticamente cuando se crea o se modifica una fila. El tipo *smalldatetime* tiene menos precisión que *datetime*. Almacena la fecha como el número de días a partir del 1 de enero del 1900, y la hora como el número de minutos a partir de media noche. Por lo tanto, solamente le será útil si su programa debe quedar fuera de circulación antes del 6 de junio del 2079. Si es usted uno de los Inmortales, o le preocupa el futuro del Planeta, no lo utilice. Del mismo modo, *smallmoney* tiene un rango que va desde -214.748,3648 hasta aproximadamente el mismo número positivo. Ya sean euros o dólares, es más dinero del que nunca he tenido.

> Aunque el tipo *varchar* de SQL Server permite almacenar cadenas de caracteres de longitud variable, los registros que contienen estos campos siguen ocupando un tamaño fijo, al menos hasta la versión 6.5. Al parecer, la versión 7 corrige 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_binddefault* para asociar valores por omisión a tipos de datos creados por el programador.

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
```

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 este 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 MS 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. Sin embargo, no es una opción recomendable, a mi entender, si realizamos inserciones desde Delphi en esa tabla y estamos mostrando sus datos en pantalla. Delphi tiene problemas para releer registros cuando su clave primaria se asigna o se modifica en el servidor, como en este caso. Sucede lo mismo que con los generadores de InterBase y las secuencias de Oracle.

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]')
)
```

Una opción interesante de SQL Server es la posibilidad de crear tablas temporales. Para esto, basta con añadir uno de los prefijos # ó ## al nombre de la tabla. En el primer caso, la tabla temporal solamente será visible para el usuario que la crea, y será eliminada en cuanto el usuario cierre su conexión activa. En el segundo caso, se trata de una tabla temporal global, que desaparecerá en cuanto el último usuario que la esté usando se desconecte.

Integridad referencial

Las versiones de este producto anteriores a la 7 realizan una pobre implementación de las restricciones de integridad referencial. En contraste, y de acuerdo a la información suministrada en la beta, la versión 7 realiza una pobre implementación de las restricciones de integridad referencial. Es decir, que al parecer todo cambia para seguir igual.

¿De qué me estoy quejando? Principalmente, de que SQL Server no define ningún tipo de acción referencial, aparte de la prohibición de borrados y de actualizaciones en la clave de la fila maestra. Y, para agravar el problema, tenemos la complicación adicional del tipo único de *triggers* permitido por el sistema, que se dispara *después* de la operación. De este modo, la simple adición de una propagación de borrados en cascada requiere que la restricción de integridad referencial se anule y se implemente totalmente "a mano". Más adelante veremos cómo.

MS SQL Server no define automáticamente los índices secundarios necesarios en la tabla dependiente. El programador debe crearlos explícitamente. Además, la única acción referencial permitida es la prohibición de borrados y actualizaciones.

Indices

La sintaxis para la creación de índices en MS SQL Server es básicamente la siguiente:

```
create [unique] [clustered | nonclustered] index Indice
       on Tabla (Columna [, Columna …])
```

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.

Seguridad en MS SQL Server

SQL Server implementa un sistema de seguridad en dos niveles bastante complicado. En el primer nivel, se definen los *logins*, que se utilizan para conectar con determinado servidor. Hasta cierto punto, estos *logins* son similares a los "usuarios" de InterBase, pues se asocian con el servidor, no con la base de datos. La validación de las contraseñas puede estar a cargo del propio SQL Server, o de Windows NT, y en este último caso, se habla de *seguridad integrada*. Con la seguridad integrada, los *logins* corresponden a usuarios del sistema operativo. Inicialmente, una instalación de SQL Server define el *login* del administrador como *sa*, sin contraseña.

En una segunda fase, al nivel ya de las bases de datos, se definen usuarios, hablando con propiedad. Los *logins* definidos en la fase anterior se asocian entonces con usuarios de una base de datos, y es a estos usuarios a los que se otorgan privilegios, me-

diante nuestras viejas y conocidas instrucciones **grant** y **revoke**. Al crearse una base de datos se define automáticamente un usuario especial como propietario de la misma, que se identifica como *dbo*: (*database owner*), y que tendrá acceso total a todos los objetos creados dentro de esa base de datos, ya sea por él mismo o por otro usuario. Por supuesto, el administrador del sistema también tiene acceso sin restricciones a esos objetos.

> Al desplegar la lista de posibles valores de la propiedad *TableName* de un componente de tablas en Delphi, las tablas existentes aparecen cualificadas con el nombre del usuario que las ha creado, que en MS SQL Server es generalmente el *dbo*. Lo mismo sucede en Oracle, aunque no en InterBase. Este comportamiento compromete la portabilidad de la aplicación entre distintos servidores, pero no es aconsejable eliminar el prefijo de usuario del nombre de la tabla. En tal caso, al BDE le resultará difícil consultar el catálogo de la base de datos para encontrar información sobre índices. Los errores que se producen como consecuencia son bastante perversos e insidiosos: cierta aplicación escrita en Delphi 3, que tuve que corregir, funcionaba perfectamente, hasta que alguien cambiaba dinámicamente el índice activo. Al concluir la ejecución, se producía un misterioso fallo de protección general. Por supuesto, todo se arregló al introducir los prefijos de usuarios en los nombres de tablas.

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, ha-

brá que considerar el caso especial en que el servidor sea MS SQL, pues los nombres de parámetros se almacenan estáticamente en el fichero *dfm*.

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. Esto es necesario porque no podemos utilizar el nivel de aislamiento *Repeatable reads* desde aplicaciones Delphi que accedan a este sistema.

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

MS 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**.

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
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* que implementa Transact-SQL, el lenguaje de programación de MS SQL Server, 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
       for {insert,update,delete}
       [with encryption]
       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.

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 respaldo:

```
create trigger GuardarBorrados
       on Clientes
       for delete as
       insert into CopiaRespaldo 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
inserted	Sí	No	Sí
deleted	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 MS SQL Server. Como los *triggers* se efectúan 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.

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 usuario que implemente la propagación de la modificación en el caso especial en que ésta afecta solamente a una fila de la tabla maestra.

Capítulo

25

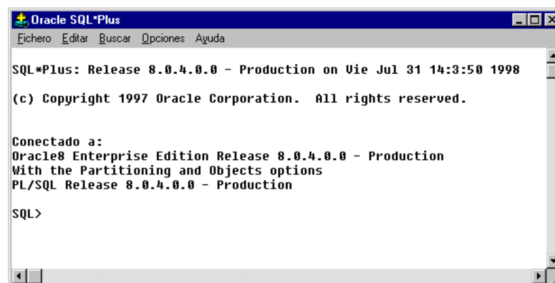
Oracle

LAS 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 4 ofrece para las mismas.

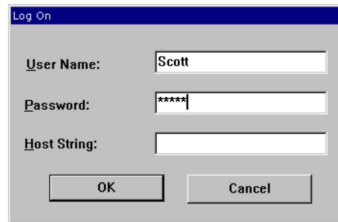
Sobreviviendo a SQL*Plus

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 5, 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
varchar2(n), nvarchar2(n)	Cadenas de caracteres de longitud variable
char(n), nchar(n)	Cadenas de caracteres de longitud fija
number(p,s)	Números, con escala y precisión
date	Fecha y hora, simultáneamente
long	Cadenas de caracteres de hasta 2GB

Tipo de dato	Significado
raw(n), long raw	Datos binarios
clob, nclob	Objetos de caracteres grandes
rowid	Posición de un registro
blob	Objetos binarios grandes
bfile	Puntero a un fichero binario externo a la base de datos
mlslabel	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]
    [ClausulaEnable];
```

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* (*table space*) 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. Por ejemplo, pueden colocarse de forma adyacente la cabecera de un pedido y las líneas de detalles asociadas. Esta técnica acelera los encuentros entre tales tablas y las relaciones *master/detail* que puede establecer una aplicación cliente. Es una posibilidad a tener en cuenta, pues afecta solamente al nivel físico de la base de datos, por lo que si mañana decide pasar la aplicación a otro tipo de servidor, a Delphi le dará exactamente lo mismo, aunque el programa no resulte tan eficiente como antes.

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**.

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
```

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
                where  Codigo = 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
                where  Codigo = 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
                where  Codigo = d.RefArticulo;
        end loop;
end;
/
```

Triggers en PL/SQL

Los *triggers* de Oracle combinan el comportamientos 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[22]. 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
        where  Codigo = :new.RefCliente;
        if i = 0 then
                raise_application_error(-20000,
                        'No existe tal cliente');
        end if;
```

[22] Informix es similar en este sentido.

```
end;
/
```

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*.

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
              :new.Codigo := CodigoCliente.NextVal;
       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*.

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 a Delphi, 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 de un grave problema: aunque permite transacciones con lecturas repetibles, estas tienen que ser sólo lectura. ¿Qué consecuencia trae esto? Supongamos que el valor secuencial se determina mediante estas dos 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;
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. Esta historia 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 4 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 <> '');
```

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 *TClientes*. 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 TClientes as
       Cli ref TClientes;
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;
);
/
```

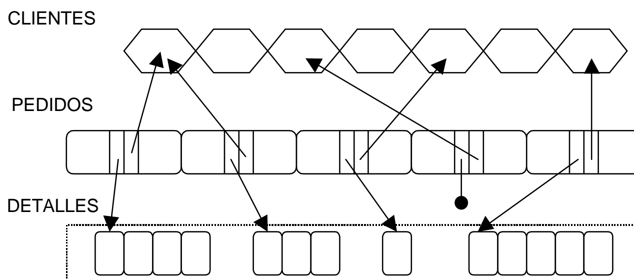
Esta vez no creamos una tabla de objetos, pero en compensación definimos un tipo de tabla anidada:

```
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 TClientes,
        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 8 permite declarar columnas que sean vectores. Estas columnas se parecen a las tablas anidadas, pero su tamaño máximo está 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.

Extensiones de Delphi para los tipos de objetos

Para poder manejar los nuevos tipos de datos de Oracle 8, Delphi 4 ha realizado cambios en el BDE, ha añadido cuatro nuevos tipos de campos, un nuevo tipo de conjunto de datos y ha efectuado modificaciones en el tipo *TDBGrid* para poder visualizar datos de objetos. Los cambios en el BDE son los siguientes:

- El parámetro *DLL32* del controlador de Oracle debe ser *sqlora8.dll*.
- *VENDOR INIT* debe valer *oci.dll*.
- *OBJECT MODE* debe valer *TRUE*.

Los nuevos tipos de campos son:

Tipo	Significado
TADTField	Para los objetos anidados
TArrayField	Representa un vector
TDataSetField	Tablas anidadas
TReferenceField	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 en la base de datos hemos definido la siguiente tabla:

```
create type TFraccion as object (
       Numerador      number(9),
       Denominador    number(9)
);
/

create table Probabilidades (
       Suceso         varchar2(30) not null primary key,
       Probabilidad   TFraccion not null
);
```

En Delphi, traemos un *TTable*, lo enganchamos a esta tabla y traemos todos los campos, mediante el comando *Add all fields*. Estos son los objetos que crea Delphi:

```
tbProb: TTable;
tbProbSUCESO: TStringField;
tbProbPROBABILIDAD: TADTField;
tbProbNUMERADOR: TIntegerField;
tbProbDENOMINADOR: TIntegerField;
```

Es decir, podemos trabajar directamente con los campos básicos de tipos simples, o atacar a la columna mediante el tipo *TADTField*. Por ejemplo, todas estas asignaciones son equivalentes:

```
tbProbNumerador.Value := 1;
tbProbProbabilidad.FieldValues[0] := 1;
tbProbProbabilidad[0] := 1;
tbProbProbabilidad.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:

Tipos ADT

SUCESO	PROBABILIDAD
Acertar Lotería	(1; 14000000)
Pagar impuestos	(1; 1)
Pillar un constipado	(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 rejillas de Delphi 4 tienen una nueva 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:

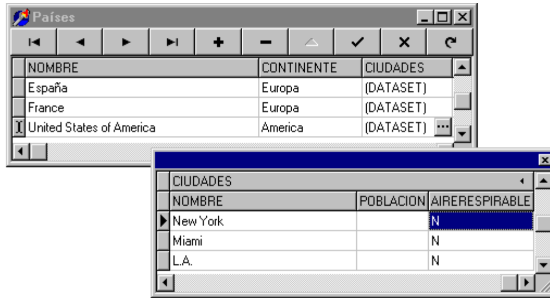
Tipos ADT

SUCESO	PROBABILIDAD	
	NUMERADOR	DENOMINADOR
Acertar Lotería	1	14000000
Pagar impuestos	1	1
Pillar un constipado	3	16

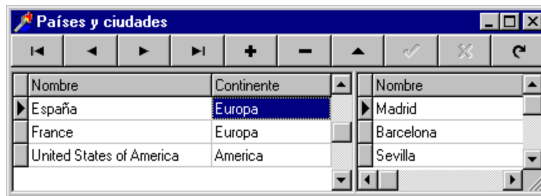
Un campo de tipo *TADTField* puede mostrarse, en modo sólo lectura, en cualquier otro control de datos, como un *TDBEdit*. Sin embargo, es más lógico que estos controles se conecten a los componentes simples del ADT. Por ejemplo, si quisiéramos editar la tabla anterior registro a registro, necesitaríamos 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 modifica-

ción aparece una ventana emergente con otra rejilla, esta vez asociada a las filas anidadas de la fila maestra activa, como en la siguiente imagen:



¿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.



> Es conveniente 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 (vea el último capítulo de esta parte), cuando 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 desciende 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].AsString);
```

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 utilizarse el siguiente código:

```
with TQuery.Create(nil) do
try
   DatabaseName := tbPedidos.DatabaseName;
   SQL.Add('select ref(Cli) from Clientes Cli');
   SQL.Add(''where Cli.Nombre = ' + tbClientesNombre.Value);
   Open;
   // Esta es la asignación deseada
   tbPedidoCliente.Assign(Fields[0]);
finally
   Free;
end;
```

Capítulo

26

Usando SQL con Delphi

TODO EL LENGUAJE SQL QUE HEMOS VISTO hasta el momento ha sido presentado desde la perspectiva del sistema de bases de datos que lo implementa. Ahora tenemos que incorporar esas instrucciones SQL dentro de nuestras aplicaciones programadas con Delphi. Aprovecharemos también la ocasión para conocer algunas peculiaridades del dialecto de SQL local implementado en el Motor de Datos de Borland.

El componente TQuery como conjunto de datos

Para enviar instrucciones SQL a la base de datos se utiliza el componente *TQuery*, que se encuentra en la página *Data Access* de la Paleta de Componentes. Desde el punto de vista de la jerarquía de herencia de la VCL, la clase *TQuery* desciende del tipo *TDBDataSet*, por lo que los objetos de consultas son conjuntos de datos. Esto quiere decir que podemos conectar una fuente de datos a un objeto de consultas para mostrar y editar su contenido desde controles de datos, que podemos movernos por sus filas, extraer información de sus campos; en definitiva, que casi todas las operaciones aplicables a las tablas son aplicables a este tipo de componente.

No obstante, un objeto *TQuery* sólo puede tratarse como un conjunto de datos en el caso especial de que la instrucción SQL que contenga sea una consulta. Si la instrucción pertenece al DDL, DCL, o es una de las instrucciones **update**, **insert** ó **delete**, no tiene sentido pensar en el resultado de la ejecución de la instrucción como si fuera un conjunto de datos. En este caso, tenemos métodos especiales para tratar con el componente.

Para una tabla, las propiedades *DatabaseName* y *TableName* determinan, en lo fundamental, el origen de los datos; para una consulta, necesitamos por lo menos asignar valores a *DatabaseName*, la base de datos contra la cual se ejecuta la instrucción, y *SQL*, la lista de cadenas que contiene la instrucción SQL en sí. Es posible omitir el valor de la propiedad *DatabaseName*. Si se omite esta propiedad, hay que especificar dentro de la instrucción SQL a qué base de datos pertenece cada tabla; más adelante veremos cómo hacerlo. Esta técnica, sin embargo, no es recomendable si queremos

trabajar con una base de datos SQL remota, pues en tal situación la evaluación de la consulta la realiza el SQL local de Delphi.

Un componente *TQuery* utilizado como conjunto de datos puede hacer uso de la propiedad *Active*, o de los métodos *Open* y *Close*, para abrir y cerrar la consulta. Una vez abierta la consulta, podemos aplicar casi todas las operaciones que son aplicables a tablas; la gran excepción son las operaciones que se implementan mediante índices. No obstante, también pueden aplicarse filtros a la consulta. El acceso a la información de cada campo se logra del mismo modo: se pueden crear campos persistentes en tiempo de diseño, o se puede acceder dinámicamente a los mismos con las propiedad *Fields*, *FieldValues* y con la función *FieldByName*.

¿Quién ejecuta las instrucciones?

Esta es una buena pregunta, pero estoy seguro de que ya intuye la respuesta. En primer lugar, si la petición se dirige a una base de datos local, esto es, si la base de datos asignada a *DatabaseName* se refiere al controlador *STANDARD*, la instrucción es interpretada en la máquina cliente por el denominado SQL Local, perteneciente al BDE. En contraste, si la propiedad *DatabaseName* se refiere a una base de datos remota, estamos ante lo que Delphi llama *passthrough SQL*, es decir, instrucciones SQL que se "pasan" al servidor remoto y que son ejecutadas por el mismo. Este comportamiento por omisión puede modificarse mediante el parámetro *SQLQRYMODE* de la configuración del BDE, aunque es recomendable dejarlo tal como está.

Las cosas se complican cuando se utilizan *consultas heterogéneas*. En este tipo de consultas, se mezclan datos de varias bases de datos. Estas consultas son interpretadas nuevamente por el SQL local. En una consulta heterogénea, los nombres de las bases de datos se indican delante de los nombres de tablas, en la cláusula **from**. Para poder utilizar este recurso, la propiedad *DatabaseName* de la consulta debe estar vacía o hacer referencia a un alias local. Por ejemplo, en la siguiente consulta mezclamos datos provenientes de una tabla de InterBase y de una tabla Paradox. Los nombres de tablas incluyen el alias utilizando la notación *:ALIAS:*, y deben encerrarse entre comillas. Claro está, necesitamos sinónimos para las tablas si queremos cualificar posteriormente los nombres de los campos:

```
select E.FULL_NAME, sum(O.ItemsTotal)
from   ":IBLOCAL:EMPLOYEE" E, ":DBDEMOS:ORDERS" O
where  O.EmpNo = E.EMP_NO
group  by E.FULL_NAME
order  by 2 desc
```

Cuando se trata de una tabla local para la cual sabemos el directorio, pero no tenemos un alias, podemos especificar el directorio en sustitución del nombre de alias:

```
select *
from   "C:\Program files\Borland\Delphi 2.0\Demos\Data\Orders.db"
```

La propiedad *Local* de la clase *TQuery* indica si la base de datos asociada a la consulta es cliente/servidor o de escritorio.

Consultas actualizables

Como sabemos, existen reglas matemáticas que determinan si una expresión relacional puede considerarse actualizable o no. En la práctica, los sistemas relacionales tienen sus propias reglas para determinar qué subconjunto de todas las expresiones posibles pueden ser actualizadas. Las reglas de actualizabilidad las encontramos cuando se definen vistas en un sistema SQL, y las volvemos a encontrar al establecer consultas sobre una base de datos desde el ordenador cliente. En Delphi, para pedir que una consulta retorne un cursor actualizable, de ser posible, es necesario asignar *True* a la propiedad lógica *RequestLive* de la consulta. El valor por omisión de esta propiedad es *False*.

No obstante, *RequestLive* es solamente una petición. El resultado de esta petición hay que extraerlo de la propiedad *CanModify* una vez que se ha activado la consulta. Si la consulta se ejecuta contra una base de datos local y su sintaxis permite la actualización, el BDE retorna un cursor actualizable; en caso contrario, el BDE proporciona un conjunto de datos de sólo lectura. Sin embargo, si la petición se establece contra un servidor SQL, puede llevarse la desagradable sorpresa de recibir un código de error si pide una consulta "viva" y el sistema se la niega.

Desafortunadamente, el algoritmo que decide si una expresión **select** es actualizable o no depende del sistema de base de datos. Aquí exponemos las reglas de actualizabilidad del SQL local; es una regla en dos partes, pues depende de si la instrucción utiliza una sola tabla o utiliza varias. Si se utiliza una sola tabla, deben cumplirse las siguientes restricciones para la actualizabilidad:

- La tabla base debe ser actualizable (…elemental, Watson…).
- No se utilizan **union**, **minus**, **intersect**, **group by** ó **having**.
- Si se utiliza **distinct**, que sea innecesario (¡!).
- No se permiten funciones de conjuntos en la selección.
- No se permiten subconsultas.
- Si existe un **order by**, que se pueda implementar mediante un índice.

La explicación de la regla 3 era demasiado larga y rompía la simetría de la lista. Quería decir simplemente que si aparece la palabra **distinct** en la cláusula de selección,

deben aparecer también todos los campos de la clave; en ese caso, la instrucción también puede escribirse sin esta opción.

Por otra parte, deben cumplirse las siguientes reglas si la cláusula **from** contiene varias tablas:

- Las tablas están formando todas un encuentro natural (*natural join*), o un encuentro externo (*outer join*) de izquierda a derecha.
- Los encuentros deben implementarse mediante índices.
- No se usa la cláusula **order by**.
- Cada tabla es una tabla base, no una vista.
- Se cumplen las restricciones aplicables de la primera lista.

InterBase, y la mayoría de los servidores SQL, no es tan melindroso a la hora de decidir si una consulta es actualizable o no: si existen un par de tablas en la cláusula **from**, la consulta no es actualizable. Sin embargo, existe una forma de permitir modificaciones sobre una consulta no actualizable. Consiste en activar las actualizaciones en caché para el componente *TQuery* y asociarle un objeto de actualización *TUpdateSQL*. Las actualizaciones en caché se controlan mediante la propiedad *CachedUpdates* de los conjuntos de datos, pero se estudiarán mucho más adelante. Los objetos de actualización, por su parte, permiten especificar reglas por separado para implementar los borrados, modificaciones e inserciones sobre la consulta; estas reglas son también instrucciones SQL.

Siempre hacia adelante

El uso más común de un objeto *TQuery* basado en una instrucción **select** es la visualización de sus datos. Esta visualización nos obliga a una implementación que nos permita movernos arbitrariamente dentro del conjunto de datos producido como resultado de la instrucción SQL. Casi todos los intérpretes SQL construyen una representación física del resultado, que puede consistir, en dependencia de si la consulta es actualizable o no, en una copia física temporal del conjunto de datos generado, o en un fichero de punteros a las filas originales de las tablas implicadas. A este tipo de estructura se le conoce con el nombre de *cursor bidireccional*.

En cambio, si solamente nos tenemos que desplazar por el cursor en una sola dirección, la implementación puede ser menos costosa en algunos casos. Por ejemplo, suponga que la consulta en cuestión consiste en una selección de filas a partir de una sola tabla, como la siguiente:

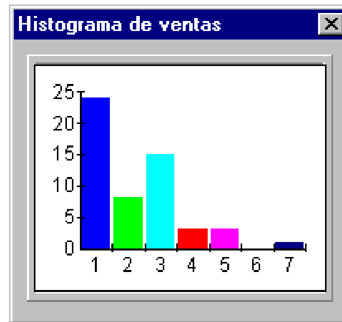
```
select *
from   Clientes
where  UltimoPedido > "4/7/96"
```

En tal caso, si solamente vamos a desplazarnos hacia adelante al usar esta consulta, el intérprete SQL lo único que tiene que hacer es abrir la tabla base, *Clientes*, e interpretar el método *Next* sobre la consulta como una sucesión de llamadas al método *Next* sobre la tabla base hasta que se cumpla la condición de la cláusula **where**.

Para ilustrar el uso de consultas unidireccionales, le mostraré cómo generar un gráfico de ventas a partir de la información de pedidos. El gráfico que nos interesa debe mostrar la cantidad de clientes por tramos de ventas totales: cuántos clientes nos han comprado hasta $50.000, cuántos de $50.001 hasta $100.000, etc. Por supuesto, necesitamos los totales de ventas por clientes, y esta información podemos extraerla mediante la siguiente consulta:

```
select count(ItemsTotal)
from   Orders
group  by CustNo
```

Esta instrucción se coloca dentro de un objeto *TQuery*, al cual se le modifica a *True* el valor de su propiedad *UniDirectional*.



Para mostrar el gráfico utilizaré el componente OCX *GraphicsServer*, dejando el estudio de *TeeChart* para un capítulo posterior. La programación de este control se divide en dos partes: primero hay que configurar la apariencia visual, lo que haremos en tiempo de diseño, y en tiempo de ejecución hay que cargar los datos en el control. La forma más sencilla de configurar visualmente el componente es invocar el menú local del mismo con el botón derecho del ratón y activar el comando para editar las propiedades del control.

> Este componente se ha importado con el nombre de *TGraph* en Delphi 4. Sea consecuente con este cambio de nombre, que afecta al nombre de la variable que debe apuntar al objeto desde el formulario.

Luego, para cargar datos en tiempo de ejecución nos basaremos en las siguientes propiedades:

Propiedad	Significado
NumSets, NumPoints	Número de series y de puntos en cada serie.
ThisSet, ThisPoint	Punteros a la celda actual. Se incrementan automáticamente al asignar valores a esta celda.
GraphData	El valor almacenado en la celda referida por las dos propiedades anteriores.

Nos interesa establecer los valores de umbral de la forma más flexible que podemos. Para esto declaramos un método en la definición de tipo del formulario:

```
type
   TForm1 = class(TForm)
      // …
   private
      procedure LlenarGrafico(const Valores: array of Currency);
   end;
```

La idea es llamar a este método desde la respuesta al evento *OnCreate* de la ventana:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
   LlenarGrafico([25000, 50000, 75000, 100000, 150000]);
end;
```

Bajo estas suposiciones, el método *LlenarGrafico* se implementa muy fácilmente:

```
procedure TForm1.LlenarGrafico(const Valores: array of Currency);
var
   I: Integer;
begin
   // Inicializar el control de gráficos
   GraphicsServer1.NumSets := 1;
   GraphicsServer1.NumPoints := High(Valores) + 2;
   for I := 1 to GraphicsServer1.NumPoints do
      GraphicsServer1.GraphData := 0;
   // Abrir la consulta y recorrer sus filas
   Query1.Open;
   try
      while not Query1.EOF do
      begin
         // Localizar el valor de umbral
         I := 0;
         while (I <= High(Valores))
            and (Query1.Fields[0].AsFloat > Valores[I]) do
            Inc(I);
         GraphicsServer1.ThisPoint := I + 1;
         GraphicsServer1.GraphData := GraphicsServer1.GraphData + 1;
         Query1.Next;
      end;
   finally
      Query1.Close;
   end;
end;
```

Observe que la consulta se examina en una pasada desde el principio hasta el final, sin necesidad de dar marcha atrás. El que realmente signifique algún adelanto establecer un cursor unidireccional o uno bidireccional depende de la implementación del intérprete SQL que ofrezca el sistema de bases de datos.

Consultas paramétricas

Es posible modificar en tiempo de ejecución el contenido de la propiedad *Sql* de una consulta. Esto se realiza en ocasiones para permitir que el usuario determine, casi siempre de una forma visual y más o menos intuitiva, qué información quiere obtener. El programa genera la instrucción SQL correspondiente y ¡zas!, el usuario queda satisfecho. Sin embargo, muchas veces los cambios que distinguen una instrucción de otra se refieren a valores constantes dentro de la expresión. En Delphi 1, las consultas paramétricas son utilizadas frecuentemente para suplir la carencia de filtros sobre conjuntos de datos.

Supongamos que el usuario quiere filtrar la tabla de clientes para mostrar los clientes de acuerdo al estado (*State*) al que pertenecen. Traemos a un formulario una rejilla de datos, *DBGrid1*, una fuente de datos, *DataSource1*, un cuadro de edición, *Edit1*, y una consulta *Query1*. Conectamos los componentes de base de datos como es usual, y modificamos las siguientes propiedades del cuadro de edición:

Propiedad	Valor
CharCase	*ecUpperCase*
MaxLength	*2*
Text	<Vacío>

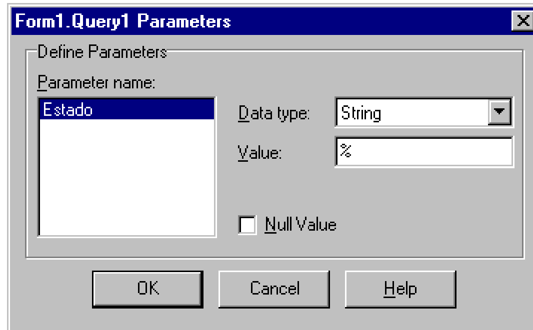
Ahora vamos a la consulta. Asignamos *dbdemos* a *DatabaseName*, y tecleamos la siguiente instrucción en la propiedad *SQL*:

```
select *
from   Customer
where  State like :Estado
```

La novedad es el operando *:Estado*, que representa un parámetro de la consulta. Es importante que los dos puntos vayan junto al nombre del parámetro, para que Delphi pueda reconocerlo como tal. En una instrucción SQL podemos utilizar tantos parámetros como queramos, y pueden utilizarse como sustitutos de constantes; nunca en el lugar de un nombre de tabla o de columna.

Después de tener el texto de la instrucción, necesitamos asignar un tipo a cada parámetro, e indicar opcionalmente un valor inicial para cada uno de ellos. Esto se hace editando la propiedad *Params*. Más adelante veremos un ejemplo en el cual es necesa-

rio no asociar tipos a parámetros, pero esta será la excepción, no la regla. En nuestro ejemplo, asignamos al único parámetro el tipo *String*, y le damos como valor inicial el carácter %; como recordará el lector, si se utiliza un signo de porcentaje como patrón de una expresión **like**, se aceptarán todas las cadenas de caracteres. Una vez que hayamos asignado los parámetros, podemos abrir la consulta, asignando a la propiedad *Active* a *True*.



La imagen anterior corresponde al editor de parámetros de Delphi 3. En Delphi 4 los parámetros se editan mediante el editor genérico de colecciones.

Para cambiar dinámicamente el valor de un parámetro de una consulta es necesario, en primer lugar, que la consulta esté inactiva. Del mismo modo que sucede con los campos, existen dos formas de acceder al valor de un parámetro: por su posición y por su nombre. Es muy recomendable utilizar el nombre del parámetro, pues es muy probable que si se modifica el texto de la instrucción SQL, la posición de un parámetro varíe también. Para acceder a un parámetro por su nombre, necesitamos la función *ParamByName*:

```
function TQuery.ParamByName(const Nombre: string): TParam;
```

De todos modos, también es posible utilizar la propiedad *Params*, mediante la cual obtenemos el puntero al parámetro por medio de su posición:

```
property Params[I: Integer]: TParam;
```

En nuestro pequeño ejemplo, la asignación al parámetro debe producirse cuando el usuario modifique el contenido del cuadro de edición. Interceptamos, en consecuencia, el evento *OnChange* del cuadro de edición:

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
   Query1.Close;
   try
      Query1.ParamByName('Estado').AsString := Edit1.Text + '%';
```

```
      finally
         Query1.Open;
      end;
   end;
```

Como se puede apreciar, se garantiza mediante un bloque de protección de recursos la reapertura de la consulta. Utilizamos la función *ParamByName* para acceder al parámetro; el nombre del parámetro se escribe ahora sin los dos puntos iniciales. Por último, hay que utilizar la propiedad *AsString* (o *AsInteger* o *AsDateTime*…) para manipular el valor del parámetro; no existe una propiedad parecida a *Value*, como en el caso de los campos, para hacer uso del valor sin considerar el tipo del parámetro.

Consultas dependientes

Es posible asociar a un parámetro el valor obtenido de una columna perteneciente a otra tabla o consulta. El cambio de parámetro se produce automáticamente cada vez que cambia la fila en el primer conjunto de datos. De este modo, se puede lograr un efecto similar a las tablas de detalles, pero sobre consultas SQL. A este tipo de consultas se le denomina *consultas dependientes* ó, en castellano antiguo: *linked queries*.

Para poder asociar un parámetro de una consulta a una tabla, siga estas instrucciones:

- Asigne a la propiedad *DataSource* de la consulta una fuente de datos enlazada a la tabla maestra. Esta es una propiedad con un nombre inadecuado. Le regalaría una botella de champagne al equipo de desarrolladores de Borland si en la próxima versión de Delphi cambiasen el nombre de la propiedad a *MasterSource*, ¡palabra de programador!
- No asigne tipo al parámetro en el editor de la propiedad *Params*.

Considere, por ejemplo, la siguiente consulta:

```
select Parts.Description, Items.Qty, Items.Discount
from   Items, Parts
where  Items.PartNo = Parts.PartNo
```

Mediante esta instrucción podemos obtener un listado de los artículos vendidos, junto a sus descripciones. Esto pudiera servirnos para sustituir a los campos de referencia de Delphi 2 si queremos mostrar las descripciones de artículos en una rejilla de Delphi 1. Pero la instrucción muestra *todos* los artículos vendidos, mientras que el uso más frecuente de esta información es mostrarla como detalles de los pedidos.

En este caso, la instrucción necesaria es la siguiente:

```
select Parts.Description, Items.Qty, Items.Discount
from   Items, Parts
where  Items.PartNo = Parts.PartNo
  and  Items.OrderNo = :OrderNo
```

El inconveniente principal de esta técnica, si se utiliza para sustituir los campos de búsqueda de Delphi, es que el resultado de la consulta no es actualizable. Le sugiero que pruebe esta consulta con la base de datos *mastsql.gdb*, que se encuentra en el directorio de demostraciones de Delphi. El intérprete SQL de InterBase es más eficiente para este tipo de cosas que el SQL local del BDE.

La preparación de la consulta

Si una consulta con parámetros va a ser abierta varias veces, es conveniente *prepararla* antes de su ejecución. Preparar una consulta quiere decir realizar su análisis sintáctico y producir el código de ejecución necesario; por supuesto, cada servidor realiza esta tarea de forma diferente. La preparación de una consulta se realiza por medio del método *Prepare* de la clase *TQuery*. Normalmente, esta operación se realiza automáticamente durante la apertura de la consulta, y la operación inversa tiene lugar cuando se cierra la consulta. Sin embargo, la preparación consume tiempo y recursos. Si los parámetros cambian varias veces durante la vida del objeto de consulta, estaremos repitiendo la misma cantidad de veces la tediosa operación de preparación. La solución es preparar la consulta explícitamente, y deshacer, consecuentemente, la preparación antes de destruir el objeto, o cuando no se vaya a utilizar por mucho tiempo. En especial, si va a utilizar una consulta dependiente, tenga en cuenta que cada vez que se cambie la fila activa de la tabla maestra, se está cerrando y reabriendo la consulta asociada.

Si las instrucciones de apertura y cierre de la consulta se realizan explícitamente, la preparación se puede programar de esta manera:

```
procedure TQueryForm.FormCreate(Sender: TObject);
begin
   if not Query1.Prepared then
      Query1.Prepare;
   Query1.Open;
end;

procedure TQueryForm.FormClose(Sender: TObject);
begin
   Query1.Close;
   if Query1.Prepared then
      Query1.UnPrepare;
end;
```

¿Y qué sucede si hace falta que la consulta esté abierta en tiempo de diseño? El problema es que cuando se dispara el evento *OnCreate* del formulario o del módulo de

datos, ya las consultas y tablas que estaban abiertas en tiempo de diseño han sido activadas. Alguien puede tener la idea de realizar la preparación en el evento *Before-Open* de estos componentes. Pero no pierda su tiempo: en pruebas realizadas por el autor, la preparación de la consulta en el evento *BeforeOpen* del conjunto de datos no reportaba ventaja alguna con respecto a la preparación automática. Lo cual quiere decir que, al producirse este evento, Delphi ya ha preparado la consulta por sí mismo y no agradece nuestra intervención.

Una solución de fuerza bruta para estos casos puede ser cerrar primeramente la consulta, prepararla y volver a abrirla, mediante un código parecido al siguiente:

```
procedure TQueryForm.FormCreate(Sender: TObject);
begin
   Query1.Close;       // Cerrar primeramente la consulta
   if not Query1.Prepared then
      Query1.Prepare;
   Query1.Open;        // Reabrir la consulta
end;
```

Pero conozco un truco mejor. Vaya a la sección **protected** de la declaración del módulo o formulario (si no hay, créela), y declare el siguiente método:

```
type
   TQueryForm = class(TForm)
      //…
   protected
      procedure Loaded; override;
      // …
   end;
```

El método *Loaded* es invocado por el formulario después de haber leído todos sus componentes hijos, pero antes de aplicar propiedades como *Active*, *Connected*, etc. Es decir, se llama justo en el momento que necesitamos. Mediante la directiva **override** informamos a Delphi que pretendemos sustituir su implementación predefinida mediante el siguiente método:

```
procedure TQueryForm.Loaded;
begin
   inherited Loaded;
   Query1.Prepare;
end;
```

Este truco de redefinir *Loaded* puede ser útil también para modificar durante la carga de una aplicación los parámetros de conexión de un componente *TDatabase*.

Actualización de datos con SQL

La mayoría de los programadores novatos piensa en SQL como en un lenguaje limitado a la selección de datos. Pero, como hemos visto, con este lenguaje podemos también crear objetos en la base de datos, eliminarlos y modificar la información asociada a ellos. Cuando se utilizan instrucciones de ese tipo con un objeto *TQuery*, éste no puede tratarse como un conjunto de datos, que es lo que hemos visto hasta el momento.

Pongamos como ejemplo que queremos aumentarle el salario a todos los empleados de nuestra base de datos. Podemos entonces traer un objeto *TQuery* al formulario (o al módulo de datos), asignarle un valor a la propiedad *DatabaseName* y escribir la siguiente instrucción en la propiedad *SQL*:

```
update Employee
set    Salary = Salary * (1 + :Puntos / 100)
```

Lo hemos complicado un poco utilizando un parámetro para el tanto por ciento del aumento. Ahora, nada de activar la consulta ni de traer fuentes de datos. Lo único que se necesita es ejecutar esta instrucción en respuesta a alguna acción del usuario:

```
procedure TForm1.AumentoClick(Sender: TObject);
begin
   Query1.ParamByName('Puntos').AsInteger := 5;
   Query1.ExecSql;
end;
```

Como se ve, la ejecución de la consulta se logra llamando al método *ExecSql*. Con anterioridad se ha asignado un valor al parámetro *Puntos*.

¿Cómo saber ahora cuántos empleados han visto aumentar sus ingresos? El componente *TQuery* tiene la propiedad *RowsAffected*, que puede ser consultada para obtener esta información:

```
procedure TForm1.AumentoClick(Sender: TObject);
begin
   Query1.ParamByName('Puntos').AsInteger := 5;
   Query1.ExecSql;
   ShowMessage(Format('Has traido la felicidad a %d personas',
      [Query1.RowsAffected]));
end;
```

En el ejemplo previo, la instrucción estaba almacenada en un objeto *TQuery* incluido en tiempo de diseño. También se pueden utilizar instrucciones almacenadas en objetos creados en tiempo de ejecución. Se puede incluso programar un método genérico que nos ahorre toda la preparación y limpieza relacionada con la ejecución de una instrucción SQL. El siguiente procedimiento es uno de mis favoritos; en mi

ordenador tengo una unidad en la cual incluyo ésta y otras funciones parecidas para utilizarlas de proyecto en proyecto:

```
function EjecutarSql(const ADB: string;
   const Instruccion: array of string): Integer;
var
   I: Integer;
begin
   with TQuery.Create(nil) do
   try
      DatabaseName := ADB;
      for I := Low(Instruccion) to High(Instruccion) do
         Sql.Add(Instruccion[I]);
      ExecSql;
      Result := RowsAffected;
   finally
      Free;
   end;
end;
```

Para suministrar la instrucción utilizo, en el segundo parámetro, un vector abierto de cadenas de caracteres. El procedimiento se usará de este modo:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   EjecutarSql('dbdemos',
      ['update employee',
       'set salary = salary * 1.05']);
end;
```

He dividido la consulta en dos líneas simplemente por comodidad. Recuerde que a partir de Delphi 2 las cadenas de caracteres no están limitadas a 255 caracteres de longitud.

Almacenar el resultado de una consulta

Y ya que vamos de funciones auxiliares, he aquí otra de mis preferidas: una que guarda el resultado de una consulta en una tabla. Bueno, ¿y no existe acaso una instrucción **insert**…**into** en SQL? Sí, pero el intérprete de SQL local de Delphi no la soportó hasta hace muy poco.

El procedimiento que explico a continuación se basa en el método *BatchMove* de los componentes *TTable*, que permite copiar datos de un conjunto de datos a una tabla. El conjunto de datos puede ser, por supuesto, una tabla, un procedimiento almacenado del tipo apropiado o, como en este caso, una consulta SQL.

```
procedure GuardarConsulta(const SourceDB: string;
   const AQuery: array of string;
   const TargetDB, ATableName: string);
var
   Q: TQuery;
   I: Integer;
begin
   Q := TQuery.Create(nil);
   try
      Q.DatabaseName := SourceDB;
      for I := Low(AQuery) to High(AQuery) do
         Q.Sql.Add(AQuery[I]);
      with TTable.Create(nil) do
      try
         DatabaseName := TargetDB;
         TableName := ATableName;
         BatchMove(Q, batCopy);
      finally
         Free;
      end;
   finally
      Q.Free;
   end;
end;
```

Una importante aplicación de este procedimiento puede ser la de realizar copias locales de datos procedentes de un servidor SQL en los ordenadores clientes.

¿Ejecutar o activar?

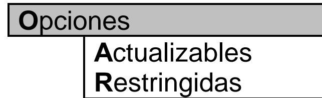
El ejemplo que voy a exponer en este epígrafe quizás no le sea muy útil para incluirlo en su próxima aplicación de bases de datos, pero le puede ayudar con la comprensión del funcionamiento de las excepciones. El ejercicio consiste en crear un intérprete para SQL, en el cual se pueda teclear cualquier instrucción y ejecutarla pulsando un botón. Y el adjetivo "cualquier" aplicado a "instrucción" es el que nos causa problemas: ¿cómo distinguir antes de ejecutar la instrucción si es una instrucción DDL, de manipulación o una consulta? Porque los dos primeros tipos necesitan una llamada al método *ExecSQL* para su ejecución, mientras que una consulta la activamos con el método *Open* o la propiedad *Active*, y necesitamos asociarle un *DataSource* para visualizar sus resultados.

Los componentes necesarios en este proyecto son los siguientes:

- *Memo1*: Un editor para teclear la instrucción SQL que deseamos ejecutar.
- *Query1*: Un componente de consultas, para ejecutar la instrucción tecleada.
- *DataSource1*: Fuente de datos, conectada al objeto anterior.
- *DBGrid1*: Rejilla de datos, conectada a la fuente de datos.
- *bnEjecutar*: Botón para desencadenar la ejecución de la instrucción.

- *cbAlias*: Un combo, con el estilo *csDropDownList*, para seleccionar un alias de los existentes.

Habilitaremos además un menú con opciones para controlar qué tipo de consultas queremos:



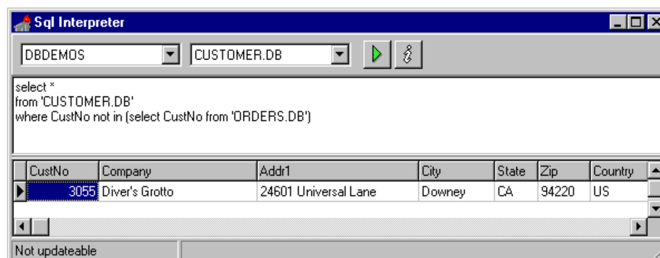
Estos dos comandos, a los cuales vamos a referirnos mediante las variable *miActualizables* y *miRestringidas*, funcionarán como casillas de verificación; la respuesta a ambas es el siguiente evento compartido:

```
procedure TForm1.CambiarOpción(Sender: TObject);
begin
   with Sender as TMenuItem do
      Checked := not Checked;
end;
```

Necesitaremos alguna forma de poder indicar el valor de la propiedad *DatabaseName* de la consulta. En vez de teclear el nombre de la base de datos, voy a utilizar un método de una clase que todavía no hemos estudiado: el método *GetDatabaseNames*, de la clase *TSession*. En el capítulo 30, sobre bases de datos y sesiones, se estudia esta clase con más detalle. Por el momento, sólo necesitamos saber que la función inicializa una lista de cadenas con los nombres de los alias disponibles en el momento actual. Esta inicialización tiene lugar durante la creación del formulario:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
   Session.GetDatabaseNames(cbAlias.Items);
   cbAlias.ItemIndex := 0;
end;
```

Cuando pulsemos el botón *bnEjecutar*, debemos transferir el contenido del memo a la propiedad *SQL* de la consulta, y el nombre del alias seleccionado, que se encuentra en la propiedad *cbAlias.Text*, a la propiedad *DatabaseName* de la consulta. Una vez realizadas estas asignaciones, intentaremos abrir la consulta mediante el método *Open*.



¡Un momento! ¿No habíamos quedado en que había que analizar si el usuario había tecleado un **select**, o cualquier otra cosa? Bueno, esa sería una solución, pero voy a aprovechar un truco de Delphi: si intentamos activar una instrucción SQL que no es una consulta, se produce una excepción, ¡pero la instrucción se ejecuta de todos modos! Lo cual quiere decir que podemos situar la llamada a *Query1.Open* dentro de una instrucción **try…except**, y capturar esa instrucción en particular, dejando que las restantes se propaguen para mostrar otros tipos de errores al usuario. La excepción que se produce en el caso que estamos analizando es de tipo *ENoResultSet*, y el mensaje de error, en la versión original, es *"Error creating cursor handle"*. He aquí la respuesta completa a la pulsación del botón *Ejecutar*:

```
procedure TForm1.bnEjecutarClick(Sender: TObject);
begin
   Query1.Close;
   Query1.DatabaseName := cbAlias.Text;
   Query1.Sql := Memo1.Lines;
   Query1.RequestLive := miActualizables.Checked;
   Query1.Constrained := miRestringidas.Checked;
   try
      Query1.Open;
      if Query1.CanModify then
         StatusBar1.SimpleText := 'Consulta actualizable'
      else
         StatusBar1.SimpleText := 'Consulta no actualizable';
   except
      on E: ENoResultSet do
         StatusBar1.SimpleText := 'Instrucción ejecutable';
   end;
end;
```

> En Delphi 1 y Delphi 2 no existe la excepción *ENoResultSet*, por lo que el error de activación del cursor corresponde a una excepción de tipo *EDatabaseError*.

Adicionalmente, hemos colocado una barra de estado para mostrar información sobre el tipo de instrucción. Si es una instrucción ejecutable, se muestra el mensaje correspondiente. Si hemos tecleado una consulta, pedimos que ésta sea actualizable o no en dependencia del estado del comando de menú *miActualizable*; el resultado de nuestra petición se analiza después de la apertura exitosa del conjunto de datos, observando la propiedad *CanModify*.

El programa puede ampliarse fácilmente añadiendo soporte para comentarios. Por ejemplo, una forma rápida, aunque no completa, de permitir comentarios es sustituir la asignación a la propiedad SQL de la consulta por el siguiente código:

```
Query1.Sql.Clear;
for I := 0 to Memo1.Lines.Count - 1 do
   if Copy(TrimLeft(Memo1.Lines[I]), 1, 2) <> '--' then
      Query1.Sql.Add(Memo1.Lines[I]);
```

De esta forma, se descartan todas las líneas cuyos dos primeros caracteres sean dos guiones consecutivos: el inicio de comentario del lenguaje modular de SQL. El lector puede completar el ejemplo eliminando los comentarios que se coloquen al final de una línea.

Utilizando procedimientos almacenados

Para ejecutar un procedimiento almacenado desde una aplicación escrita en Delphi debemos utilizar el componente *TStoredProc*, de la página *Data Access* de la Paleta de Componentes. Esta clase hereda, al igual que *TTable* y *TQuery*, de la clase *TDBDataSet*. Por lo tanto, técnicamente es un conjunto de datos, y esto quiere decir que se le puede asociar un *TDataSource* para mostrar la información que contiene en controles de datos. Ahora bien, esto solamente puede hacerse en ciertos casos, en particular, para los procedimientos de selección de Sybase. ¿Recuerda el lector los procedimientos de selección de InterBase, que explicamos en el capítulo sobre *triggers* y procedimientos almacenados? Resulta que para utilizar estos procedimientos necesitamos una instrucción SQL, por lo cual la forma de utilizarlos desde Delphi es por medio de un componente *TQuery*.

En casi todos los casos, la secuencia de pasos para utilizar un *TStoredProc* es la siguiente:

- Asigne el nombre de la base de datos en la propiedad *DatabaseName*, y el nombre del procedimiento almacenado en *StoredProcName*.
- Edite la propiedad *Params*. *TStoredProc* puede asignar automáticamente los tipos a los parámetros, por lo que el objetivo de este paso es asignar opcionalmente valores iniciales a los parámetros de entrada.
- Si el procedimiento va a ejecutarse varias veces, prepare el procedimiento, de forma similar a lo que hacemos con las consultas paramétricas.
- Asigne, de ser necesario, valores a los parámetros de entrada utilizando la propiedad *Params* o la función *ParamByName*.
- Ejecute el procedimiento mediante el método *ExecProc*.
- Si el procedimiento tiene parámetros de salida, después de su ejecución pueden extraerse los valores desde la propiedad *Params* o por medio de la función *ParamByName*.

Pongamos por caso que queremos estadísticas acerca de cierto producto, del que conocemos su código. Necesitamos saber cuántos pedidos se han realizado, qué cantidad se ha vendido, por qué valor y el total de clientes interesados. Toda esa información puede obtenerse mediante el siguiente procedimiento almacenado:

```
create procedure EstadisticasProducto(CodProd int)
       returns (TotalPedidos int, CantidadTotal int,
                TotalVentas int, TotalClientes int)
as
declare variable Precio int;
begin
       select Precio
       from   Articulos
       where  Codigo = :CodProd
       into   :Precio;
       select count(Numero), count(distinct RefCliente)
       from   Pedidos
       where  :CodProd in
              (select RefArticulo from Detalles
               where  RefPedido = Numero)
       into   :TotalPedidos, :TotalClientes;
       select sum(Cantidad),
              sum(Cantidad*:Precio*(100-Descuento)/100)
       from   Detalles
       where  Detalles.RefArticulo = :CodProd
       into   :CantidadTotal, :TotalVentas;
end ^
```

Para llamar al procedimiento desde Delphi, configuramos en el módulo de datos un componente *TStoredProc* con los siguientes valores:

Propiedad	Valor
DatabaseName	El alias de la base de datos
StoredProcName	*EstadisticasProducto*

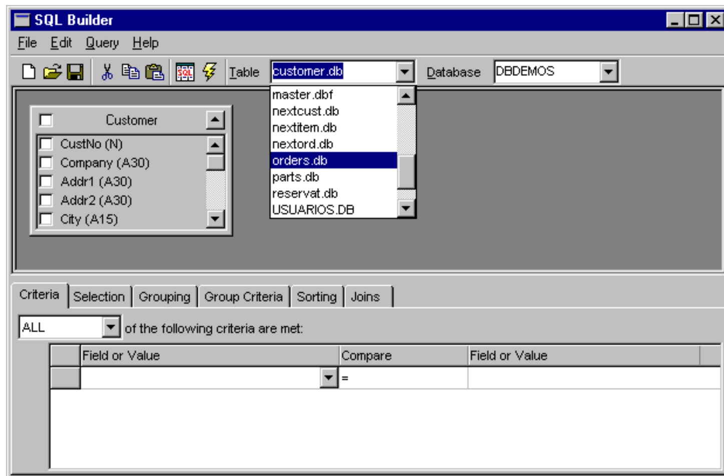
Después, para ejecutar el procedimiento y recibir la información de vuelta, utilizamos el siguiente código:

```
procedure TForm1.MostrarInfo(Sender: TObject);
var
   S: string;
begin
   S := '';
   if not InputQuery('Información', 'Código del producto', S)
      or (Trim(S) = '') then Exit;
   with modDatos.StoredProc1 do
   begin
      ParamByName('CodProd').AsString := S;
      ExecProc;
      ShowMessage(Format('Pedidos: %d'#13'Clientes: %d'#13 +
         'Cantidad: %d'#13'Total: %m',
         [ParamByName('TotalPedidos').AsInteger,
          ParamByName('TotalClientes').AsInteger,
          ParamByName('CantidadTotal').AsInteger,
          ParamByName('TotalVentas').AsFloat]));
   end;
end;
```

Al total de ventas se le ha dado formato con la secuencia de caracteres *%m*, que traduce un valor real al formato nacional de moneda.

Visual Query Builder

Las versiones cliente/servidor de Delphi vienen acompañadas con una utilidad denominada *Visual Query Builder*, ó Constructor Visual de Consultas. No la busque dentro del directorio de ejecutables de Delphi, pues no es una herramienta que se ejecute por separado. Su objetivo es ayudar en la creación de consultas dentro de un componente *TQuery*, y se activa mediante la opción *SQL builder* del menú de contexto asociado a un *TQuery*.

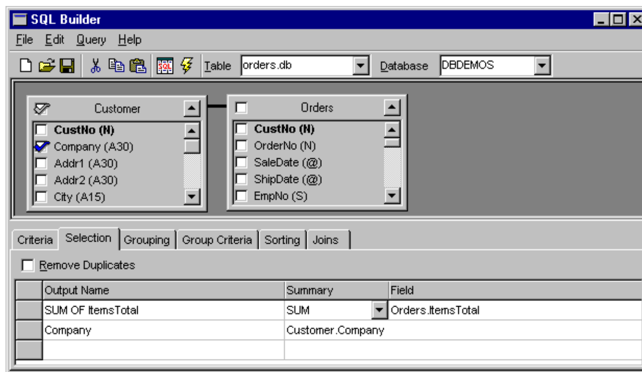


Para ilustrar el uso de la herramienta, diseñaremos una consulta que nos muestre un listado de clientes junto al total de dinero que han invertido en pedidos. Queremos además que el listado quede ordenado en forma descendente de acuerdo al total gastado; al que más nos haya comprado le enviaremos una tarjeta de felicitación por Navidades. Añadimos sobre un formulario vacío un componente *TQuery*, y asignamos *dbdemos* a la propiedad *DatabaseName* de la consulta. Después pulsamos el botón derecho del ratón sobre la misma, y activamos el comando *SQL Builder*.

Para añadir tablas, debemos seleccionarlas en el combo de la izquierda. En nuestro caso, sabemos que los nombres de clientes se encuentran en la tabla *customer.db*, y que los totales por factura están en *orders.db*: las cabeceras de pedidos; añadimos entonces ambas tablas. No se preocupe si añade una tabla de más o de menos, pues más adelante se pueden eliminar y añadir tablas. Ya tenemos la cláusula **from** de la consulta.

Como el lector sabe, si en una consulta mencionamos dos tablas y no la relacionamos, obtendremos el temible *producto cartesiano*: todas las combinaciones de filas posibles, aún las más absurdas. Para relacionar estas dos tablas entre sí, arrastramos el campo *CustNo* de *customer* sobre el campo del mismo nombre de *orders*. Debe aparecer una línea que une a ambos campos. Si se ha equivocado, puede seleccionar la línea, eliminarla con la tecla SUPR y reintentar la operación. Ya tenemos la cláusula **where**.

El próximo paso es indicar qué columnas se mostrarán como resultado de la consulta. Realice un doble clic sobre la columna *Company* de la tabla de clientes (el nombre de la empresa), y sobre *ItemsTotal*, de la tabla de pedidos (el total por factura). Esto determina la cláusula **select**. Si quiere ver el texto generado para la consulta, pulse un botón que dice "SQL"; si quiere ver los datos que devuelve la consulta, pulse el botón que tiene el rayo de Júpiter.



Todavía no está lista la consulta, pues se muestra una fila por cada pedido. Nos hace falta agrupar el resultado, para lo cual activamos la página *Selection*, y pulsamos el botón derecho del ratón sobre el campo *ItemsTotal*. En el menú que se despliega indicamos que este campo debe mostrar realmente una estadística (*Summary*). Entonces se divide en dos la celda, y en la nueva celda de la izquierda debemos seleccionar qué tipo de estadística necesitamos: en nuestro caso, *SUM*. Después ejecutamos la consulta generada, para comprobar el resultado de la operación; el texto de la consulta, hasta aquí, es el siguiente:

```
SELECT SUM( Orders.ItemsTotal ), Customer.Company
FROM   "customer.db" Customer
       INNER JOIN "orders.db" Orders
       ON  (Customer.CustNo = Orders.CustNo)
GROUP  BY Customer.Company
```

Nos falta ordenar el resultado por la segunda columna; seleccionamos la pagina *Sorting*, añadimos la columna *ItemsTotal* al criterio de ordenación, y pedimos que se ordene descendentemente. La consulta final es la siguiente:

```
SELECT SUM( Orders.ItemsTotal ) Orders."SUM OF ItemsTotal",
       Customer.Company
FROM   "customer.db" Customer
       INNER JOIN "orders.db" Orders
       ON  (Customer.CustNo = Orders.CustNo)
GROUP  BY Customer.Company
ORDER  BY Orders."SUM OF ItemsTotal" DESC
```

Por supuesto, para un programador experto en SQL puede ser más rápido y sencillo teclear directamente el texto de la instrucción. Recuerde también que ésta es una utilidad para tiempo de diseño. Si queremos permitir la generación visual de consultas al usuario final de nuestras aplicaciones, existen buenas herramientas en el mercado diseñadas para este propósito.

> La interfaz de esta herramienta ha cambiado en Delphi 4 con respecto a versiones anteriores. El último paso efectuado, añadir un criterio de ordenación basado en una expresión, generaba en versiones anteriores una instrucción incorrecta que había que corregir a mano.

Capítulo

27

Comunicación cliente/servidor

TODA LA COMUNICACIÓN ENTRE EL Motor de Datos de Borland y los servidores SQL tiene lugar mediante sentencias SQL, incluso cuando el programador trabaja con tablas, en vez de con consultas. Para los desarrolladores en entornos cliente/servidor es de primordial importancia comprender cómo tiene lugar esta comunicación. En la mayoría de los casos, el BDE realiza su tarea eficientemente, pero hay ocasiones en las que tendremos que echarle una mano.

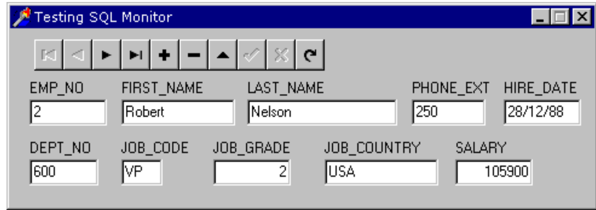
El propósito de este capítulo es enseñarle cómo detectar estas situaciones. Para lograrlo, veremos cómo el BDE traduce a instrucciones SQL las instrucciones de navegación y búsqueda sobre tablas y consultas. La forma en que se manejan las actualizaciones será estudiada más adelante.

Nuestra arma letal: SQL Monitor

Necesitamos un espía que nos cuente qué está pasando entre el servidor y nuestro cliente, y para esta labor contrataremos al SQL Monitor. Esta utilidad puede lanzarse desde el menú de programas de Windows, o directamente desde el menú *Database*, opción *SQL Monitor*, del propio Delphi. La ventana principal de este programa muestra las distintas instrucciones enviadas por el BDE y las respuestas que éste recibe del servidor. Podemos especificar qué tipos de instrucciones o de respuestas queremos mostrar en esta ventana mediante el comando de menú *Options|Trace options*. En ese mismo cuadro de diálogo se ajusta el tamaño del *buffer* que albergará las instrucciones. En principio, dejaremos las opciones tal y como vienen de la fábrica. Más adelante podrá desactivar algunas de ellas para mostrar una salida más compacta y legible.

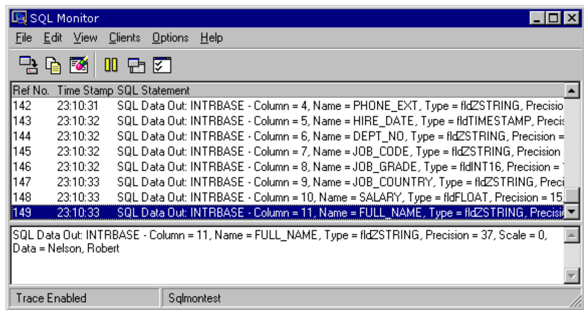
También necesitaremos una aplicación de prueba, que utilice una base de datos cliente/servidor. Para simplificar, utilizaremos un *TTable* asociado a la tabla *employee* del alias de InterBase *iblocal*. Crearemos los objetos de acceso a campos, los arrastraremos a la superficie del formulario, y añadiremos una barra de navegación. Importante: no se le ocurra utilizar una rejilla por el momento, pues se complicará la lectura e interpretación de los resultados de SQL Monitor.

La siguiente imagen muestra nuestra aplicación de pruebas en funcionamiento:



¿Tabla o consulta?

Estando "apagada" la aplicación, lance SQL Monitor, mediante el método que más le guste. Luego, ejecute la aplicación y vea la salida generada:



¡Nada menos que 149 entradas[23], solamente por abrir una tabla y leer el primer registro! Hagamos una prueba. Sustituya la tabla con una consulta que tenga la siguiente instrucción:

```
select * from employee
```

Repita ahora el experimento, ¡y verá que con sólo 15 entradas se puede comenzar a ejecutar la aplicación!

Hay un chiste (muy malo) acerca de unos científicos y un cangrejo. Al desdichado crustáceo le van arrancando las patas, mientras le ordenan verbalmente que se mueva. Al final, el animal se queda tieso en la mesa y no obedece las órdenes. Resultado anotado por los científicos: un cangrejo sin patas no oye. Parecida es la conclusión a la que llegaron algunos "gurús" de Delphi al ver estos resultados: hay que

---

[23] El número concreto de entradas puede variar, en función del sistema de bases de datos del servidor, de la ubicación del cliente, las opciones de trazado, etc. Los números que se den más adelante son también orientativos.

utilizar siempre consultas, en vez de tablas, si se está programando en un entorno cliente/servidor.

Y es que hay trampa en el asunto. ¿Se ha fijado que la consulta tiene la propiedad *RequestLive* igual a *False*? Cámbiela a *True* y repita la prueba, para que vea cómo vuelve a dispararse el contador de entradas en el monitor. Y pruebe después ir al último registro, tanto con la consulta como con la tabla, para que vea que la ventaja inicial de la consulta desaparece en este caso.

¿Qué está pasando? ¿Cómo podemos orientarnos entre la maraña de instrucciones del SQL Monitor? La primera regla de supervivencia es:

"Concéntrese en las instrucciones SQL Prepare y SQL Execute"

La razón es que éstas son las órdenes que envía el BDE al servidor. Repitiendo la apertura de la tabla, ¿con qué sentencias SQL nos tropezamos? Helas aquí:

```
select rdb$owner_name, rdb$relation_name, rdb$system_flag,
       rdb$view_blr, rdb$relation_id
from   rdb$relations
where  rdb$relation_name = 'employee'
```

El propósito de la sentencia anterior es comprobar si existe o no la tabla *employee*. Si esta instrucción diera como resultado un conjunto de filas vacío, fallaría la apertura de la tabla.

```
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
```

No hace falta ser un especialista en InterBase para darse cuenta de lo que está pasando. El BDE está extrayendo de las tablas del sistema la información sobre qué campos, índices y restricciones están definidas para esta tabla. Estos datos se almacenan dentro de las propiedades *FieldDefs* e *IndexDefs* de la tabla.

Finalmente, se abre la consulta básica para extraer datos de la tabla:

```
select emp_no, first_name, last_name, phone_ext, hire_date, dept_no,
       job_code, job_grade, job_country, salary, full_name
from   employee
order  by emp_no asc
```

Anote como detalle el que la consulta ordene las filas ascendentemente por el campo *Emp_No*, que es la clave primaria de esta tabla. Dentro de poco comprenderemos por qué.

La caché de esquemas

Traiga un botón al formulario y asocie el siguiente método con su evento *OnClick*:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   Table1.Close;
   Table1.Open;
end;
```

Si pulsamos este botón durante la ejecución de la aplicación, veremos que la segunda vez que se abre la misma tabla no se produce la misma retahíla de sentencias que al inicio, sino que directamente se pasa a leer el primer registro del cursor. La explicación es que la tabla *ya* tiene la información de su esquema almacenada en las propiedades *FieldDefs* e *IndexDefs*. Esta es una buena optimización, porque disminuye el tráfico de datos en la red. Sin embargo, cada vez que se vuelve a ejecutar la aplicación partimos de cero, y hay que traer otra vez todos los datos del catálogo. Imagine una empresa con cincuenta empleados, todos conectando su ordenador a las 9:15 de la mañana (sí, porque de las 9:00 hasta entonces, café y cotilleo) y arrancando su aplicación ...

Este problema es el que resuelve la opción *ENABLE SCHEMA CACHE* del BDE, que vimos en el capítulo sobre la configuración del Motor de Datos. Así que ya sabe por qué es recomendable activar siempre la caché de esquemas.

Operaciones de navegación simple

Volvemos a las pruebas con la aplicación. Limpie el *buffer* de SQL Monitor, y pulse el botón de la barra de navegación que lo lleva al último registro de la tabla. Esta es la sentencia generada por el BDE, después de cerrar el cursor activo:

```
select emp_no, first_name, last_name, phone_ext, hire_date, dept_no,
       job_code, job_grade, job_country, salary, full_name
from   employee
order  by emp_no desc
```

¡Ha cambiado el criterio de ordenación! Por supuesto, cuando el BDE ejecute la siguiente sentencia **fetch** el registro leído será el último de la tabla. Si seguimos navegando hacia atrás, con el botón *Prior*, el BDE solamente necesita ejecutar más instrucciones **fetch**, como en el *moon walk* de Michael Jackson: parece que nos movemos hacia atrás, pero en realidad nos movemos hacia delante.

Este truco ha sido posible gracias a la existencia de una clave primaria única sobre la tabla. Algunos sistemas SQL admiten que una clave primaria sea nula, permitiendo un solo valor nulo en esa columna, por supuesto. El problema es que, según el estándar SQL, al ordenar una secuencia de valores siendo algunos de ellos nulos, estos últimos siempre aparecerán en la misma posición: al principio o al final. Por lo que invertir el criterio de ordenación en el **select** no nos proporcionará los mismos datos en sentido inverso. El BDE no podrá practicar sus habilidades, y nos obligará a leer el millón de registros de la tabla a través de nuestra frágil y delicada red.

Esto mismo sucede con un *TQuery*, sea actualizable o no. Cuando vamos al final del cursor, siempre se leen todos los registros intermedios. Cangrejo sin patas no oye, ¿verdad?

> Cuando se indica un criterio de ordenación para la tabla, ya sea mediante *IndexName* o *IndexFieldNames*, se cambia la cláusula **order by** del cursor del BDE. Sin embargo, sucede algo curioso cuando el criterio se especifica en *IndexName*: el BDE extrae los campos del índice para crear la sentencia SQL. Si la tabla es de InterBase, aunque el índice sea descendente la cláusula **order by** indicará el orden ascendente. Esto, evidentemente, es un *bug*, pues nos fuerza a utilizar una consulta si queremos ver las filas de una tabla ordenadas en forma descendente por determinada columna.

Búsquedas exactas con Locate

Añada al formulario un cuadro de edición (*TEdit*) y un botón. Con estos componentes vamos a organizar una búsqueda directa por código. Cree la siguiente respuesta al evento *OnClick* del botón:

```
procedure TForm2.Button2Click(Sender: TObject);
begin
   if not Table1.Locate('EMP_NO', Edit1.Text, []) then
      Beep;
end;
```

Esta es una búsqueda exacta. Primero se prepara la siguiente instrucción:

```
select emp_no, first_name, last_name, phone_ext, hire_date, dept_no,
       job_code, job_grade, job_country, salary, full_name
from   employee
where  emp_no = ?
```

La sentencia tiene un parámetro. 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 emp_no, first_name, last_name, phone_ext, hire_date, dept_no,
       job_code, job_grade, job_country, salary, full_name
from   employee
where  emp_no < ?
order  by emp_no desc
```

Es decir, el BDE abre un cursor descendente con los registros cuyo código es menor que el actual.

Búsquedas parciales

Cambiemos ahora la respuesta al evento *OnClick* del botón de búsqueda, para que efectúe una búsqueda parcial sobre la columna del apellido del empleado:

```
procedure TForm2.Button2Click(Sender: TObject);
begin
   if not Table1.Locate('LAST_NAME', Edit1.Text,
      [loPartialKey]) then Beep;
end;
```

Si tecleamos GUCK, encontraremos el registro de Mr. Guckenheimer, cuyo código de empleado es el 145. En este caso, el BDE dispara dos consultas consecutivas sobre la tabla. Esta es la primera:

```
select emp_no, first_name, last_name, phone_ext, hire_date, dept_no,
       job_code ,job_grade ,job_country ,salary ,full_name
from   employee
where  last_name = ?
```

La consulta anterior intenta averiguar si hay alguien cuyo apellido sea exactamente "Guck". Si no sucede tal cosa, se dispara la segunda consulta:

```
select emp_no, first_name, last_name, phone_ext, hire_date, dept_no,
       job_code ,job_grade ,job_country ,salary ,full_name
from   employee
where  last_name > ?
order  by last_name asc
```

Evidentemente, si un apellido comienza con "Guck", es posterior alfabéticamente a este prefijo, y es esto lo que busca la segunda consulta: el primer registro cuyo apellido es mayor que el patrón de búsqueda tecleado. Eficiente, ¿no es cierto? Sin embargo, una pequeña modificación puede arruinar la buena reputación del BDE:

```
procedure TForm2.Button2Click(Sender: TObject);
begin
   if not Table1.Locate('LAST_NAME', Edit1.Text,
      [loPartialKey, loCaseInsensitive]) then Beep;
end;
```

La búsqueda ahora es también insensible a mayúsculas y minúsculas. Si teclea nuevamente GUCK, estando al principio de la tabla, verá cómo en el SQL Monitor aparece una instrucción **fetch** por cada registro de la tabla. De lo que se puede deducir que si la tabla tuviera un millón de registros y estuviéramos buscando el último, podríamos dejar la máquina encendida e irnos a disfrutar de una semana de merecidas vacaciones.

> El lector puede pensar, como pensó en su momento el autor, que la culpa de este comportamiento la tiene InterBase, que no permite índices insensibles a mayúsculas y minúsculas. No obstante, lo mismo sucede con Oracle y con SQL Server, como puede comprobar fácilmente quien tenga acceso a estos sistemas.

Una solución para búsquedas parciales rápidas

¿Qué pasa si para nuestra aplicación cliente/servidor son indispensables las búsquedas parciales rápidas insensibles a mayúsculas y minúsculas? Por ejemplo, queremos que el usuario pueda ir tecleando letras en un cuadro de edición, y en la medida en que teclea, se vaya desplazando la fila activa de la tabla que tiene frente a sí. En este caso, lo aconsejable es ayudar un poco al BDE.

Coloque un componente *TQuery* en el formulario que hemos estado utilizando, conéctelo a la base de datos y suminístrele la siguiente instrucción SQL:

```
select emp_no
from   employee
where  upper(last_name) starting with :patron
```

Con esta consulta pretendemos localizar los empleados cuyos apellidos comienzan con el prefijo que pasaremos en el parámetro *patron*. Edite la propiedad *Params* de la consulta y asígnele a este parámetro el tipo **string**. Como la consulta se ejecuta en el servidor, nos ahorramos todo el tráfico de red que implica el traernos todos los registros. Claro, ahora es responsabilidad del servidor el implementar eficientemente la consulta.

Por último, elimine el botón de búsqueda junto con el método asociado a su *OnClick*, y asocie esta respuesta al evento *OnChange* del cuadro de edición:

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
   if Edit1.Text = '' then
      Table1.First
   else
   begin
      Query1.Params[0].AsString := AnsiUpperCase(Edit1.Text);
      Query1.Open;
      try
         if Query1.EOF or not Table1.Locate('EMP_NO',
            Query1.Fields[0].Value, []) then Beep;
      finally
         Query1.Close;
      end;
   end;
end;
```

Cuando el usuario teclea algo en el cuadro de edición, se ejecuta la consulta para recuperar los códigos de los posibles empleados. Evidentemente, si la consulta está vacía no existen tales empleados. En caso contrario, se escoge el primero de ellos y se busca su fila, utilizando una búsqueda exacta por el código mediante *Locate*, la cual ya sabemos que es rápida y segura.

El algoritmo anterior puede mejorarse con un par de trucos. Primero, puede prepararse la consulta antes de utilizarla, para ahorrarnos la compilación de la misma cada vez que busquemos algo. Y podemos hacer lo mismo que el BDE: utilizar una primera consulta que intente localizar el registro que corresponda exactamente al valor tecleado, antes de buscar una aproximación.

Búsquedas con filtros latentes

Para terminar este capítulo, analizaremos cómo el BDE implementa los filtros y rangos para las bases de datos cliente/servidor. Asigne la siguiente expresión en la propiedad *Filter* de la tabla:

```
Job_Grade = 5 and Salary >= 30000
```

A continuación active el filtro mediante la propiedad *Filtered*, y ejecute la aplicación. El cursor que abre el BDE añade la expresión de filtro dentro de su cláusula **where**:

```
select emp_no, first_name, last_name, phone_ext, hire_date, dept_no,
       job_code, job_grade, job_country, salary, full_name
from   employee
where  (job_grade = ? and salary >= ?)
order  by emp_no asc
```

Aunque se están utilizando parámetros para las constantes, hasta aquí no hay ningún misterio. Llegamos a la conclusión, además, que los filtros se implementan eficientemente en sistemas cliente/servidor, pues la responsabilidad de la selección de filas queda a cargo del servidor. Sin embargo, es una conclusión prematura. Considere la inocente expresión:

```
Last_Name = 'B*'
```

Como hemos dicho en un capítulo anterior, con esta expresión se pretende seleccionar los empleados cuyos apellidos comienzan con la letra B. Al activar el filtro, podemos comprobar con SQL Monitor que el BDE trae al cliente todos los registros de la tabla, y los descarta localmente, a pesar de que la expresión anterior es equivalente a esta otra, que sabemos que se implementa eficientemente:

```
Last_Name >= 'B' and LastName < 'C'
```

El mismo problema se presenta cuando se añade la opción *foCaseInsensitive* a la propiedad *FilterOptions*. Por lo tanto, hay que tener cuidado con las expresiones que vamos a utilizar como filtro.

> Los rangos se implementan de forma similar a las expresiones de filtro equivalentes. En tablas cliente/servidor es preferible, entonces, utilizar filtros en vez de rangos, pues son menos las limitaciones de los primeros: no hay que tener las filas ordenadas, se pueden establecer condiciones para los valores de varias filas simultáneamente...

Es interesante ver cómo se las arregla el BDE para navegar con *FindFirst*, *FindPrior*, *FindNext* y *FindLast* por las filas definidas por un filtro no activo, o filtro latente. Supongamos que *Filtered* sea *False*, y que a *Filter* le asignamos la expresión:

```
Job_Country <> 'USA'
```

Cuando se ejecuta el método *FindFirst*, el BDE lanza la siguiente sentencia:

```
select emp_no, first_name, last_name, phone_ext, hire_date, dept_no,
       job_code, job_grade, job_country, salary, full_name
from   employee
where  job_country <> ?
order  by emp_no asc
```

Para buscar el siguiente (*FindNext*), se utiliza el código de registro actual:

```
select emp_no, first_name, last_name, phone_ext, hire_date, dept_no,
       job_code, job_grade, job_country, salary, full_name
from   employee
where  job_country <> ? and emp_no > ?
order  by emp_no asc
```

Las operaciones *FindLast* y *FindPrior* se implementan de forma similar, invirtiendo solamente el orden de las filas. Y por supuesto, pueden surgir problemas si se utilizan filtros con búsquedas parciales o insensibles a mayúsculas y minúsculas.

> He dejado el análisis de las operaciones de actualización sobre bases de datos SQL para el capítulo 31.

5

Actualizaciones y concurrencia

- Actualizaciones
- Eventos de transición de estados
- Bases de datos y sesiones
- Transacciones y control de concurrencia
- Actualizaciones en caché
- Libretas de bancos

Parte

Capítulo

28

Actualizaciones

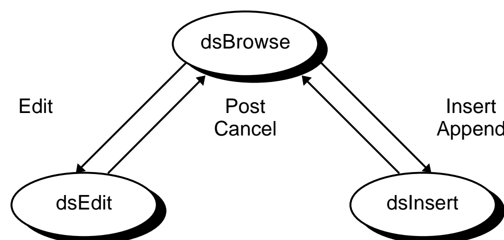
EL VIAJE MÁS LARGO comienza con un simple paso. Incluso las bases de datos más grandes han comenzado como tablas vacías … y alguien se ha encargado de introducir los datos. En este capítulo iniciaremos el estudio de los métodos de actualización. Se trata de un tema extenso, por lo cual nos limitaremos al principio a considerar actualizaciones sobre tablas aisladas, en entornos de un solo usuario, dejando para más adelante las actualizaciones coordinadas y el control de transacciones y concurrencia.

Los estados de edición y los métodos de transición

Como ya he explicado en el capítulo 15, cuando un conjunto de datos se encuentra en el estado *dsBrowse*, no es posible asignar valores a los campos, pues se produce una excepción. Para realizar modificaciones, hay que cambiar el estado del conjunto de datos a uno de los estados *dsEdit* ó *dsInsert*. Para realizar la transición a estos estados, debemos utilizar el método *Edit*, si es que deseamos realizar modificaciones sobre el registro activo; si queremos añadir registros debemos utilizar *Insert* ó *Append*.

```
// Para modificar
Table1.Edit;
// … secuencia de modificación …
```

```
// Para añadir un nuevo registro
Table1.Append;
// … secuencia de inserción …
```



Cuando estudiamos los controles de datos, vimos que se podía teclear directamente valores sobre los mismos, y modificar de este modo los campos, sin necesidad de llamar explícitamente al método *Edit*. Este comportamiento se controla desde la fuente de datos, el objeto *DataSource*, al cual se conectan los controles de datos. Si

este componente tiene el valor *True* en su propiedad *AutoEdit*, una modificación en el control provocará el paso del conjunto de datos al estado *Edit*. Puede ser deseable desactivar la edición automática cuando existe el riesgo de que el usuario modifique inadvertidamente los datos que está visualizando.

El método *Edit* debe releer siempre el registro actual, porque es posible que haya sido modificado desde otro puesto desde el momento en que lo leímos por primera vez.

Se puede aprovechar el comportamiento de *Edit*, que es válido tanto para bases de datos de escritorio como para bases de datos cliente/servidor, para releer el registro actual sin necesidad de llamar al método *Refresh*, que es potencialmente más costoso. Para asegurarnos de que la fila activa tiene los valores actuales, ejecute el siguiente par de instrucciones:

```
Table1.Edit;
Table1.Cancel;
```

Es necesario llamar a *Cancel* para devolver la tabla al estado original *dsBrowse*.

Asignaciones a campos

Bien, ya tenemos los campos de la tabla en un modo receptivo a nuestras asignaciones. Ahora tenemos que realizar dichas asignaciones, y ya hemos visto cómo realizar asignaciones a un campo al estudiar la implementación de los campos calculados. Recordemos las posibilidades:

- Asignación por medio de variables de campos creados mediante el Editor de Campos. Es la forma más eficiente de asignación, y la más segura, pues se comprueba su validez en tiempo de compilación.

```
tbClientes.Edit;
tbClientesLastInvoiceDate.Value := Date; // Ultima factura
// …
```

- Asignación mediante los objetos de campo creados en tiempo de ejecución, por medio de la función *FieldByName* o la propiedad *Fields*. *FieldByName* es menos estable frente a cambios de nombres de columnas, y respecto a errores tipográficos. Tampoco suministra información en tiempo de compilación acerca del tipo de campo. La propiedad *Fields* debe utilizarse solamente en casos especiales, pues nos hace depender además del orden de definición de las columnas.

```
tbClientes.Edit;
tbClientes.FieldByName('LastInvoiceDate').AsDateTime := Date;
// …
```

- Asignación mediante la propiedad *FieldValues*, ya sea de forma implícita o explícita. Es la forma menos eficiente de todas, pues realiza la búsqueda de la columna y se asigna a una propiedad *Variant*, pero es quizás la más flexible.

```
tbClientes.Edit;
tbClientes['LastInvoiceDate'] := Date;
// Equivalente a: tbClientes.FieldValues['LastInvoiceDate'] := Date
```

La asignación de valores a campos mediante la propiedad *FieldValues* nos permite asignar el valor **null** de SQL a un campo. Para esto, utilizamos la constante variante especial *Null*:

```
tbClientes['LastInvoiceDate'] := Null;
```

Pero también puede utilizarse el método *Clear* del campo:

```
tbClientesLastInvoiceDate.Clear;
```

Es necesario tener bien claro que, aunque en Paradox y dBase un valor nulo se representa mediante una cadena vacía, esto no es así para tablas SQL.

Otra posibilidad es utilizar el método *Assign* para copiar el contenido de un campo en otro. Por ejemplo, si *Table1* y *Table2* son tablas con la misma estructura de campos, el siguiente bucle copia el contenido del registro activo de *Table2* en el registro activo de *Table1*; se asume que antes de este código, *Table1* se ha colocado en alguno de los estados de edición:

```
for I := 0 to Table1.FieldCount - 1 do
   Table1.Fields[I].Assign(Table2.Fields[I]);
```

Cuando se copia directamente el contenido de un campo a otro con *Assign* deben coincidir el tipo de los campos y sus tamaños. Sin embargo, si los campos son campos BLOB, esta restricción se relaja. Incluso puede asignarse a estos campos el contenido de un memo o de una imagen:

```
Table1Foto.Assign(Image1.Picture);
```

Por último, la propiedad *Modified* nos indica si se han realizado asignaciones sobre campos del registro activo que no hayan sido enviadas aún a la base de datos:

```
procedure GrabarOCancelar(ADataSet: TDataSet);
begin
   if ADataSet.State in dsEditModes then
```

```
    if ADataSet.Modified then
      ADataSet.Post
    else
      ADataSet.Cancel;
end;
```

Confirmando las actualizaciones

Una vez realizadas las asignaciones sobre los campos, podemos elegir entre confirmar los cambios o descartar las modificaciones, regresando en ambos casos al estado inicial, *dsBrowse*. Para confirmar los cambios se utiliza el método *Post*, indistintamente de si el conjunto de datos se encontraba en el estado *dsInsert* o en el *dsEdit*. El método *Post* corresponde, como ya hemos explicado, al botón que tiene la marca de verificación de las barras de navegación. Como también hemos dicho, *Post* es llamado implícitamente por los métodos que cambian la fila activa de un conjunto de datos, pero esta técnica es recomendada sólo para acciones inducidas por el usuario, nunca como recurso de programación. Siempre es preferible un *Post* explícito.

Si la tabla o consulta se encontraba inicialmente en el modo *dsInsert*, los valores actuales de los campos se utilizan para crear un nuevo registro en la tabla base. Si por el contrario, el estado inicial es *dsEdit*, los valores asignados modifican el registro activo. En ambos casos, y esto es importante, si la operación es exitosa la fila activa de la tabla corresponde al registro nuevo o al registro modificado.

Para salir de los estados de edición sin modificar el conjunto de datos, se utiliza el método *Cancel*, que corresponde al botón con la "X" en la barra de navegación. *Cancel* restaura el registro modificado, si el estado es *dsEdit*, o regresa al registro previo a la llamada a *Insert* ó *Append*, si el estado inicial es *dsInsert*. Una característica interesante de *Cancel* es que si la tabla se encuentra en un estado diferente a *dsInsert* ó *dsEdit* no pasa nada, pues se ignora la llamada.

Por el contrario, es una precondición de *Post* que el conjunto de datos se encuentre alguno de los estados de edición; de no ser así, se produce una excepción. Hay un método poco documentado, llamado *CheckBrowseMode*, que se encarga de asegurar que tras su llamada, el conjunto de datos quede en el modo *dsBrowse*. Si la tabla o la consulta se encuentra en alguno de los modos de edición, se intenta una llamada a *Post*. Si el conjunto de datos está inactivo, se lanza entonces una excepción. Esto nos ahorra repetir una y otra vez la siguiente instrucción:

```
if Table1.State in dsEditModes then Table1.Post;
```

Es muy importante, sobre todo cuando trabajamos con bases de datos locales, garantizar que una tabla siempre abandone el estado *Edit*. La razón, como veremos más adelante, es que para tablas locales *Edit* pide un bloqueo, que no es devuelto hasta

que se llame a *Cancel* ó *Post*. Una secuencia correcta de edición por programa puede ser la siguiente:

```
Table1.Edit;
try
   // Asignaciones a campos …
   Table1.Post;
except
   Table1.Cancel;
   raise;
end;
```

Diferencias entre Insert y Append

¿Por qué *Insert* y también *Append*? Cuando se trata de bases de datos SQL, los conceptos de inserción *in situ* y de inserción al final carecen de sentido, pues en este tipo de sistemas no existe el concepto de posición de registro. Por otra parte, el formato de tablas de dBase no permite una implementación eficiente de la inserción *in situ*, por lo cual la llamada al método *Insert* es siempre equivalente a una llamada a *Append*.

En realidad, en el único sistema en que estos dos métodos tienen un comportamiento diferente es en Paradox, en el caso especial de las tablas definidas sin índice primario. En este caso, *Insert* es realmente capaz de insertar el nuevo registro después del registro activo. Pero este tipo de tablas tiene poco uso, pues no se pueden definir índices secundarios en Paradox si no existe antes una clave primaria.

Pero la explicación anterior se refiere solamente al resultado final de ambas operaciones. Si estamos trabajando con una base de datos cliente/servidor, existe una pequeña diferencia entre *Insert* y *Append*, a tener en cuenta especialmente si paralelamente estamos visualizando los datos de la tabla en una rejilla. Cuando se realiza un *Append*, la fila activa se desplaza al final de la tabla, por lo que el BDE necesita leer los últimos registros de la misma. Luego, cuando se grabe el registro, la fila activa volverá a desplazarse, esta vez a la posición que le corresponde de acuerdo al criterio de ordenación activo. En el peor de los casos, esto significa releer dos veces la cantidad de registros que pueden aparecer simultáneamente en pantalla. Por el contrario, si se trata de *Insert*, solamente se produce el segundo desplazamiento, pues inicialmente la fila activa crea un "hueco" en la posición en que se encontraba antes de la inserción. Esta diferencia puede resultar o no significativa.

> Si la tabla en que se está insertando contiene registros ordenados por algún campo secuencial, o por la fecha de inserción, y está ordenada por ese campo, es preferible utilizar *Append*, pues lo normal es que el registro quede definitivamente al final del cursor.

Como por azar…

¿Un pequeño ejemplo? Vamos a generar aleatoriamente filas para una tabla; esta operación es a veces útil para comprobar el funcionamiento de ciertas técnicas de Delphi. La tabla para la cual generaremos datos tendrá una estructura sencilla: una columna *Cadena*, de tipo cadena de caracteres, y una columna *Entero*, de tipo numérico. Supondremos que la clave primaria de esta tabla consiste en el campo *Cadena*; para el ejemplo actual es indiferente qué clave está definida. Lo primero será crear una función que nos devuelva una cadena alfabética aleatoria de longitud fija:

```
function RandomString(Longitud: Byte): string;
const
   Vocales = 'AEIOU';
var
   I: Integer;
begin
   Result := '';
   SetLength(Result, Longitud);
   for I := 1 to Longitud do
      if (I > 1) and
         not (Result[I-1] in ['A','E','I','O','U','N','S']) then
         Result[I] := Vocales[Random(5) + 1]
      else
         Result[I] := Chr(Random(26) + Ord('A'));
end;
```

Me he tomado incluso la molestia de favorecer las secuencias consonante/vocal. El procedimiento que se encarga de llenar la tabla es el siguiente:

```
procedure LlenarTabla(Tabla: TTable; CantRegistros: Integer);
var
   Intentos: Integer;
begin
   Randomize;
   Intentos := 3;
   while CantRegistros > 0 do
   try
      Tabla.Append;
      Tabla['Cadena'] := RandomString(
         Tabla.FieldByName('Cadena').Size);
      Tabla['Entero'] := Random(MaxInt);
      Tabla.Post;
      Intentos := 3;
      Dec(CantRegistros);
   except
      Dec(Intentos);
      if Intentos = 0 then raise;
   end;
end;
```

La mayoría de las excepciones se producirán por violaciones de la unicidad de la clave primaria. En definitiva, las excepciones son ignoradas, a no ser que sobrepasemos el número predefinido de intentos; esto nos asegura contra el desbordamiento

de la capacidad de un disco y otros factores imprevisibles. El número de registros se decrementa solamente cuando se produce una grabación exitosa.

Métodos abreviados de inserción

Del mismo modo que *FindKey* y *FindNearest* son formas abreviadas para la búsqueda basada en índices, existen métodos para simplificar la inserción de registros en tablas y consultas. Estos son los métodos *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 el puntero vacío, **nil**, en la posición correspondiente:

```
Table1.InsertRecord(['Valor1', nil, 'Valor3']);
// …
Table1.AppendRecord([RandomString(Tabla.FieldByName('Cadena').Size),
   Random(MaxInt)]);
```

En Delphi 2, 3 y 4 contamos también con el método *SetFields*, que asigna valores a los campos de una tabla a partir de un vector de valores:

```
Table1.Edit;
Table1.SetFields(['Valor1', nil, 'Valor3']);
Table1.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!

Actualización directa vs variables en memoria

En la programación tradicional para bases de datos locales lo común, cuando se leen datos del teclado para altas o actualizaciones, es leer primeramente los datos en va-

riables de memoria y, posteriormente, transferir los valores leídos a la tabla. Esto es lo habitual, por ejemplo, en la programación con Clipper. El equivalente en Delphi sería ejecutar un cuadro de diálogo "normal", con controles *TEdit*, *TComboBox* y otros extraídos de la página *Standard* y, si el usuario acepta los datos tecleados, poner la tabla en modo de edición o inserción, según corresponda, asignar entonces el resultado de la edición a las variables de campo y terminar con un *Post*:

```
if DialogoEdicion.ShowModal = mrOk then
begin
   Table1.Insert;
   Table1Nombre.Value := DialogoEdicion.Edit1.Text;
   Table1Edad.Value := StrToInt(DialogoEdicion.Edit2.Text);
   Table1.Post;
end;
```

Esta técnica es relativamente simple, gracias a que la actualización tiene lugar sobre una sola tabla. Cuando los datos que se suministran tienen una estructura más compleja, es necesario recibir y transferir los datos utilizando estructuras de datos más complicadas. Por ejemplo, para entrar los datos correspondientes a una factura, hay que modificar varios registros que pertenecen a diferentes tablas: la cabecera del pedido, las distintas líneas de detalles, los cambios en el inventario, etc. Por lo tanto, los datos de la factura se leen en variables de memoria y, una vez que el usuario ha completado la ficha de entrada, se intenta su grabación en las tablas.

Por el contrario, el estilo preferido de programación para bases de datos en Delphi consiste en realizar siempre las actualizaciones directamente sobre las tablas, o los campos de la tabla, utilizando los componentes de bases de datos. ¿Qué ganamos con esta forma de trabajo?

- Evitamos el código que copia el contenido del cuadro de edición al campo. Esta tarea, repetida para cada pantalla de entrada de datos de la aplicación, puede convertirse en una carga tediosa, y el código generado puede consumir una buena parte del código total de la aplicación.
- La verificación de los datos es inmediata; para lograrlo en Clipper hay que duplicar el código de validación, o esperar a que la grabación tenga lugar para que salten los problemas. Ciertamente, el formato xBase es demasiado permisivo al respecto, y la mayor parte de las restricciones se verifican por la aplicación en vez de por el sistema de base de datos, pero hay que duplicar validaciones tales como la unicidad de las claves primarias. Esto también cuesta código y, lo peor, tiempo de ejecución.

Y ahora viene la pregunta del lector:

– Vale, tío listo, pero ¿qué haces si después de haber grabado la cabecera de un pedido y cinco líneas de detalles encuentras un error en la sexta línea? ¿Vas bo-

rrando una por una todas las inserciones e invirtiendo cada modificación hecha a registros existentes?

Realmente, esto es todo un problema, sobre todo en Delphi 1 cuando se trabaja sobre bases de datos locales, y está relacionado con las actualizaciones de objetos complejos cuyos datos se almacenan sobre varias tablas. En este capítulo no podemos dar una solución completa al problema. La pista consiste, sin embargo, en utilizar el mecanismo de transacciones, que será estudiado más adelante, en combinación con las actualizaciones en caché. Así que, por el momento, confiad en mí.

Automatizando la entrada de datos

Lo que hemos visto, hasta el momento, es la secuencia de pasos necesaria para crear o modificar registros por medio de programación. Antes mencioné la posibilidad de utilizar componentes *data-aware* para evitar la duplicación de los datos tecleados por el usuario en variables y estructuras en memoria, dejando que los propios controles de edición actúen sobre los campos de las tablas. Supongamos que el usuario está explorando una tabla en una rejilla de datos. Si quiere modificar el registro que tiene seleccionado, puede pulsar un botón que le hemos preparado con el siguiente método de respuesta:

```
procedure TForm1.bnEditarClick(Sender: TObject);
begin
   Table1.Edit;
   DialogoEdicion.ShowModal;
   // Hasta aquí, por el momento …
end;
```

La ventana *DialogoEdicion* debe contener controles *data-aware* para modificar los valores de las columnas del registro activo de *Table1*. *DialogoEdicion* tiene también un par de botones para cerrar el cuadro de diálogo, los típicos *Aceptar* y *Cancelar*. He aquí lo que debe suceder al finalizar la ejecución del cuadro de diálogo:

- Si el usuario pulsa el botón *Aceptar*, debemos grabar, o intentar grabar, los datos introducidos.
- Si el usuario pulsa el botón *Cancelar*, debemos abandonar los cambios, llamando al método *Cancel*.

Lo más sencillo es verificar el valor de retorno de la función *ShowModal*, para decidir qué acción realizar:

```
// Versión inicial
procedure TForm1.bnEditarClick(Sender: TObject);
begin
   Table1.Edit;
```

```
    if DialogoEdicion.ShowModal = mrOk then
      Table1.Post
    else
      Table1.Cancel;
  end;
```

Pero este código es muy malo. La llamada a *Post* puede fallar por las más diversas razones, y el usuario recibirá el mensaje de error con el cuadro de diálogo cerrado, cuando ya es demasiado tarde. Además, esta técnica necesita demasiadas instrucciones para cada llamada a un diálogo de entrada y modificación de datos.

Por lo tanto, debemos intentar la grabación cuando el cuadro de diálogo está todavía activo. El primer impulso del programador es asociar la pulsación del botón *Aceptar* con una llamada a *Post*, y una llamada a *Cancel* con el botón *Cancelar*. Pero esto nos obliga a escribir demasiado código cada vez que creamos una nueva ventana de entrada de datos, pues hay que definir tres manejadores de eventos: dos para los eventos *OnClick* de ambos botones, y uno para el evento *OnCloseQuery*, del formulario. En este último evento debemos preguntar si el usuario desea abandonar los cambios efectuados al registro, si es que existen y el usuario ha cancelado el cuadro de diálogo.

En mi opinión, el mejor momento para realizar la grabación o cancelación de una modificación o inserción es durante la respuesta al evento *OnCloseQuery*. Durante ese evento podemos, basándonos en el resultado de la ejecución modal que está almacenado en la propiedad *ModalResult* del formulario, decidir si grabamos, cancelamos o sencillamente si nos arrepentimos sinceramente de abandonar nuestros cambios. El código que dispara el evento *OnCloseQuery* tiene prevista la posibilidad de que se produzca una excepción durante su respuesta; en este caso, tampoco se cierra la ventana. Así, es posible evitar que se cierre el cuadro de diálogo si falla la llamada a *Post*. El algoritmo necesario se puede parametrizar y colocar en una función que puede ser llamada desde cualquier diálogo de entrada de datos:

```
function PuedoCerrar(AForm: TForm; ATable: TDataSet): Boolean;
begin
  Result := True;
  if AForm.ModalResult = mrOk then
    ATable.Post
  else if not ATable.Modified
    or (Application.MessageBox('¿Desea abandonar los cambios?',
      'Atención', MB_ICONQUESTION + MB_YESNO) = IdYes) then
    ATable.Cancel
  else
    Result := False;
end;
```

La llamada típica a esta función es como sigue:

```
procedure TDialogoEdicion.FormCloseQuery(Sender: TObject;
   var CanClose: Boolean);
begin
   CanClose := PuedoCerrar(Self, Table1);
end;
```

Esta función, *PuedoCerrar*, desempeñará un papel importante en este libro. A partir de este momento desarrollaremos variantes de la misma para aprovechar las diversas técnicas (transacciones, actualizaciones en caché) que vayamos estudiando.

> La función *PuedoCerrar* puede definirse también como un método protegido en un formulario de prototipo. Este formulario puede utilizarse en el proyecto para que el resto de los diálogos de entrada de datos hereden de él, mediante la herencia visual.

Entrada de datos continua

Muchas aplicaciones están tan orientadas a la entrada de datos que, en aras de la facilidad de uso, no es conveniente tener que estar invocando una y otra vez la ficha de entrada de datos por cada registro a insertar. En este tipo de programas es preferible activar una sola vez el cuadro de diálogo para la entrada de datos, y redefinir el sentido del botón *Aceptar* de modo tal que grabe los datos introducidos por el usuario y vuelva a preparar inmediatamente las condiciones para insertar un nuevo registro.

Es muy fácil modificar un cuadro de diálogo semejante a los desarrollados en la sección anterior, para que adopte el nuevo estilo de interacción. Basta con asignar *mrNone* a la propiedad *ModalResult* del botón *Aceptar*; de este modo, pulsar el botón no implica cerrar el cuadro de diálogo. En compensación, hay que teclear el siguiente código en la pulsación de dicho botón:

```
procedure TForm1.bnOkClick(Sender: TObject);
begin
   Table1.Post;
   Table1.Append;
end;
```

Pero mi solución favorita, que nos permite utilizar menos eventos y teclear menos, es modificar el método *PuedoCerrar*, añadiéndole un parámetro que active o desactive la entrada de datos continua:

```
function PuedoCerrar(AForm: TForm; ATable: TDataSet;
   ModoContinuo: Boolean): Boolean;
var
   PrevState: TDataSetState;
```

```
begin
   Result := True;
   if AForm.ModalResult = mrOk then
   begin
      PrevState := ATable.State;
      ATable.Post;
      if ModoContinuo and (PrevState = dsInsert) then
      begin
         ATable.Append;
         Result := False;
      end;
   end
   else if not ATable.Modified
      or (Application.MessageBox('¿Desea abandonar los cambios?',
         'Atención', MB_ICONQUESTION + MB_YESNO) = IdYes) then
      ATable.Cancel
   else
      Result := False;
end;
```

En este caso, la propiedad *ModalResult* del botón *Aceptar* debe seguir siendo *mrOk*. Lo único que varía es la forma de llamar a *PuedoCerrar* durante el evento *OnCloseQuery*:

```
procedure TForm1.FormCloseQuery(Sender: TObject;
   var CanClose: Boolean);
begin
   CanClose := PuedoCerrar(Self, Table1, True);
end;
```

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
   Table1.First;
   while not Table1.EOF do
      if Condicion(Table1) then
         Table1.Delete
         // No se llama a Next en este caso
      else
         Table1.Next;
end;
```

Recuerde que eliminar una fila puede provocar la propagación de borrados, si existen restricciones de integridad referencial definidas de este modo.

Puede ser útil, en ocasiones, el método *EmptyTable*, que elimina todos los registros de una tabla. La tabla no tiene que estar abierta; si lo está, debe haberse abierto con la propiedad *Exclusive* igual a *True*.

Compresión de tablas locales

Una de las omisiones más curiosas de Delphi es la de métodos a nivel de tablas para eliminar registros borrados lógicamente de tablas en formato Paradox y dBase; los desarrolladores de Borland se han mostrado tenaces en mantener a estas funciones alejadas del juego. Se puede, sin embargo, utilizar directamente las funciones equivalentes del API del BDE, siempre y cuando respetemos ciertas reglas. Comenzaré con una función para eliminar los borrados lógicos de dBase:

```
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;
```

Para compilar el procedimiento anterior, es necesario que la unidad en que se encuentre haga referencia a las unidades *DB*, *DBTables* y *BDE*. La función correspondiente para Paradox es la siguiente:

```
procedure PackParadox(const ADatabase, ATable: string);
var
   ADB: TDatabase;
   SaveKC: Boolean;
   PdxStruct: CRTblDesc;
begin
   with TTable.Create(nil) do
   try
      DatabaseName := ADatabase;
      TableName := ATable;
      Exclusive := True;
      Open;
      ADB := Database;
      SaveKC := ADB.KeepConnection;
      ADB.KeepConnection := True;
      try
         Close;
```

```
            FillChar(PdxStruct, SizeOf(PdxStruct), 0);
            StrPCopy(PdxStruct.szTblName, ATable);
            PdxStruct.bPack := True;
            Check(DbiDoRestructure(ADB.Handle, 1, @PdxStruct,
               nil, nil, nil, False));
         finally
            ADB.KeepConnection := SaveKC;
         end;
      finally
         Free;
      end;
   end;
```

En este caso, también debemos hacer mención a la unidad *SysUtils*, debido a la presencia del procedimiento *StrPCopy*.

En ambos procedimientos se intenta abrir la tabla en modo exclusivo, de modo que si los utiliza en una aplicación que tiene todas las tablas abiertas en un módulo de datos, debe cerrar las tablas afectadas antes de comprimirlas.

Otra función de bajo nivel relacionada con la optimización de tablas es *DbiRegenIndexes*, que sirve para actualizar los índices de las tablas dBase, FoxPro y Paradox. Puede ejecutarse mediante esta simple instrucción:

```
Check(DbiRegenIndexes(Table1.Handle));
```

La tabla, sin embargo, debe estar abierta en modo exclusivo.

Capítulo

29

Eventos de transición de estados

LA PARTE VERDADERAMENTE COMPLICADA del diseño y programación con Delphi viene cuando se trata de establecer y forzar el cumplimiento de las reglas de consistencia, o como la moda actual las denomina en inglés: *business rules*. Algunas de estas reglas tienen un planteamiento sencillo, pues se tratan de condiciones que pueden verificarse analizando campos aislados. Esto ya sabemos cómo hacerlo, utilizando propiedades de los campos, como *Required*, *MinValue*, *MaxValue* y *EditMask* o, si la condición de validación es muy complicada, el evento *OnValidate*.

Otras reglas imponen condiciones sobre los valores de varios campos de un mismo registro a la vez; en este capítulo veremos cómo realizar este tipo de validaciones. Y las reglas más complejas requieren la coordinación entre los valores de varios registros de una o más tablas: claves primarias, integridad referencial, valores previamente calculados, etc. Para casi todas estas restricciones, necesitaremos el uso, en Delphi, de los llamados *eventos de transición*, que se disparan automáticamente durante los cambios de estado de los conjuntos de datos. Aprenderemos a utilizar estos eventos tanto para imponer las condiciones correspondientes como para detectar el incumplimiento de las mismas.

Cuando el estado cambia…

Como corresponde a una arquitectura basada en componentes activos, Delphi ofrece una amplia gama de eventos que son disparados por los conjuntos de datos. Ya hemos visto algunos de estos eventos: *OnCalcFields*, para los campos calculados, *OnFilterRecord*, para el filtrado de filas. Pero la mayoría de los eventos producidos por las tablas y consultas se generan cuando se producen cambios de estado en estos componentes. Esta es la lista de los eventos disponibles en Delphi 2:

Método de transición	Eventos generados
Open	*BeforeOpen, AfterOpen*
Close	*BeforeClose, AfterClose*
Edit	*BeforeEdit, OnEditError, AfterEdit*

Método de transición	Eventos generados
Insert	*BeforeInsert, OnNewRecord, AfterInsert*
Post	*BeforePost, OnPostError, AfterPost*
Cancel	*BeforeCancel, AfterCancel*
Delete	*BeforeDelete, OnDeleteError, AfterDelete*

Además de los eventos listados, falta mencionar a los eventos *OnUpdateError* y *OnUpdateRecord*, relacionados con las actualizaciones en caché, y *OnServerYield*, que se puede utilizar únicamente con servidores Sybase. Delphi 3 añade a esta lista los métodos *BeforeScroll* y *AfterScroll*.

> Los eventos *BeforeScroll* y *AfterScroll* son más específicos que el evento *OnDataChange* del componente *TDataSource*. Este último evento, además de dispararse cuando cambia la fila activa, también se genera cuando se realizan cambios en los valores de los campos de la fila activa.

Algunos de los eventos de la lista están relacionados con lo que sucede antes y después de que la tabla ejecute el método de transición; sus nombres comienzan con los prefijos *Before* y *After*. Otros están relacionados con los errores provocados durante este proceso: *OnEditError*, *OnDeleteError* y *OnPostError*; estos eventos, en particular, no existen en Delphi 1. Queda además *OnNewRecord*, que es disparado por la tabla después de entrar en el estado *dsInsert*, y a continuación del cual se limpia el indicador de modificaciones del registro activo. Es natural que, ante tanta variedad de eventos, el programador se sienta abrumado y no sepa por dónde empezar. No obstante, es relativamente sencillo marcarnos una guía acerca de qué eventos tenemos que interceptar de acuerdo a nuestras necesidades.

Reglas de empresa: ¿en el servidor o en el cliente?

Muchas veces, los programadores que conocen algún sistema SQL y que están aprendiendo Delphi se plantean la utilidad de los eventos de transición de estado, si el sistema al que se accede es precisamente un sistema SQL cliente/servidor. El argumento es que la implementación de las reglas de empresa debe tener lugar preferentemente en el servidor, por medio de *triggers* y procedimientos almacenados. La implementación de reglas en el servidor nos permite ahorrar código en los clientes, nos ofrece más seguridad y, en ocasiones, más eficiencia.

Sin embargo, el hecho básico del cual debemos percatarnos es el siguiente: los eventos de transición de Delphi se refieren a *acciones que transcurren durante la edición*, en el lado cliente de la aplicación. Aunque en algunos casos realmente tenemos la libertad de implementar alguna regla utilizando cualquiera de estas dos técnicas, hay ocasio-

nes en que esto no es posible. La siguiente tabla muestra una equivalencia entre los *triggers* y los eventos de transición de estados:

Trigger	Insert	Update	Delete
Antes	*BeforePost*	*BeforePost*	*BeforeDelete*
Después	*AfterPost*	*AfterPost*	*AfterDelete*

Como se aprecia, solamente un pequeño grupo de eventos de transición están representados en esta tabla. No hay equivalente en SQL a los eventos *BeforeEdit*, *AfterCancel* ó *BeforeClose*, por mencionar algunos. Por otra parte, incluso cuando hay equivalencia en el comportamiento, desde el punto de vista del usuario puede haber diferencias. Por ejemplo, como veremos en la siguiente sección, el evento *OnNewRecord* puede utilizarse para asignar valores por omisión a los campos de registros nuevos. Esto mismo puede efectuarse con el *trigger* **before insert**. Sin embargo, si utilizamos *OnNewRecord*, el usuario puede ver durante la edición del registro los valores que van a tomar las columnas implicadas, cosa imposible si se implementa el *trigger*.

Por lo tanto, lo ideal es una mezcla de ambas técnicas, que debe decidirse en base a los requisitos de cada aplicación. Más adelante, cuando estudiemos las actualizaciones en caché, profundizaremos más en este tema.

Inicialización de registros: el evento OnNewRecord

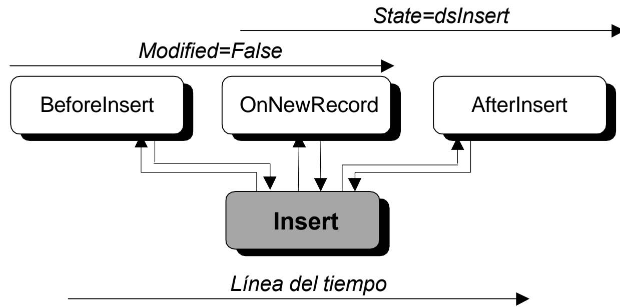
Posiblemente el evento más popular de los conjuntos de datos sea *OnNewRecord*. Aunque su nombre no comienza con *After* ni con *Before*, se produce durante la transición al estado *dsInsert*, cuando ya el conjunto de datos se encuentra en dicho estado, pero antes de disparar *AfterInsert*. Este evento se utiliza para la asignación de valores por omisión durante la inserción de registros.

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 por ejemplo, la tabla *employee.db* de la base de datos *dbdemos*, y visualicémosla 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 para Delphi no hemos modificado nada en el mismo. Recuerde que esta información nos la ofrece la propiedad *Modified* de los conjuntos de datos.



Para notar 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.

> En el capítulo 16 vimos que la propiedad *DefaultExpression* de los campos sirve para asignar automáticamente valores por omisión sencillos a estos componentes. Sin embargo, dadas las limitaciones de las expresiones SQL que son reconocidas por el intérprete local, en muchos casos hay que recurrir también al evento *OnNewRecord* para poder completar la inicialización. Al parecer, existe un *bug* que impide que estos dos mecanismos puedan utilizarse simultáneamente. Por lo tanto, si necesita utilizar *OnNewRecord* en una tabla, es muy importante que limpie todas las propiedades *DefaultExpression* de los campos de dicha tabla. Por lo menos, hasta nuevo aviso.

Validaciones a nivel de registros

En los casos en que se necesita imponer una condición sobre varios campos de un mismo registro, la mejor forma de lograrlo es mediante el evento *BeforePost*. Se verifica la condición y, en el caso en que no se cumpla, se aborta la operación con una excepción, del mismo modo que hacíamos con el evento *OnValidate* de los campos. Hay que tener en cuenta que *BeforePost* se llama igualmente para las inserciones y las modificaciones. Para distinguir entre estas dos situaciones, tenemos que utilizar, por supuesto, la propiedad *State* del conjunto de datos. Por ejemplo, para impedir que un empleado nuevo gane una cantidad superior a cierto límite salarial podemos programar la siguiente respuesta a *BeforePost*:

```
procedure TmodDatos.tbEmpleadosBeforePost(DataSet: TDataSet);
begin
   if tbEmpleados.State = dsInsert then
      if tbEmpleadosSalary.Value > TopeSalarial then
         DatabaseError('Demasiado dinero para un aprendiz');
end;
```

Asumimos que *TopeSalarial* es una constante definida en el módulo, o una variable convenientemente inicializada. El procedimiento *DatabaseError* ya ha sido utilizado en el capítulo sobre acceso a campos, durante la respuesta al evento *OnValidate*.

Este manejador de eventos no impedirá, sin embargo, modificar posteriormente el salario una vez realizada la inserción. Además, hubiera podido implementarse en respuesta al evento *OnValidate* del componente *tbEmpleadosSalary*, pues únicamente verifica los valores almacenados en un solo campo. Una alternativa válida tanto para las altas como para las modificaciones es realizar la validación comprobando la antigüedad del trabajador, fijando un tope para los empleados que llevan menos de un año en la empresa:

```
procedure TmodDatos.tbEmpleadosBeforePost(DataSet: TDataSet);
begin
   if (Date - tbEmpleadosHireDate.Value < 365)
      and (tbEmpleadosSalary.Value > TopeSalarial) then
         DatabaseError('¡Este es un enchufado!');
end;
```

Este criterio de validación no puede implementarse correctamente utilizando sólo los eventos *OnValidate*, pues involucra simultáneamente a más de un campo de la tabla. Observe el uso de la función *Date* (fecha actual), y de la resta entre fechas para calcular la diferencia en días.

Antes y después de una modificación

Cuando simulamos *triggers* en Delphi tropezamos con un inconveniente: no tenemos nada parecido a las variables de contexto *new* y *old*, para obtener los valores nuevos y antiguos del registro que se actualiza. Si necesitamos estos valores debemos almacenarlos manualmente, y para ello podemos utilizar el evento *BeforeEdit*. Luego, podemos aprovechar los valores guardados anteriormente en el evento *AfterPost*, para efectuar acciones similares a las de un *trigger*.

Pongamos por caso que queremos mantener actualizada la columna *ItemsTotal* de la tabla de pedidos cada vez que se produzca un cambio en las líneas de detalles asociadas. Para simplificar nuestra exposición, supondremos que sobre la tabla de detalles se ha definido un campo calculado *SubTotal*, que se calcula multiplicando la cantidad de artículos por el precio extraído de la tabla de artículos y que tiene en cuenta el descuento aplicado. Tenemos que detectar las variaciones en este campo para refle-

jarlas en la columna *ItemsTotal* de la fila activa de pedidos. Asumiremos, por supuesto, que existe una relación *master/detail* entre las tablas de pedidos y detalles.

Como primer paso, necesitamos declarar una variable, *OldSubTotal*, para almacenar el valor de la columna *SubTotal* antes de cada modificación. Esto lo hacemos en la declaración de la clase correspondiente al módulo de datos, en su sección **private** (pues esta declaración no le interesa a nadie más):

```
type
   TmodDatos = class(TDataModule)
      // …
   private
      OldSubTotal: Currency;
   end;
```

Cuando declaramos una variable, debemos preocuparnos inmediatamente por su inicialización. Esta variable, en particular, por estar definida dentro de una clase, se inicializa con 0 al crearse el módulo de datos. Cuando vamos a realizar modificaciones en una línea de detalles, necesitamos el valor anterior de la columna *SubTotal*:

```
procedure TmodDatos.tbLineasBeforeEdit(Sender: TDataSet);
begin
   OldSubTotal := tbLineas['SubTotal'];
end;
```

Como la actualización de *ItemsTotal* se realizará en el evento *AfterPost*, que también se dispara durante las inserciones, es conveniente inicializar también *OldSubTotal* en el evento *BeforeInsert*:

```
procedure TmodDatos.tbLineasBeforeInsert(Sender: TDataSet);
begin
   OldSubTotal := 0;
end;
```

El mismo resultado se obtiene si la inicialización de *OldSubTotal* para las inserciones se realiza en el evento *OnNewRecord*, pero este evento posiblemente tenga más código asociado, y es preferible separar la inicialización de variables de contexto de la asignación de valores por omisión.

Entonces definimos la respuesta al evento *AfterPost* de la tabla de líneas de detalles:

```
procedure TmodDatos.tbLineasAfterPost(Sender: TDataSet);
var
   DiffSubTotal: Currency;
begin
   DiffSubTotal := tbLineas['SubTotal'] - OldSubTotal;
   if DiffSubTotal <> 0 then
   begin
      if not (tbPedidos in dsEditModes) then tbPedidos.Edit;
```

```
            tbPedidos['ItemsTotal'] := tbPedidos['ItemsTotal']
               + DiffSubTotal;
         end;
      end;
```

Como la tabla de pedidos es la tabla maestra de la tabla de detalles, no hay necesidad de localizar el registro del pedido, pues es el registro activo. He decidido no llamar al método *Post* sobre la tabla de pedidos, asumiendo que estamos realizando la edición simultánea de la tabla de pedidos con la de líneas de detalles. En los capítulos sobre transacciones y actualizaciones en caché veremos métodos para automatizar la grabación de datos durante la edición e inserción de objetos complejos.

Le propongo al lector que implemente el código necesario para mantener actualizado *ItemsTotal* durante la eliminación de líneas de detalles.

Propagación de cambios en cascada

Una aplicación útil de los eventos de transición de estados es la propagación en cascada de cambios cuando existen restricciones de integridad referencial entre tablas. Normalmente, los sistemas SQL y Paradox permiten especificar este comportamiento en forma declarativa durante la definición de la base de datos; ya hemos visto cómo hacerlo al estudiar las restricciones de integridad referencial. Pero hay sistemas que no implementan este recurso, o lo implementan parcialmente. En tales casos, tenemos que recurrir a *triggers* o a eventos de transición de estados.

Si estamos programando para tablas dBase anteriores a la versión 7, y queremos establecer la propagación de borrados desde la tabla de pedidos a la de líneas de detalles, podemos crear un manejador de eventos en este estilo:

```
procedure TmodDatos.tbPedidosBeforeDelete(Dataset: TDataSet);
begin
   tbLineas.First;
   while not tbLineas.EOF do
      tbLineas.Delete;
end;
```

Observe que no se utiliza el método *Next* para avanzar a la próxima fila de la tabla dependiente, pues el método *Delete* se encarga de esto automáticamente. Curiosamente, Delphi trae un ejemplo de borrado en cascada, pero el ciclo de borrado se implementa de este modo:

```
while Tabla.RecordCount > 0 do
   // … etcétera …
```

Este código funciona, por supuesto, pero si la tabla pertenece a una base de datos SQL, *RecordCount* es una función peligrosa, como ya hemos explicado.

Si lo que queremos, por el contrario, es prohibir el borrado de pedidos con líneas asociadas, necesitamos esta otra respuesta:

```
procedure TmodDatos.tbPedidosBeforeDelete(Dataset: TDataSet);
begin
   if not tbLineas.IsEmpty then
      DatabaseError('Este pedido tiene líneas asociadas');
end;
```

Incluso si el sistema permite la implementación de los borrados en cascada, puede interesarnos interceptar este evento, para dejar que el usuario decida qué hacer en cada caso. Por ejemplo:

```
procedure TmodDatos.tbPedidosBeforeDelete(Dataset: TDataSet);
begin
   tbLineas.First;
   if not tbLineas.EOF then
      if MessageDlg('¿Eliminar líneas de detalles?', mtConfirmation,
         [mbYes, mbNo], 0) = mrYes then
         repeat
            tbLineas.Delete;
         until tbLineas.EOF
      else
         Abort;
end;
```

Note el uso de *Abort*, la excepción silenciosa, para interrumpir la ejecución de la operación activa y evitar la duplicación innecesaria de mensajes al usuario.

Actualizaciones coordinadas master/detail

He aquí otro caso especial de coordinación mediante eventos de conjuntos de datos, que es aplicable cuando existen pares de tablas en relación *master/detail*. Para precisar el ejemplo, supongamos que son las tablas *orders.db* e *items.db*, del alias *dbdemos*: una tabla de pedidos y su tabla asociada de líneas de detalles. En la mayoría de las ocasiones existe una restricción de integridad referencial definida en la tabla esclava. Por lo tanto, para poder grabar una fila de la tabla de detalles tiene que existir previamente la correspondiente fila de la tabla maestra. Para garantizar la existencia de la fila maestra, podemos dar respuesta al evento *BeforeInsert* de la tabla dependiente:

```
procedure TmodDatos.tbDetallesBeforeInsert(Dataset: TDataSet);
begin
   if tbPedidos.State = dsInsert then
   begin
      tbPedidos.Post;
      tbPedidos.Edit;
   end;
end;
```

En este caso, además, hemos colocado automáticamente la tabla maestra en modo de edición, por si son necesarias más actualizaciones sobre la cabecera, como el mantenimiento de la columna *ItemsTotal* que vimos anteriormente.

Esta misma técnica es la que he recomendado en el capítulo sobre Oracle, en relación con las tablas anidadas. Es recomendable guardar los cambios en la tabla maestra antes de añadir registros a la tabla anidada. También es bueno realizar un *Post* sobre la tabla maestra una vez que se ha añadido el registro en la tabla anidada:

```
procedure TmodDatos.tbAnidadaAfterInsert(Dataset: TDataSet);
begin
   Insertando := True;
end;

procedure TmodDatos.tbAnidadaAfterEdit(Dataset: TDataSet);
begin
   Insertando := False;
end;

procedure TmodDatos.tbAnidadaAfterPost(Dataset: TDataSet);
begin
   if Insertando then
      tbMaestra.CheckBrowseMode;
end;
```

En este ejemplo, la variable lógica *Insertando* debe haber sido definida en la sección **private** del módulo de datos.

Antes y después de la apertura de una tabla

Puede ser útil especificar acciones asociadas a la apertura y cierre de tablas. Si nuestra aplicación trabaja con muchas tablas, es conveniente que éstas se abran y cierren por demanda; si estas tablas representan objetos complejos, es posible expresar las dependencias entre tablas en los eventos *Before* y *AfterOpen*, y *Before* y *AfterClose*. En el ejemplo de la entrada de pedidos, la tabla de pedidos, *tbPedidos*, funciona en coordinación con la tabla de líneas de detalles, *tbLineas*. Supongamos, por un momento, que necesitamos también una tabla para resolver las referencias a clientes, *tbRefClientes*, y otra para las referencias a artículos, *tbRefArticulos*. Podemos entonces programar los siguientes métodos como respuesta a los eventos *BeforeOpen* y *AfterClose* de la tabla de pedidos:

```
procedure TmodDatos.tbPedidosBeforeOpen(DataSet: TDataSet);
begin
   tbRefClientes.Open;
   tbRefArticulos.Open;
   tbLineas.Open;
end;
```

```
procedure TmodDatos.tbPedidosAfterClose(DataSet: TDataSet);
begin
   tbLineas.Close;
   tbRefArticulos.Close;
   tbRefClientes.Close;
end;
```

De esta manera, las tres tablas dependientes pueden estar cerradas durante el diseño y carga de la aplicación, y ser activadas por demanda, en el momento en que se abre la tabla de pedidos.

El ejemplo que he utilizado quizás no sea el mejor. Delphi abre automáticamente las tablas de referencia cuando abre una tabla que tiene campos de este tipo. Sin embargo, al cerrar la tabla maestra, no cierra automáticamente las tablas de referencia.

Vaciando los buffers

Paradox, dBase y Access tienen un grave problema: si se cae el sistema después de una actualización, pueden estropearse las tablas irreversiblemente. Las modificaciones realizadas por una aplicación se guardan en *buffers* internos del BDE, desde donde son transferidas al disco durante los intervalos en que la aplicación está ociosa. Un corte de tensión, por ejemplo, puede dejar una modificación confirmada en un fichero índice, pero no en el fichero de datos principal, lo cual puede ser mortal en ocasiones.

La mejor cura consiste en transferir los *buffers* modificados en cuanto podamos, aún al coste de ralentizar la aplicación. Delphi ofrece el siguiente método, aplicable a los conjuntos de datos del BDE:

```
procedure TBDEDataSet.FlushBuffers;
```

El mayor grado de seguridad se obtiene cuando llamamos a *FlushBuffers* inmediatamente después de cada modificación. Traducido al cristiano: en los eventos *AfterPost* y *AfterDelete* de las tablas y consultas actualizables:

```
procedure TmodDatos.Table1AfterPost(Sender: TObject);
begin
   (Sender as TDBDataSet).FlushBuffers;
end;
```

Por supuesto, existen soluciones intermedias, como vaciar los *buffers* solamente después de terminar transacciones largas.

Los eventos de detección de errores

La política de control de errores de Delphi 1 estaba basada completamente en las excepciones, aplicando a rajatabla la Tercera Regla de Marteens:

"Escriba la menor cantidad posible de instrucciones **try...except***"*

Es decir, si se producía un error durante la grabación de un registro, se producía una excepción que iba abortando las funciones pendientes en la pila de ejecución del programa, y que terminaba su andadura en la instrucción de captura del ciclo de mensajes. Un cuadro de mensajes mostraba al usuario lo errado de sus planteamientos, y ¡vuelta a intentar la operación!

Bajo esta estrategia, ¿cómo mostrar mensajes más específicos? No nos interesa que el usuario de nuestro programa vea mensajes como: "Violación de unicidad"[24]. Nos interesa que el mensaje diga ahora "Código de cliente repetido", y que en otra circunstancia diga "Código de artículo ya utilizado". La única posibilidad que nos dejaba Delphi 1 era encerrar en instrucciones **try...except** todas las llamadas que pudieran fallar, y esto no es siempre posible pues algunos métodos, como *Post*, son llamados implícitamente por otras operaciones.

En Delphi 2 se introdujeron los llamados *eventos de detección de errores*, que se disparan cuando se produce un error en alguna operación de un conjunto de datos, pero antes de que se eleve la excepción asociada:

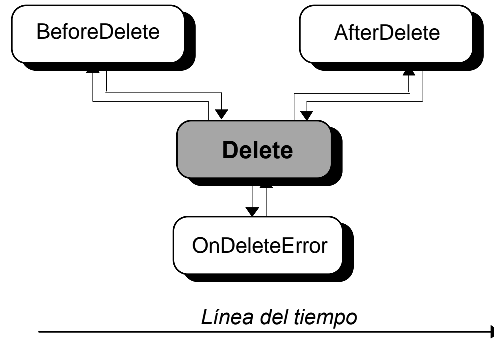
```
property OnEditError: TDataSetErrorEvent;
property OnPostError: TDataSetErrorEvent;
property OnDeleteError: TDataSetErrorEvent;
```

La declaración del tipo de estos eventos es la siguiente:

```
type
   TDataSetErrorEvent = procedure (DataSet: TDataSet;
      E: EDatabaseError; var Action: TDataAction) of object;
```

Estos eventos, cuando se disparan, lo hacen durante el "núcleo" de la operación asociada, después de que ha ocurrido el evento *Before*, pero antes de que ocurra el evento *After*. El siguiente diagrama representa el flujo de eventos durante la llamada al método *Delete*. Similar a este diagrama son los correspondientes a los métodos *Edit* y *Post*:

[24] Suponiendo que los mensajes de error están traducidos, y no obtengamos: *"Key violation"*



Dentro del manejador del evento, podemos intentar corregir la situación que produjo el error, e indicar que la operación se reintente; para esto hay que asignar *daRetry* al parámetro *DataAction*. Podemos también dejar que la operación falle, y que Delphi muestre el mensaje de error; para esto se asigna *daFail* a *DataAction*, que es el valor inicial de este parámetro. Por último, podemos elegir mostrar el mensaje dentro del manejador y que Delphi luego aborte la operación sin mostrar mensajes adicionales; esto se logra asignando *daAbort* al parámetro *DataAction*.

El ejemplo más sencillo de respuesta a un evento de detección de errores es la implementación de un ciclo de reintentos infinito, sin intervención del usuario, cuando se produce un error de bloqueo. Como estudiaremos en el capítulo sobre control de concurrencia, cuando se produce una situación tal, se dispara el evento *OnEditError*. Una posible respuesta es la siguiente:

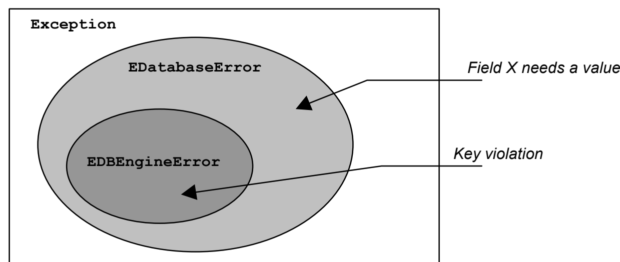
```
procedure TmodDatos.Table1EditError(DataSet: TDataSet;
   E: EDatabaseError; var Action: TDataAction);
var
   T: Integer;
begin
   // Calcular un momento de 0.5 a 1 segundo después de ahora
   T := GetTickCount + 500 + Random(500);
   // Esperar a ese momento
   while GetTickCount < T do
      Application.ProcessMessages;
   // Reintentar
   Action := daRetry;
end;
```

En este ejemplo, el reintento se produce después de una espera aleatoria que varía desde medio segundo hasta un segundo: de este modo se logra un tráfico menor en la red si se produce una colisión múltiple entre aplicaciones. Recuerde que la función *GetTickCount* devuelve el número de milisegundos transcurridos desde el arranque de Windows.

La estructura de la excepción EDBEngineError

En el caso del evento *OnEditError*, era relativamente fácil crear un manejador para el evento, pues los errores de edición tienen en su mayor parte una única causa: un bloqueo no concedido, en el caso de Paradox y dBase, y un registro eliminado por otro usuario, en cliente/servidor. Pero en el caso de los errores producidos por *Post* y *Delete*, las causas del error pueden variadas. En este caso, tenemos que aprovechar la información que nos pasan en el parámetro *E*, que es la excepción que está a punto de generar el sistema. Y hay que aprender a interpretar el tipo *EDatabaseError*.

La clase de excepción *EDatabaseError* es la clase base de la jerarquía de excepciones de base de datos. Sin embargo, aunque existen algunos errores que se reportan directamente mediante esta excepción, la mayoría de los errores de bases de datos disparan una excepción de tipo *EDBEngineError*. La causa es sencilla: los errores del Motor de Datos pueden haber sido generados por el propio motor, pero también por un servidor SQL. Debemos conocer, entonces, el error original del servidor, pero también la interpretación del mismo que hace el BDE. La excepción generada por el BDE, *EDBEngineError*, tiene una estructura capaz de contener esta lista de errores. Por regla, si el error de base de datos es detectado por el BDE se lanza una excepción *EDBEngineError*; si el error es detectado por un componente de la VCL, se lanza un *EDatabaseError*.



Dentro del código fuente de la VCL, encontraremos a menudo una función *Check* definida en la unidad *DBTables*, cuyo objetivo es transformar un error del BDE en una excepción *EDBEngineError*. Por ejemplo:

```
procedure TDatabase.StartTransaction;
var
   TransHandle:  HDBIXAct;
begin
   CheckActive;
   if not IsSQLBased and (TransIsolation <> tiDirtyRead) then
      DatabaseError(SLocalTransDirty, Self);
   Check(DbiBeginTran(FHandle, EXILType(FTransIsolation),
      TransHandle));
end;
```

Observe que la llamada a la función del API del BDE se realiza dentro de *Check*. De este modo, el resultado de *DbiBeginTran* es analizado por esta función: si es cero, no pasa nada, pero en caso contrario, se dispara la excepción. Observe también en la línea anterior cómo se llama a una función *DatabaseError* cuando Delphi detecta por sí mismo una condición errónea. Esta función es la encargada de disparar las excepciones *EDatabaseError*. Existe también una versión *DatabaseErrorFmt*, en la que se permite pasar cadenas de formato, al estilo de la función *Format*.

Un objeto de la clase *EDBEngineError* tiene, además de las propiedades heredadas de *EDatabaseError*, las siguientes propiedades:

```
property ErrorCount: Integer;
property Errors[Index: Integer]: TDBError;
```

El tipo *TDBError* corresponde a cada error individual. He aquí las propiedades disponibles para esta clase:

Propiedad	Significado
ErrorCode	El código del error, de acuerdo al BDE
Category	Categoría a la que pertenece el error
SubCode	Subcódigo del error
NativeError	Si el error es producido por el servidor, su código de error
Message	Cadena con el mensaje asociado al error.

NativeError corresponde al código *SqlCode* propio de cada servidor SQL. De todas estas propiedades, la más útil es *ErrorCode*, que es el código de error devuelto por el motor de datos. ¿Cómo trabajar con estos códigos? Delphi trae un ejemplo, *dberrors*, en el cual se explica cómo aprovechar los eventos de detección de errores. En el programa de demostración se utiliza un método bastante barroco para formar el código de error a partir de una dirección base y de unos valores numéricos bastante misteriosos. Curiosamente, todo este revuelo es innecesario, pues en la unidad *BDE* estos códigos de error ya vienen declarados como constantes que podemos utilizar directamente.

Para profundizar un poco más en el sistema de notificación de errores, cree una aplicación sencilla con una tabla, una rejilla y una barra de navegación. Luego asigne el siguiente manejador compartido por los tres eventos de errores, los correspondientes a *Post*, *Edit* y *Delete*:

```
procedure TForm1.Table1PostError(DataSet: TDataSet;
   E: EDatabaseError; var Action: TDataAction);
var
   S: string;
   I: Integer;
```

```
begin
   if E is EDBEngineError then
   begin
      with EDBEngineError(E) do
         for I := 0 to ErrorCount - 1 do
         begin
            if S <> '' then AppendStr(S, #13);
            with Errors[I] do
               AppendStr(S, Format('%.4x (%d): %s',
                  [ErrorCode, NativeError, Message]));
         end;
      DatabaseError(S);
   end;
end;
```

Por cada error que aparece en la lista, añadimos una línea al mensaje que estamos componiendo, mostrando el valor de *ErrorCode* en hexadecimal (pronto veremos por qué), el valor de *NativeError* y el mensaje específico de ese error. Cuando tenemos el mensaje, lanzamos una excepción de clase *EDatabaseError*. Dicho en otras palabras: no dejamos reaccionar al manejador del evento. Esta técnica es correcta, y se emplea con frecuencia cuando se quiere cambiar sencillamente el mensaje de error que se mostrará al usuario.

Ahora de lo que se trata es de hacer fallar a la aplicación por todos las causas que podamos, para ver cómo protesta Delphi. Por ejemplo, cuando se produce una violación de unicidad, éste es el resultado que he obtenido, haciendo funcionar a la aplicación contra una tabla de Paradox:

```
2601 (0): Key violation
```

El cero en *NativeError* indica que no es un error del servidor, pues no hay servidor en este caso. ¿Se ha dado cuenta de que he ignorado las dos propiedades *Category* y *SubCode* de los errores? Es que en este caso, la primera vale *$26*, y la segunda *$01*. El signo de dólar no significa que sean precios, sino que son valores hexadecimales. Con estos dos datos, usted puede bucear en la interfaz de la unidad *BDE*, para encontrar que corresponden a la constante *DBIERR_KEYVIOL*.

Sin embargo, si conecto la tabla a InterBase, éste es el error que obtengo:

```
2601 (0): Key violation
3303 (-803): Violation of PRIMARY or UNIQUE key constraint
"INTEG_56" on table "PARTS"
```

Primero el BDE ha detectado el error *-803* del servidor. ¿Recuerda que en el capítulo sobre *triggers* y procedimientos almacenados, al hablar de las excepciones mencionaba la variable **sqlcode**? Pues este código negativo corresponde al valor asumido por esta variable después del error. El problema es que no existe un estándar para los códigos de errores nativos en SQL, pero el SQL Link del BDE amablemente interpreta el error por nosotros, convirtiéndolo en nuestro conocido *$2601*: *"Key violation"*, que es

lo que vería el usuario. Hay una regla que se puede inferir a partir de este ejemplo y otros similares: los códigos nativos del servidor vienen acompañando al *ErrorCode $3303*, correspondiente a la constante simbólica *DBIERR_UNKNOWNSQL*.

Hagamos fallar una vez más al BDE. Esta vez he puesto una restricción en la propiedad *Constraints* de la tabla, y he intentado modificar un registro con valores incorrectos. Esto es lo que he obtenido:

```
2EC4 (0): Constraint failed. Expression:
2EAE (0): La cantidad de pedidos debe ser positiva.
```

Esta vez tenemos dos errores, y ninguno de ellos proviene del servidor, pues se trata de una restricción a verificar en el lado cliente. El primer error corresponde a la constante *DBIERR_USERCONSTRERR* (parece una maldición polaca). Y el segundo es *DBIERR_CONSTRAINTFAILED*.

Aplicaciones de los eventos de errores

¿Qué podemos hacer en un evento de detección de errores? Hemos visto que en algunos casos se podía reintentar la operación, después de efectuar algunas correcciones, o de esperar un tiempo. Sin embargo, en la mayoría de los casos, no puede hacerse mucho. Tendremos que esperar al estudio de las actualizaciones en caché para disponer de herramientas más potentes, que nos permiten recuperarnos con efectividad de todo un rango de errores.

Sin embargo, la mayor aplicación de estos eventos es la *contextualización* de los mensajes de errores. Soy consciente de que acabo de escribir una palabra de 17 letras, ¿no debía haber dicho mejor *traducción*? Resulta que no. Un usuario intenta insertar un nuevo cliente, pero le asigna un nombre o un código ya existente. Delphi le responde enfadado: *"Key violation"*. Pero usted servicialmente traduce el mensaje: *"Violación de clave"*. Y a la mente del usuario, que no conoce a Codd y la historia de su perro, viene una serie desconcertante de asociaciones de ideas. Lo que teníamos que haber notificado era: *"Código o nombre de cliente ya existe"*[25]. Es decir, teníamos que adecuar el mensaje general a la situación particular en que se produjo.

Pero antes tendremos que simplificar un poco la forma de averiguar qué error se ha producido. La siguiente función nos va a ayudar en ese sentido, "aplanando" las excepciones de tipo *EDBEngineError* y produciendo un solo código de error que resuma toda la información, o la parte más importante de la misma:

---

[25] Y yo me pregunto: ¿por qué la mayoría de los mensajes en inglés y castellano prescinden de los artículos, las preposiciones y, en algunos casos, de los tiempos verbales? Yo Tarzán, tu Juana...

```
function GetBDEError(E: EDatabaseError): Integer;
var
   I: Integer;
begin
   if E is EDBEngineError then
      with EDBEngineError(E) do
         for I := 0 to ErrorCount - 1 do
         begin
            Result := Errors[I].ErrorCode;
            if Result <> DBIERR_USERCONSTRERR then
               if Result = DBIERR_CONSTRAINTFAILED then
                  DatabaseError(Errors[I].Message)
               else if Errors[I].NativeError = 0 then
                  Exit;
         end;
   Result := -1;
end;
```

Si la excepción no es del BDE, la función devuelve -1, es decir, nada concreto. En caso contrario, husmeamos dentro de *Errors* buscando un error con el código nativo igual a cero: una interpretación del BDE. Pero cuidado, pues si el error era la maldición polaca nos quedamos quietos; en tal caso estamos ante la primera línea de error del fallo de una restricción cliente, que no nos dice nada. Esperamos entonces a llegar al segundo error (*DBIERR_CONSTRAINTFAILED*) y levantamos la excepción inmediatamente, pues la propiedad *Message* contiene el mensaje de error creado por el programador en su idioma materno. En caso contrario, terminamos la función dejando como resultado ese primer error del BDE correspondiente a la parte cliente de la aplicación.

Ahora las aplicaciones de esta técnica. El siguiente método muestra cómo detectar los errores que se producen al borrar una fila de la cual dependen otras gracias a una restricción de integridad referencial. A diferencia de los ejemplos que hemos visto antes, este es un caso de detección de errores *a posteriori*:

```
procedure TmodDatos.tbPedidosDeleteError(DataSet: TDataSet;
   E: EDatabaseError; var Action: TDataAction);
begin
   if GetBDEError(E) = DBIERR_DETAILRECORDSEXIST then
      if MessageDlg('¿Eliminar también las líneas de detalles?',
         mtConfirmation, [mbYes, mbNo], 0) = mrYes then
      begin
         tbLineas.First;
         while not tbLineas.EOF do
            tbLineas.Delete;
         // Reintentar el borrado
         Action := daRetry;
      end
      else
         // Fallar sin mostrar otro mensaje
         Action := daAbort;
end;
```

Solamente hemos intentado solucionar el error cuando el código de error generado es *DBIERR_DETAILRECORDSEXIST*, que hemos encontrado en la unidad *BDE*.

Como último ejemplo, programaremos una respuesta que puede ser compartida por varias tablas en sus eventos *OnPostError*, y que se ocupa de la traducción genérica de los mensajes de excepción más comunes:

```
procedure TmodDatos.ErrorDeGrabacion(DataSet: TDataSet;
   E: EDatabaseError; var Action: TDataAction);
begin
   case GetBDEError(E) of
      DBIERR_KEYVIOL:
         DatabaseError('Clave repetida en la tabla ' +
            TTable(DataSet).TableName);
      DBIERR_FOREIGNKEYERR:
         DatabaseError('Error en clave externa. Tabla: ' +
            TTable(DataSet).TableName);
   end;
end;
```

Observe que si *GetBDEError* no encuentra un código apropiado, este manejador de evento no hace nada, y el programa dispara la excepción original. Si el error ha sido provocado por el fallo de una restricción en el lado cliente, la función *GetBDEError* dispara una nueva excepción con el mensaje personalizado introducido por el programador. Y si se trata de una violación de clave primaria, o de una integridad referencial, al menos se da un mensaje en castellano.

Una vez más, la orientación a objetos…

Los eventos de detección de errores nos muestran la forma correcta de entender la Programación Orientada a Objetos. Tomemos por caso la detección de los errores de violación de unicidad. Este es un error que se produce durante la ejecución del método *Post*. Por lo tanto, pudiéramos encerrar las llamadas al método *Post* dentro de instrucciones **try…except**, y en cada caso tratar la excepción correspondiente. Este estilo de programación se orienta a la *operación*, más bien que al *objeto*, y nos fuerza a repetir el código de tratamiento de excepciones en cada caso en que la operación se emplea. Incluso, lo tendremos difícil si, como es el caso, la operación puede también producirse de forma implícita.

En cambio, utilizando los eventos de detección de errores, especificamos el comportamiento ante la situación de error una sola vez, asociando el código al objeto. De este modo, sea cual sea la forma en que se ejecute el método *Post*, las excepciones son tratadas como deseamos.

Capítulo

30

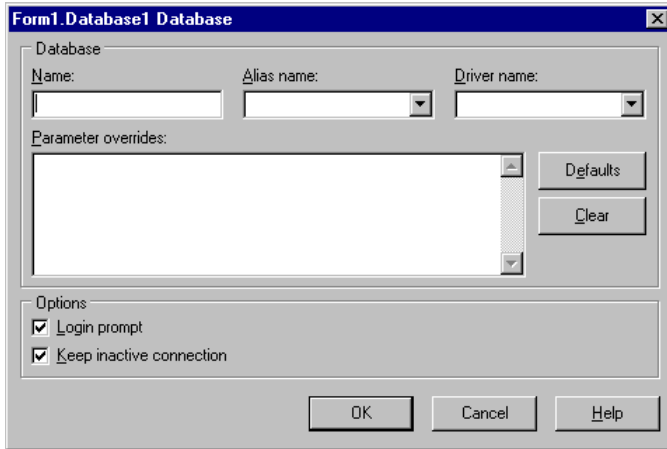
Bases de datos y sesiones

EN LA JERARQUÍA DE OBJETOS manejada por el BDE, las bases de datos y las sesiones ocupan los puestos más altos. En este capítulo estudiaremos la forma en que los componentes *TDatabase* controlan las conexiones a las bases de datos, dejando para el capítulo siguiente el trabajo con transacciones. También veremos las posibilidades que nos ofrece el uso de los componentes de la clase *TSession*.

El componente TDatabase

Los componentes *TDatabase* de Delphi representan y administran las conexiones del BDE a sus bases de datos. Por ejemplo, este componente lleva la cuenta de las tablas y consultas activas en un instante determinado. Recíprocamente, las tablas y consultas están conectadas en tiempo de ejecución a un objeto *TDatabase*, que puede haber sido definido explícitamente por el programador, utilizando en tiempo de diseño el componente *TDatabase* de la paleta de componentes, o haber sido creado implícitamente por Delphi en tiempo de ejecución. Para saber si una base de datos determinada ha sido creada por el programador en tiempo de diseño o es una base de datos temporal creada por Delphi, tenemos la propiedad de tiempo de ejecución *Temporary*, en la clase *TDatabase*. Con las bases de datos se produce la misma situación que con los componentes de acceso a campos: que pueden definirse en tiempo de diseño o crearse en tiempo de ejecución con propiedades por omisión. Como veremos, las diferencias entre componentes *TDatabase* persistentes y dinámicos son mayores que las existentes entre ambos tipos de componentes de campos.

Las propiedades de un objeto *TDatabase* pueden editarse también mediante un cuadro de diálogo que aparece al realizar una doble pulsación sobre el componente:



Un objeto *TDatabase* creado explícitamente define siempre un alias local a la sesión a la cual pertenece. Básicamente, existen dos formas de configurar tal conexión:

- 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.
- 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.

Haya sido creado por Delphi en tiempo de ejecución, o por el programador en tiempo de diseño, un objeto de base de datos nos sirve para:

- Modificar los parámetros de conexión de la base de datos: contraseñas, conexiones establecidas, conjuntos de datos activos, etc.
- Controlar transacciones y actualizaciones en caché.

El control de transacciones se trata en el próximo capítulo, y las actualizaciones en caché se dejan para más adelante.

Objetos de bases de datos persistentes

Comenzaremos con los objetos de bases de datos que el programador incluye en tiempo de diseño. La propiedad fundamental de estos objetos es *DatabaseName*, que corresponde al cuadro de edición *Name* del editor de propiedades. El valor alma-

cenado en *DatabaseName* se utiliza para definir un alias local a la aplicación. La forma en que se define este alias local depende de cuál de las dos propiedades, *AliasName* ó *DriverName*, sea utilizada por el programador. *AliasName* y *DriverName* son propiedades de uso mutuamente excluyente: si se le asigna algo a una, desaparece el valor almacenado en la otra. En este sentido se parecen al par *IndexName* e *IndexFieldNames* de las tablas. O al Yang y el Yin de los taoístas.

Si utilizamos *AliasName* estaremos definiendo un alias basado en otro alias existente. El objeto de base de datos puede utilizarse entonces para controlar las tablas pertenecientes al alias original. ¿Qué sentido tiene esto? La respuesta es que es posible modificar los parámetros de conexión del alias original. Esto quiere decir que podemos añadir parámetros nuevos en la propiedad *Params*. Esta propiedad, declarada de tipo *TStrings*, está inicialmente vacía para los objetos *TDatabase*. Se puede, por ejemplo, modificar un parámetro de conexión existente:

```
SQLPASSTHRUMODE=NOT SHARED
ENABLE SCHEMA CACHE=TRUE
```

El último parámetro permite acelerar las operaciones de apertura de tablas, y puede activarse cuando la aplicación no modifica dinámicamente el esquema relacional de la base de datos creando, destruyendo o modificando la estructura de las tablas. El significado del parámetro *SQLPASSTHRU MODE* lo estudiaremos en el capítulo sobre transacciones y control de concurrencia. Otro motivo para utilizar un alias local que se superponga sobre un alias persistente es la intercepción del evento de conexión a la base de datos (*login*). Pero muchas veces los programadores utilizan el componente *TDatabase* sólo para declarar una variable de este tipo que controle a las tablas pertinentes. Si está utilizando esta técnica con este único propósito, existen mejores opciones, como veremos dentro de poco.

La otra posibilidad es utilizar *DriverName*. ¿Recuerda cómo se define un alias con la configuración del BDE? Es el mismo proceso: *DatabaseName* indica el nombre del nuevo alias, mientras que *DriverName* especifica qué controlador, de los disponibles, queremos utilizar. Para configurar correctamente el alias, hay que introducir los parámetros requeridos por el controlador, y para esto utilizamos también la propiedad *Params*. De este modo, no necesitamos configurar alias persistentes para acceder a una base de datos desde un programa escrito en Delphi.

Cambiando un alias dinámicamente

Un buen ejemplo de aplicación que necesita utilizar alias locales, o de sesión, es aquella que, trabajando con tablas locales en formato Paradox o dBase, necesite cambiar periódicamente de directorio de trabajo. Por ejemplo, ciertas aplicaciones de contabilidad y gestión están diseñadas para trabajar con diferentes ejercicios o em-

presas, cuyas tablas se almacenan en distintos directorios del ordenador. Si se utilizan alias persistentes, es engorroso hacer uso de la utilidad de configuración del BDE, o de los métodos del componente *TSession* para definir o redefinir un alias persistente cada vez que tenemos que cambiar el conjunto de tablas con las cuales se trabaja.

Sin embargo, es relativamente fácil lograr este resultado si las tablas de la aplicación se conectan a un objeto *TDatabase* definido en tiempo de diseño, y este objeto define un alias local a la aplicación. Supongamos que la aplicación contiene, posiblemente en el módulo de datos, un objeto de tipo *TDatabase* con las siguientes propiedades:

Propiedad	Valor
Name	*Database1*
DatabaseName	*MiBD*
DriverName	*STANDARD*

Las tablas del módulo de datos, por supuesto, se conectarán por medio del alias *MiBD*, definido por este objeto. Para simplificar, supondré que dentro del módulo de datos solamente se ha colocado una tabla, *Table1*. Se puede asignar algún directorio inicial en la propiedad *Params* del componente *TDatabase*, incluyendo una línea con el siguiente formato:

```
PATH=C:\Archivos de programa\Contabilidad
```

El cambio de directorio debe producirse a petición del usuario de la aplicación; tras cada cambio, debe quedar grabado el camino al nuevo directorio dentro de un fichero de extensión *ini*. El siguiente método se encarga de cerrar la base de datos, cambiar el valor del parámetro *PATH* y reabrir la tabla, conectando de este modo la base de datos. Si todo va bien, se graba el nuevo directorio en un fichero de configuración de extensión *ini*:

```
procedure TForm1.NuevoDirectorio(const ADir: string);
begin
   DataModule1.Database1.Close;
   DataModule1.Database1.Params.Values['PATH'] := ADir;
   DataModule1.Table1.Open;   // Conecta también la BD
   with TIniFile.Create(ChangeFileExt(ParamStr(0), '.INI')) do
   try
      WriteString('Database', 'Path', ADir);
   finally
      Free;
   end;
end;
```

Posiblemente, el método anterior se utilice después de que el usuario elija el directorio de trabajo mediante un cuadro de diálogo apropiado. Durante la carga del formulario se llama a *NuevoDirectorio* para utilizar el último directorio asignado, que debe encontrarse en el fichero de inicialización:

```
procedure TForm1.FormCreate(Sender: TObject);
var
   S: string;
begin
   with TIniFile.Create(ChangeFileExt(ParamStr(0), '.INI')) do
   try
      S := ReadString('Database', 'Path', '');
   finally
      Free;
   end;
   if S <> '' then NuevoDirectorio(S);
end;
```

Esta misma técnica puede aplicarse a otros tipos de controladores de bases de datos.

Bases de datos y conjuntos de datos

Es posible, en tiempo de ejecución, conocer a qué base de datos está conectado cierto conjunto de datos, y qué conjuntos de datos están activos y conectados a cierta base de datos. La clase *TDatabase* define las siguientes propiedades para este propósito:

```
property DatasetCount: Integer;
property Datasets[I: Integer]: TDataset;
```

Por ejemplo, se puede saber si alguno de los conjuntos de datos conectados a una base de datos contiene modificaciones en sus campos, sin que se le haya aplicado la operación *Post*:

```
function HayModificaciones(ADatabase: TDatabase): Boolean;
var
   I: Integer;
begin
   I := ADatabase.DatasetCount - 1;
   while (I >= 0) and not ADatabase.Datasets[I].Modified do Dec(I);
   Result := I >= 0;
end;
```

En la sección anterior definimos un método *NuevoDirectorio* que cerraba una base de datos estándar, cambiaba su directorio asociado y volvía a abrirla, abriendo una tabla conectada a la misma. Ahora estamos en condiciones de generalizar este algoritmo, recordando qué conjuntos de datos estaban abiertos antes de cerrar la conexión para restaurarlos más adelante:

```
procedure TForm1.NuevoDirectorio(ADB: TDatabase;
   const ADir: string);
var
   I: Integer;
   Lista: TList;
begin
```

```
    Lista := TList.Create;
    try
       // Recordar qué conjuntos de datos estaban abiertos
       for I := 0 to ADB.DataSetCount - 1 do
          Lista.Add(ADB.DataSets[I]);
       // Cambiar el directorio
       ADB.Close;
       ADB.Params.Values['PATH'] := ADir;
       ADB.Open;
       // Reabrir los conjuntos de datos
       for I := 0 to ADB.DataSetCount - 1 do
          TDataSet(Lista[I]).Open;
    finally
       Lista.Free;
    end;
    with TIniFile.Create(ChangeFileExt(ParamStr(0), '.INI')) do
    try
       WriteString('Database', 'Path', ADir);
    finally
       Free;
    end;
 end;
```

Por otra parte, un conjunto de datos activo tiene una referencia a su base de datos por medio de la propiedad *Database*. Antes mencionaba el hecho de que muchos programadores utilizan un componente *TDatabase* que definen sobre un alias persistente con el único propósito de tener acceso al objeto de bases de datos al que se conectan las tablas. La alternativa, mucho más eficiente, es declarar una variable de tipo *TDatabase* en la sección pública de declaraciones del módulo de datos, o en algún otro sitio conveniente, e inicializarla durante la creación del módulo:

```
type
   TDataModule1 = class(TDataModule)
      // …
   public
      Database: TDatabase;
   end;

implementation

procedure TDataModule1.DataModuleCreate(Sender: TObject);
begin
   Database := Table1.Database;
   Database.TransIsolation := tiDirtyRead;
end;
```

La asignación realizada sobre la propiedad *TransIsolation* es bastante frecuente cuando se trata con tablas Paradox y dBase, y se quieren utilizar transacciones locales. Esto lo explicaremos en el siguiente capítulo.

Parámetros de conexión

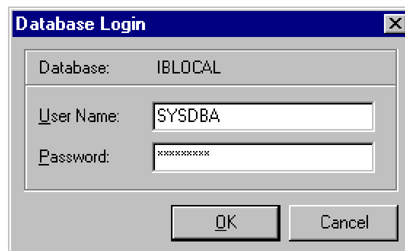
Hay aplicaciones que comienzan con todas sus tablas abiertas, y solamente las cierran al finalizar. Pero hay otras que van abriendo y cerrando tablas según sea necesario. ¿Qué debe suceder cuando todas las tablas han sido cerradas? Todo depende de la propiedad *KeepConnections*, del componente *TDatabase* al cual se asocian las tablas. Esta propiedad vale *True* por omisión, lo cual quiere decir que una vez establecida la conexión, ésta se mantiene aunque se cierren posteriormente todas las tablas. Si por el contrario, *KeepConnections* vale *False*, al cerrarse el último conjunto de datos activo de la base de datos, se desconecta la base de datos.

El problema es que una conexión a una base de datos consume recursos, especialmente si la base de datos se encuentra en un servidor. Típicamente, por cada usuario conectado, el software servidor debe asignar un proceso o hilo (*thread*), junto con memoria local para ese proceso. Así que en ciertos casos es conveniente que, al cerrarse la última tabla, se desconecte también la base de datos. Pero también sucede que el restablecimiento de la conexión es costoso, y si tenemos una base de datos protegida por contraseñas, el proceso de reapertura no es transparente para el usuario (a no ser que tomemos medidas). Por lo tanto, depende de usted lograr un buen balance entre estas dos alternativas.

La petición de contraseñas

Cuando se intenta establecer la conexión de un objeto *TDatabase* a su correspondiente base de datos, si la propiedad *LoginPrompt* del objeto es *True* y se trata de una base de datos SQL, el Motor de Datos debe pedir, de un modo u otro, la contraseña del usuario. Cómo se hace, depende de si hay algún método asociado con el evento *OnLogin* del objeto o no. Si el evento no está asignado, Delphi muestra un cuadro de diálogo predefinido, a través del cual se puede indicar el nombre del usuario y su contraseña. El nombre del usuario se inicializa con el valor extraído del parámetro *USER NAME* del alias:

```
Database1.Params.Values['USER NAME']
```



Si, por el contrario, interceptamos el evento, es responsabilidad del método receptor asignar un valor al parámetro *PASSWORD* de la base de datos. Es bastante común interceptar este evento para mostrar un diálogo de conexión personalizado. Otro motivo para interceptar este evento puede ser la necesidad de quedarnos con el nombre del usuario, quizás para validar más adelante ciertas operaciones, o para llevar un registro de conexiones, si el sistema no lo hace automáticamente. Incluso, si queremos hacer trampas, es posible programar una especie de "caballo de Troya" para espiar las contraseñas, ya que también estarán a nuestra disposición; todo depende de la ética del programador.

La declaración del evento *OnLogin* es la siguiente:

```
type
   TLoginEvent = procedure(Database: TDatabase;
      LoginParams: TStrings) of object;
```

Una posible respuesta a este evento puede ser:

```
procedure TmodDatos.Database1Login(Database: TDatabase;
   LoginParams: TStrings);
begin
   if FNombreUsuario = '' then
      if dlgPassword.ShowModal = mrOk then
      begin
         FNombreUsuario := dlgPassword.Edit1.Text;
         FPassword := dlgPassword.Edit2.Text;
      end;
   LoginParams.Values['USER NAME'] := FNombreUsuario;
   LoginParams.Values['PASSWORD'] := FPassword;
end;
```

Este método asume que hay una sola base de datos en la aplicación. Estamos almacenando el nombre de usuario y su contraseña en los campos *FNombreUsuario* y *FPassword*, declarados como privados en la definición del módulo. Si es la primera vez que nos conectamos a la base de datos, aparece el diálogo mediante el cual pedimos los datos de conexión; aquí hemos nombrado *dlgPassword* a este formulario. A partir de la primera conexión, los datos del usuario no vuelven a necesitarse. Esto puede ser útil cuando intentamos mantener cerradas las conexiones inactivas (*KeepConnections* igual a *False*), pues es bastante engorroso que el usuario tenga que estar tecleando una y otra vez la contraseña cada vez que se reabre la base de datos.

Hay que tener cuidado, sin embargo, con la técnica anterior, pues no ofrece la posibilidad de verificar si los datos suministrados son correctos. De no serlo, cualquier intento posterior de conexión falla, pues ya están almacenados en memoria un nombre de usuario y contraseña no válidos.

El directorio temporal de Windows

El lector sabe que el parámetro *ENABLE SCHEMA CACHE* de los controladores SQL permite acelerar la conexión de una aplicación a un servidor, porque indica al BDE que almacene la información de esquema de las tablas en el cliente. Sabe también que el parámetro *SCHEMA CACHE DIR* sirve para indicar en qué directorio situar esta información de esquema. Sin embargo, es muy poco probable que alguien suministre un directorio estático para este parámetro, pues el directorio debe existir en el momento en que se realiza la conexión a la base de datos. Por ejemplo, ¿qué pasaría si quisiéramos que esta información se almacenara siempre en el directorio temporal de Windows? Pues que tendríamos problemas si la aplicación puede ejecutarse indistintamente en Windows NT o en Windows 95, ya que ambos sistemas operativos definen diferentes ubicaciones para sus directorios temporales.

La solución consiste en utilizar también el evento *OnLogin* para cambiar el directorio de la caché de esquemas *antes* de que se abra la base de datos:

```
procedure TmodDatos.Database1Login(Database: TDatabase;
   LoginParams: TStrings);
var
   S: string;
begin
   SetLength(S, 255);
   SetLength(S, GetTempPath(255, PChar(S)));
   if S[Length(S)] = '\' then
      Delete(S, Length(S), 1);
   Database1.Params.Values['ENABLE SCHEMA CACHE'] := 'TRUE';
   Database1.Params.Values['SCHEMA CACHE DIR'] := S;
   // …
end;
```

El truco ha consistido en modificar el valor del parámetro directamente en la base de datos, no en el objeto *LoginParams* del evento.

También vale redefinir el método *Loaded* del módulo de datos, de forma similar a como hicimos para preparar las consultas explícitamente en el capítulo 26.

Problemas con la herencia visual

Si un módulo de datos contiene un componente *TDatabase* no es posible, en Delphi 2, derivar módulos del mismo por medio de la herencia. La explicación es sencilla: un objeto *TDatabase* define, con su sola presencia, un alias local para la aplicación en la que se encuentra. Por lo tanto, si hubieran dos objetos de esta clase con iguales propiedades en dos módulos diferentes de la misma aplicación, y esto es lo que sucede en la herencia visual, se intentaría definir el mismo alias local dos veces.

Existen dos soluciones para este problema, si estamos interesados en utilizar herencia visual a partir de un módulo de datos y no podemos actualizarnos a Delphi 3 ó 4. La primera consiste en no utilizar objetos *TDatabase* persistentes en el módulo; si lo único que queremos es acceso fácil al objeto *TDatabase* asociado a las tablas, para controlar transacciones, por ejemplo, podemos utilizar con la misma facilidad la propiedad *Database* de los conjuntos de datos. La otra solución, en el caso de que la anterior no valga, es situar el objeto de bases de datos en un módulo aparte de las tablas. En este caso, el módulo que contiene las tablas puede servir como clase base para la herencia, mientras que el módulo de la base de datos puede utilizarse directamente o ser copiado en nuestro proyecto.

Aislar la base de datos en un módulo separado puede ser conveniente por otros motivos. En el epígrafe anterior vimos una forma de evitar que el usuario teclee innecesariamente sus datos cada vez que se inicia una conexión. Habíamos mencionado también el problema de validar la conexión, para reintentarla en caso de fallo. Si tenemos la base de datos aislada en un módulo, que se debe crear antes de los módulos que contienen las tablas, podemos controlar la conexión del siguiente modo, durante la creación del módulo:

```
procedure TmodDatosDB.modDatosDBCreate(Sender: TObject);
var
   Intentos: Integer;
begin
   // Tres reintentos como máximo
   for Intentos := 1 to 3 do
      try
         Database1.Open;
         // Si todo va bien, nos vamos
         Exit;
      except
         // Reiniciar los datos de usuario, y volver a probar
         FNombreUsuario := '';
         FPassword := '';
      end;
   // Si no se puede, terminar la aplicación
   Application.Terminate;
end;
```

La condición necesaria para que el código anterior funcione es que la base de datos esté cerrada en tiempo de diseño.

El problema expuesto anteriormente se resuelve a partir de Delphi 3 con una nueva propiedad: *HandleShared*. Si esta propiedad es *True*, dos bases de datos pueden tener el mismo nombre y compartir el mismo *handle* del BDE. Si colocamos un *TDatabase* en un módulo, basta con activar esta propiedad para poder derivar módulos por herencia sin ningún tipo de problemas.

Sesiones

Si seguimos ascendiendo en la jerarquía de organización de objetos del BDE, pasaremos de las bases de datos a las sesiones. El uso de sesiones en Delphi nos permite lograr los siguientes objetivos:

- Cada sesión define un usuario diferente que accede al BDE. Si dentro de una aplicación queremos sincronizar acciones entre procesos, sean realmente concurrentes o no, necesitamos sesiones.
- Las sesiones nos permiten administrar desde un mismo sitio las conexiones a bases de datos de la aplicación. Esto incluye la posibilidad de indicar valores por omisión a las propiedades de las bases de datos.
- 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.
- Mediante las sesiones, podemos controlar el proceso de conexión a tablas Paradox protegidas por contraseñas.

En la primera versión de Delphi sólo era necesaria una sesión por aplicación, porque podía ejecutarse un solo hilo por aplicación. En consecuencia, en esta versión la clase *TSession* no estaba disponible como componente en la Paleta, y para tener acceso a la única sesión del programa teníamos la variable global *Session*. A partir de Delphi 2, además de poder crear sesiones en tiempo de diseño, seguimos teniendo la variable *Session*, que esta vez se refiere a la sesión por omisión. En cambio, se añade la nueva variable global *Sessions*, del tipo *TSessionList*, que permite el acceso a todas las sesiones existentes en la aplicación.

Tanto *Sessions*, como *Session*, están declaradas, en Delphi 2, en la unidad *DB* (en *DBTables*, a partir de Delphi 3) y son creadas automáticamente por el código de inicialización de esta unidad en el caso de que ésta sea mencionada por alguna de las unidades de su proyecto.

Cada sesión es un usuario

Hemos explicado que cada sesión define un acceso diferente al BDE, como si fuera un usuario distinto. La consecuencia más importante de esto es que el BDE levanta barreras de contención entre estos diferentes usuarios. Por ejemplo, si en una sesión se abre una tabla en modo exclusivo, dentro de la misma sesión se puede volver a abrir la tabla en este modo, pues la sesión no se bloquea a sí misma. Lo mismo ocurre con cualquier posible bloqueo a nivel de registro. Es fácil realizar una demostración práctica de esta peculiaridad. Sobre un formulario vacío coloque dos objetos de tipo *TSession* y configúrelos del siguiente modo:

Propiedad	Primera sesión	Segunda sesión
Name	*Session1*	*Session2*
SessionName	*S1*	*S2*

Traiga también un par de tablas, con las siguientes propiedades:

Propiedad	Primera tabla	Segunda tabla
Name	*Table1*	*Table2*
DatabaseName	*dbdemos*	*dbdemos*
TableName	*biolife.db*	*biolife.db*
Exclusive	*True*	*True*

Por último, sitúe dos botones sobre el formulario, con las etiquetas *"Una sesión"* y *"Dos sesiones"*. El primer botón intentará abrir las dos tablas en exclusiva durante la misma sesión; el segundo hará lo mismo, asignando primero diferentes sesiones. La respuesta a estos botones será compartida:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   Table1.Close;
   Table2.Close;
   Table1.SessionName := 'S1';
   if Sender = Button1 then
      Table2.SessionName := 'S1'
   else
      Table2.SessionName := 'S2';
   Table1.Open;
   Table2.Open;
end;
```

El resultado, por supuesto, será que el primer botón podrá ejecutar su código sin problemas, mientras que el segundo botón fallará en su intento.

> Ha sido fundamental que la tabla del ejemplo perteneciera a una base de datos de escritorio. La mayoría de los sistemas SQL ignoran la propiedad *Exclusive* de las tablas.

Sesiones e hilos paralelos

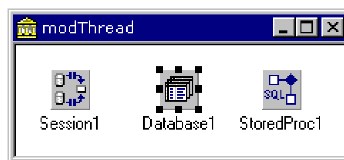
La principal aplicación de estas propiedades de la sesión es poder realizar operaciones de bases de datos en distintos hilos de la misma aplicación; cada hilo enchufa sus componentes de bases de datos por medio de una sesión diferente. Los servidores de automatización y las extensiones ISAPI/NSAPI para servidores de Internet, que estudiaremos en capítulos posteriores, permiten que varios clientes se conecten a la

misma instancia de la aplicación. A cada cliente se le asigna un hilo diferente, por lo que es esencial utilizar sesiones para evitar conflictos entre las peticiones y modificaciones de datos. Todo esto lo veremos en su momento.

Ahora bien, ¿es apropiado utilizar hilos en aplicaciones clientes "normales"? El ejemplo más socorrido en los libros de Delphi es el de una aplicación MDI que abre una ventana hija basada en el resultado de la ejecución de una consulta. Como la ejecución de la consulta por el servidor (o por el intérprete local) puede tardar, para que el usuario no pierda el control de la interfaz de la aplicación, la apertura de la consulta se efectúa en un hilo separado. La técnica es correcta, y los motivos impecables. Pero yo nunca haría tal disparate en una aplicación real, pues cada ventana lanzada de esta manera consumiría una conexión a la base de datos, que es un recurso generalmente limitado y costoso. Aún después de haber terminado la ejecución del hilo que abre la consulta, el objeto *TQuery* sigue conectado a una sesión separada, como si hubiera un Dr. Jekyll y un Mr. Hide dentro de mi ordenador personal.

Se me ocurre, sin embargo, un ejemplo ligeramente diferente en el que sí es recomendable utilizar un hilo en paralelo. Sustituya en el párrafo anterior la palabra "consulta" por "procedimiento almacenado". Supongamos que nuestra base de datos cliente/servidor tiene definido un procedimiento que realiza una operación de mantenimiento larga y costosa, y que ese procedimiento debemos lanzarlo desde la aplicación. En principio, el procedimiento no devuelve nada importante. Si utilizamos la técnica de lanzamiento convencional, tenemos que esperar a que el servidor termine para poder continuar con la aplicación. Y esta espera es la que podemos evitar utilizando hilos y sesiones. ¿Qué diferencia hay con respecto al ejemplo de la consulta? Una fundamental: que cuando termina la ejecución del procedimiento podemos desconectar la sesión adicional, mientras que no podemos hacer lo mismo con la consulta hasta que no cerremos la ventana asociada.

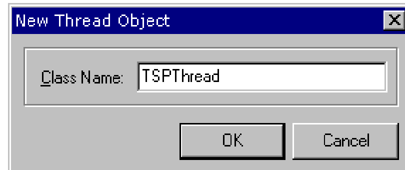
Iniciamos una aplicación. Ponga en la ventana principal una rejilla conectada a una tabla perteneciente a una base de datos cliente/servidor. Ahora cree un módulo de datos, llámelo *modThread* y coloque en él los siguientes tres objetos:



En el objeto *Session1* solamente es necesario asignar un nombre en *SessionName*, digamos que sea *S1*. Este mismo nombre debe copiarse en la propiedad homónima del *TDatabase*. Configure, además, este componente para que podamos conectarnos a la misma base de datos que estamos explorando en la ventana principal. Finalmente, cambie también *SessionName* en el procedimiento almacenado, y engánchelo a algún

procedimiento almacenado de la base de datos cuya ejecución requiera bastante tiempo. Y muy importante: ¡deje inactivos a todos los objetos! No queremos que esta sesión esté abierta desde que arranque la aplicación.

Vamos ahora a programar el hilo que se encargará de ejecutar el procedimiento. Ejecute *File|New* para obtener el diálogo del Depósito de Objetos, y realice una doble pulsación en el icono *Thread object*:



Este experto crea, en una unidad aparte, un esqueleto de clase que debemos modificar del siguiente modo:

```
type
   TSPThread = class(TThread)
   protected
      procedure Execute; override;
   public
      constructor Create;
   end;
```

Hemos añadido un constructor al objeto. He aquí el cuerpo de los métodos:

```
constructor TSPThread.Create;
begin
   inherited Create(True);    // Crear un hilo "suspendido"
   FreeOnTerminate := True;
   Resume;                    // Continuar la ejecución
end;

procedure TSPThread.Execute;
begin
   modThread.Database1.Open;
   try
      modThread.StoredProc1.ExecProc;
   finally
      modThread.Session1.Close;
   end;
end;
```

El constructor crea inicialmente el hilo en estado "suspendido", es decir, no comienza inmediatamente su ejecución. Antes de lanzarlo, asigna *True* a la propiedad *FreeOnTerminate*, para que la memoria del objeto *TSPThread* sea liberada al finalizar la ejecución del hilo. Lo que haga el hilo estará determinado por el contenido del método *Execute*. En éste se accede a los objetos necesarios (sesión, base de datos, procedimiento almacenado) mediante la variable global *modThread*; recuerde que los hilos

comparten el mismo espacio de memoria dentro de una aplicación. He abierto explícitamente la base de datos antes de ejecutar el procedimiento, y después me he asegurado de que se cierra la sesión (y con ella la base de datos). Quizás usted tenga que retocar un poco el código para que no se vuelva a pedir el nombre de usuario y su contraseña al abrirse la base de datos.

Con este objeto a nuestra disposición, lo único que tiene que hacer la ventana principal para ejecutar el procedimiento almacenado en segundo plano es lo siguiente:

```
procedure TForm1.Consolidacion1Click(Sender: TObject);
begin
   TSPThread.Create;
end;
```

Al crearse el objeto del hilo, automáticamente se inicia su ejecución. Recuerde que la última instrucción del constructor es *Resume*. La destrucción del objeto creado es automática, y ocurre cuando el procedimiento almacenado ha finalizado su acción, y se ha roto la segunda conexión a la base de datos.

Información sobre esquemas

La clase *TSession* tiene métodos para extraer la información sobre esquemas de las bases de datos registradas por el BDE. Para comenzar por la cima de la jerarquía, tenemos *GetDriverNames*, para recuperar los nombres de controladores instalados:

```
procedure TSession.GetDriverNames(Lista: TStrings);
```

Este método, y la mayoría de los que siguen a continuación que devuelven una lista de nombres, vacían primero la lista de cadenas antes de asignar valores. En este caso, en *Lista* queda la lista de controladores registrados en el BDE; el controlador *STANDARD* se refiere a Paradox y dBase. Una vez que tenemos un nombre de controlador podemos averiguar sus parámetros:

```
procedure TSession.GetDriverParams(const Controlador: string;
   Lista: TStrings);
```

Para obtener la lista de bases de datos y alias, se utilizan *GetDatabaseNames* y *GetAliasName*. La diferencia entre ambos métodos es que el primero devuelve, además de los alias persistentes, los alias locales declarados mediante objetos *TDatabase*; el segundo se limita a los alias persistentes. Además, ya hemos utilizado *GetDatabaseNames* en un capítulo anterior. Tenemos además la función *GetAliasDriverName* y *GetAliasParams* para extraer la información asociada a un alias determinado:

```
procedure TSession.GetDatabaseNames(Lista: TStrings);
procedure TSession.GetAliasNames(Lista: TStrings);
function  TSession.GetAliasDriverName(const Alias: string): string;
procedure TSession.GetAliasParams(const Alias: string;
   Lista: TStrings);
```

Una vez que tenemos un alias en la mano, podemos averiguar qué tablas existen en la base de datos asociada. Para esto utilizamos el método *GetTableNames*:

```
procedure TSession.GetTableNames(const Alias, Patron: string;
   Extensiones, TablasDelSistema: Boolean; Lista: TStrings);
```

El parámetro *Patron* permite filtrar las bases de datos; la cadena vacía se utiliza para seleccionar todas las tablas. El parámetro *Extensiones* sirve para incluir o eliminar las extensiones de ficheros en los sistemas de bases de datos locales. Por último, *Tablas-DelSistema* se utiliza en las bases de datos SQL para incluir o descartar las tablas que el sistema de bases de datos crea automáticamente.

Del mismo modo, para una base de datos SQL se puede utilizar el siguiente método que devuelve los procedimientos almacenados definidos:

```
procedure TSession.GetStoredProcNames(const Alias: string;
   Lista: TStrings);
```

El MiniExplorador de Bases de Datos

El ejemplo siguiente muestra una forma sencilla de llenar un árbol con la información de alias y tablas existentes. Necesitamos un formulario con un componente *TTreeView*, de la página *Win32* de la paleta, alineado a la izquierda, un cuadro de lista, *TListBox*, que ocupe el resto del formulario, y un objeto *TImageList*, también de la página *Win32*, para que contenga los iconos que vamos a mostrar al lado de los alias y las tablas. Para inicializar este último objeto, pulsamos dos veces sobre el mismo. En el editor que aparece, utilizamos el botón *Add* para cargar un par de imágenes en el control. Utilizaremos la primera para representar los alias, dejando la segunda para las tablas.

Ahora debemos interceptar el evento *OnCreate* del formulario, para inicializar la lista de alias presentes en el ordenador:

```
procedure TForm1.FormCreate(Sender: TObject);
var
   List: TStrings;
   AliasPtr: TTreeNode;
   I: Integer;
begin
   List := TStringList.Create;
   try
      Session.GetDatabaseNames(List);
```

```
      for I := 0 to List.Count - 1 do
      begin
        AliasPtr := TreeView1.Items.Add(nil, List[I]);
        AliasPtr.ImageIndex := 0;
        TreeView1.Items.AddChild(AliasPtr, '');
      end;
    finally
      List.Free;
    end;
  end;
```

He utilizado el truco de añadir un hijo ficticio al nodo alias, para que aparezca el botón de expansión. Solamente cuando se pulse por primera vez este botón, se buscarán las tablas pertenecientes al alias, con el objetivo de minimizar el tiempo de carga de la aplicación. La expansión del nodo se realiza en respuesta al evento *OnExpanding* del visualizador de árboles:

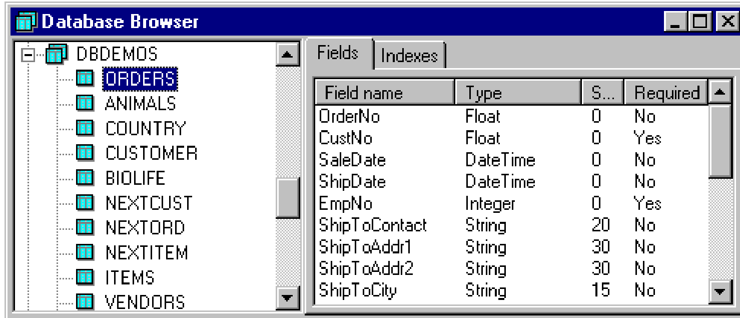
```
procedure TForm1.TreeView1Expanding(Sender: TObject;
  Node: TTreeNode; var AllowExpansion: Boolean);
var
  List: TStrings;
  I: Integer;
begin
  if Node.Data <> nil then Exit;
  Node.DeleteChildren;
  List := TStringList.Create;
  try
    Session.GetTableNames(Node.Text, '', False, False, List);
    for I := 0 to List.Count - 1 do
      with TreeView1.Items.AddChild(Node, List[I]) do
      begin
        ImageIndex := 1;
        SelectedIndex := 1;
      end;
    Node.Data := Node;
  finally
    List.Free;
  end;
end;
```

Estoy utilizando otro truco "sucio" para saber si un nodo ha sido expandido o no. Los nodos de árboles tienen una propiedad *Data*, de tipo *Pointer*, en la cual podemos guardar la información que se nos antoje. Para este ejemplo, si *Data* contiene el puntero vacío estamos indicando que el nodo aún no ha sido expandido; cuando el nodo se expanda, *Data* pasará a apuntar al propio nodo.

Por último, cuando seleccionemos un nodo alias, extraeremos la información del mismo y la visualizaremos en el cuadro de listas de la derecha:

```
procedure TForm1.TreeView1Change(Sender: TObject; Node: TTreeNode);
begin
  if Node.Parent = nil then
  begin
    ListBox1.Items.BeginUpdate;
```

```
        try
          Session.GetAliasParams(Node.Text, ListBox1.Items);
          ListBox1.Items.Insert(0,
            'DRIVER=' + Session.GetAliasDriverName(Node.Text));
        finally
          ListBox1.Items.EndUpdate;
        end;
      end;
    end;
```



Observe el uso del método *BeginUpdate* para evitar el parpadeo provocado por la actualización de la pantalla mientras insertamos cadenas en el control.

Gestión de alias a través de TSession

También se pueden crear alias y modificar parámetros de alias existentes mediante los objetos de sesión. Para añadir nuevos alias, utilizamos los siguientes métodos:

```
procedure TSession.AddAlias(const Nombre, Ctrldor: string;
  Lista: TStrings);
procedure TSession.AddStandardAlias(const Nombre, Dir,
  Ctrldor: string);
```

AddAlias es el método más general para añadir un nuevo alias, mientras que *AddStandardAlias* simplifica las cosas cuando queremos crear un alias para bases de datos locales. El tipo de alias que se cree, persistente o local, depende de la propiedad *ConfigMode* de la sesión; esta variable puede asumir los valores *cmPersistent*, *cmSession* y *cmAll*. El último valor es el valor por omisión. Si queremos que el alias creado sea local, debemos utilizar el valor *cmSession*.

Para modificar o eliminar un alias existente, se pueden utilizar los siguientes métodos:

```
procedure TSession.ModifyAlias(const Alias: string;
  Parametros: TStrings);
procedure TSession.DeleteAlias(const Alias: string);
```

Por último, para guardar la configuración y que los cambios sean permanentes hay que utilizar el siguiente método:

```
procedure TSession.SaveConfigFile;
```

En nuestro mini-explorador de bases de datos añadimos un par de comandos de menú: *Crear alias* y *Eliminar alias*. Para eliminar un alias utilizamos el siguiente procedimiento:

```
procedure TForm1.miDeleteAliasClick(Sender: TObject);
begin
   with TreeView1.Selected do
      if Parent = nil then
         if MessageDlg('Eliminando el alias ' + Text +
            '.'#13'¿Está seguro?',
            mtConfirmation, [mbYes, mbNo], 0) = mrYes then
         begin
            Session.DeleteAlias(Text);
            Free;
         end;
end;
```

Para crear un nuevo alias, necesitaremos una segunda ventana en la aplicación, a la cual nombraremos *dlgNewAlias*. Esta ventana, configurada como diálogo, contiene los siguientes objetos:

Componente	Función
Edit1	Contiene el nombre del nuevo alias
ComboBox1	De estilo *csDropDownList*, contiene las cadenas *PARADOX* y *DBASE*. Es el nombre del controlador por omisión.
DirectoryListBox1	Para seleccionar un directorio.

Necesitamos también botones para aceptar y cancelar la ejecución del diálogo. La respuesta al comando de creación de alias es la siguiente:

```
procedure TForm1.miNewAliasClick(Sender: TObject);
var
   AliasPtr: TTreeNode;
begin
   with dlgNewAlias do
      if ShowModal = mrOk then
      begin
         Session.AddStandardAlias(Edit1.Text,
            DirectoryListBox1.Directory, ComboBox1.Text);
         AliasPtr := TreeView1.Items.Add(nil, Edit1.Text);
         AliasPtr.ImageIndex := 0;
         TreeView1.Items.AddChild(AliasPtr, '');
      end;
end;
```

Para simplificar la explicación, hemos creado un alias para el controlador estándar. La creación de alias para otros controladores no plantea mayores dificultades.

En Delphi 1, la creación de alias persistentes se realizaba mediante llamadas directas al BDE. Curiosamente, no existía ninguna llamada en la interfaz del BDE para eliminar un alias una vez creado.

Directorios privados, de red y contraseñas

Como si las sesiones no hicieran bastante ya, también se ocupan de la configuración de los directorios *NetDir* y *PrivateDir* del Borland Database Engine, y de la gestión de contraseñas de usuarios de Paradox. Ya hemos visto, al configurar el BDE, la función de estos directorios. Sepa ahora que puede cambiarlos desde un programa Delphi utilizando objetos *TSession*, si no se han efectuado aún conexiones a bases de datos. Para cambiar estas propiedades es útil el evento *OnStartup*, que se dispara justamente antes de iniciar la sesión.

En cuanto a las contraseñas de las tablas Paradox, el mecanismo de gestión de las mismas es diferente al de las bases de datos SQL típicas. Aquí, las contraseñas se definen a nivel de tabla, no de la base de datos. Por lo tanto, la contraseña se pide al intentar abrir una tabla protegida mediante este recurso. Este es el comportamiento de Delphi por omisión, y no hay que programar algo especial para trabajar con este tipo de tablas. Sin embargo, por causa de que las contraseñas son locales a tablas, si tenemos un par de tablas protegidas por la misma contraseña, tendremos que teclearla dos veces para abrir las dos tablas. Esto puede ser bastante engorroso, por lo cual el BDE permite almacenar a nivel de sesión un conjunto de contraseñas permitidas. El cuadro de apertura para tablas con contraseña permite almacenar en la sesión actual la contraseña suministrada. De este modo, si se guarda la contraseña durante la apertura de la primera tabla, ésta no será solicitada al abrir la siguiente tabla.

Los siguientes métodos de las sesiones trabajan sobre la lista de contraseñas disponibles en la sesión:

```
procedure TSession.AddPassword(const Password: string);
procedure TSession.RemovePassword(const Password: string);
procedure TSession.RemoveAllPasswords;
```

Si queremos saltarnos el diálogo de petición de contraseñas de Delphi, podemos realizar una llamada a *AddPassword* en cualquier momento previo a la apertura de la tabla protegida. Por lo general, el mejor momento para esto es durante el evento *OnPassword* del componente *TSession*. Este evento se dispara cuando se produce un error al abrir una tabla por no disponer de los suficientes derechos. *OnPassword* pertenece al tipo de eventos que controlan un bucle de reintentos; para esto cuenta con un parámetro lógico *Continue*, con el cual podemos controlar el fin del bucle:

```
procedure TForm1.Session1Password(Sender: TObject;
   var Continue: Boolean);
var
   S: string;
begin
   S := '';
   if InputQuery('Dis-moi, miroir magique…',
      '¿Cuál es el mejor lenguaje de la Galaxia?', S)
      and (S = 'Delphi') then
   begin
      Session1.AddPassword('BuenChico');
      Continue := True;
   end;
end;
```

Capítulo

31

Transacciones y control de concurrencia

HASTA EL MOMENTO HEMOS ASUMIDO que solamente nuestra aplicación tiene acceso, desde un solo puesto, a los datos con los que trabaja. Aún sin trabajar con bases de datos SQL en entornos cliente/servidor, esta suposición es irreal, pues casi cualquier escenario de trabajo actual cuenta con varios ordenadores conectados en una red puesto a puesto; a una aplicación para los formatos de datos más sencillos, como Paradox y dBase, se le exigirá que permita el acceso concurrente a éstos.

El programador típico de bases de datos locales asocia la solución de los problemas de concurrencia con la palabra mágica "bloqueos"; es posible que para localizar este capítulo hubiera buscado esa palabra en su título. 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.

Este capítulo comienza explicando la forma en que implícitamente Delphi resuelve los problemas más sencillos del acceso concurrente. Después pasa a explorar el concepto de transacción, como operación atómica e independiente. Y finalmente aprenderemos como esta técnica puede ayudarnos en la introducción de datos correspondientes a objetos complejos.

El Gran Experimento

Estamos en la Grecia clásica, y un grupo de atenienses discuten acerca de cuántos dientes debe tener un camello. Unos dicen que, teniendo en cuenta su parentesco con los caballos, los camellos tienen el mismo número de piezas dentales que los equinos. Otros afirman que un camello que se precie debe tener treinta y dos piezas, porque la armonía matemática exige una potencia de dos en la boca del rumiante

(como podéis observar, la fábula ha sido adaptada para nuestros tiempos modernos). En esto, tercia un árabe en la discusión diciendo que un camello de buena cuna posee exactamente veinticuatro molares, ni más ni menos. Los académicos, mirando despectivamente al hijo del desierto, le preguntan en qué teoría se basa para tan atrevida afirmación. "Muy sencillo" — dice el buen señor — "soy camellero y los he contado".

En esta misma situación me encuentro con frecuencia cuando me preguntan cómo funcionan los bloqueos en Delphi. Salvando las distancias, hay un experimento muy sencillo que nos enseña bastante: ejecute una aplicación que utilice una tabla dos veces en la misma máquina. Recuerde, por favor, que ya no estamos en MS-DOS. Este experimento puede realizarse también desde dos máquinas diferentes conectadas en una red, con el alias de la aplicación configurado de modo tal que ambos apunten a la misma base de datos o directorio. Pero los resultados son los mismos y el experimento resulta más sencillo del modo que proponemos.

La aplicación en sí será muy sencilla: una tabla, una fuente de datos (*TDataSource*), una rejilla de datos (*TDbGrid*) y una barra de navegación (*TDbNavigator*), esta última para facilitarnos las operaciones sobre la tabla.

El Gran Experimento: tablas locales

En su primera versión, la tabla debe referirse a una base de datos local, a una tabla en formato Paradox ó dBase. Para lograr que la misma aplicación se ejecute dos veces sin necesidad de utilizar el Explorador de Windows o el menú Inicio, cree el siguiente método en respuesta al evento *OnCreate* del formulario principal:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
   Top := 0;
   Width := Screen.Width div 2;
   Height := Screen.Height;
   if (CreateSemaphore(nil, 0, 1,
      PChar(ExtractFileName(Application.ExeName))) <> 0) and
      (GetLastError = ERROR_ALREADY_EXISTS) then
      Left := Screen.Width div 2
   else
   begin
      Left := 0;
      WinExec(PChar(Application.ExeName), SW_NORMAL);
   end;
end;
```

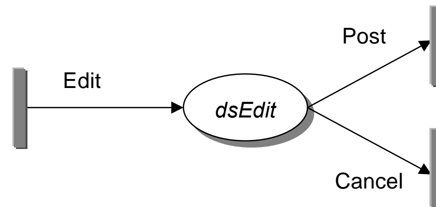
Para detectar la presencia de la aplicación he utilizado un semáforo binario, que creamos con la función *CreateSemafore*. Para que se reconozca globalmente el semáforo hay que asignarle un nombre, que estamos formando a partir del nombre de la aplicación.

Ejecute dos instancias de la aplicación y efectúe entonces la siguiente secuencia de operaciones:

- Sitúese sobre un registro cualquiera de la rejilla, en la primera aplicación. Digamos, por conveniencia, que éste sea el primer registro.
- Ponga la tabla en estado de edición, pulsando el botón del triángulo (*Edit*) de la barra de navegación. El mismo efecto puede obtenerlo tecleando cualquier cosa en la rejilla, pues la fuente de datos tiene la propiedad *AutoEdit* con el valor por omisión, *True*. En cualquiera de estos dos casos, tenga cuidado de no cambiar la fila activa, pues se llamaría automáticamente a *Post*, volviendo la tabla al modo *dsBrowse*.
- Deje las cosas tal como están en la primera aplicación, y pase a la segunda.
- Sitúese en el mismo registro que escogió para la primera aplicación e intente poner la tabla en modo de edición, pulsando el correspondiente botón de la barra de navegación, o tecleando algo.

He escrito "intentar", porque el resultado de esta acción es un mensaje de error: "Registro bloqueado por otro usuario". Si tratamos de modificar una fila diferente no encontraremos problemas, lo que quiere decir que el bloqueo se aplica solamente al registro que se está editando en la otra aplicación, no a la tabla completa. También puede comprobar que el registro vuelve a estar disponible en cuanto guardamos las modificaciones realizadas en la primera aplicación, utilizando el botón de la marca de verificación (√) o moviéndonos a otra fila. Lo mismo sucede si se cancela la operación.

La conclusión a extraer de este experimento es que, para las tablas locales, el método *Edit*, que se llama automáticamente al comenzar alguna modificación sobre la tabla, intenta colocar un bloqueo sobre la fila activa de la tabla. Este bloqueo puede eliminarse de dos formas: con el método *Post*, al confirmar los cambios, y con el método *Cancel*, al descartarlos.



El Gran Experimento: tablas SQL

Repetiremos ahora el experimento, pero cambiando el formato de la tabla sobre la cual trabajamos. Esta vez conecte con una tabla de InterBase, da lo mismo una que otra. Ejecute nuevamente la aplicación dos veces y siga estos pasos:

- En la primera aplicación, modifique el primer registro de la tabla, pero no confirme la grabación, dejando la tabla en modo de inserción.
- En la segunda aplicación, ponga la tabla en modo de edición y modifique el primer registro de la tabla. Esta vez no debe ocurrir error alguno. Grabe los cambios en el disco.
- Regrese a la primera aplicación e intente grabar los cambios efectuados en esta ventana.

En este momento, *sí* que tenemos un problema. La excepción se presenta con el siguiente mensaje: "No se pudo realizar la edición, porque otro usuario cambió el registro". Hay que cancelar los cambios realizados, y entonces se releen los datos de la tabla, pues los valores introducidos por la segunda aplicación aparecen en la primera.

Pesimistas y optimistas

La explicación es, en este caso, más complicada. Acabamos de ver en acción un mecanismo *optimista* de control de edición. En contraste, a la técnica utilizada con las bases de datos locales se le denomina *pesimista*. Ya hemos visto que un sistema de control pesimista asume, al intentar editar un registro, que cualquier otro usuario puede estar editando este registro, por lo que para iniciar la edición "pide permiso" para hacerlo. Pedir permiso es el símil de colocar un bloqueo (*lock*): si hay realmente otro usuario editando esta fila se nos denegará dicho permiso.

Delphi transforma la negación del bloqueo en una excepción. Antes de lanzar la excepción, se nos avisa mediante el evento *OnEditError* de la tabla. En la respuesta a este evento tenemos la posibilidad de reintentar la operación, fallar con la excepción o fallar silenciosamente, con una excepción *EAbort*. En la lejana época en la que los Flintstones hacían de las suyas y la mayor parte de las aplicaciones funcionaban en modo *batch*, era de suma importancia decidir cuándo la aplicación que no obtenía un bloqueo se cansaba de pedirlo. Ahora, sencillamente, se le puede dejar la decisión al usuario. He aquí una simple respuesta al evento *OnEditError*, que puede ser compartido por todas las tablas locales de una aplicación:

```
procedure TForm1.Table1EditError(DataSet: TDataSet;
   E: EDatabaseError; var Action: TDataAction);
```

```
begin
   if MessageDlg(E.Message, mtWarning, [mbRetry, mbAbort], 0)
      = mrAbort then
      Action := daAbort
   else
      Action := daRetry
end;
```

Otra posibilidad es programar un bucle infinito de reintentos. En este caso, es recomendable reintentar la operación transcurrido un intervalo de tiempo prudencial; cuando llamamos por teléfono y está la línea ocupada no marcamos frenéticamente el mismo número una y otra vez, sino que esperamos a que la otra persona termine su llamada. En este código muestro además cómo esperar un intervalo de tiempo aleatorio:

```
procedure TForm1.Table1EditError(DataSet: TDataSet;
   E: EDatabaseError; var Action: TDataAction);
var
   I, Tiempo: Integer;
begin
   I := GetTickCount;
   Tiempo := 1000 + Random(1000);
   // Esperar entre 1 y 2 segundos
   while GetTickCount - I < Tiempo do
      Application.ProcessMessages;
   Action := daRetry;
end;
```

El evento *OnEditError* no existe en la versión 1 de Delphi. La única forma de controlar este tipo de errores era encerrar la llamada al método *Edit* dentro de una instrucción **try**…**except**. Por supuesto, las llamadas implícitas, como las provocadas por teclear sobre un control *data-aware*, quedaban fuera de esta protección.

El sistema pesimista es el más adecuado para bases de datos locales, pero para bases de datos SQL en entornos cliente/servidor, donde toda la comunicación transcurre a través de la red, y en la que se trata de maximizar la cantidad de usuarios que pueden acceder a las bases de datos, no es la mejor política. En este tipo de sistemas, el método *Edit*, que marca el comienzo de la operación de modificación, no intenta colocar el bloqueo sobre la fila que va a cambiar. Es por eso que no se produce una excepción al poner la misma fila en modo de edición por dos aplicaciones simultáneas.

Las dos aplicaciones pueden realizar las asignaciones al *buffer* de registro sin ningún tipo de problemas. Recuerde que este *buffer* reside en el ordenador cliente. La primera de ellas que termine, puede enviar sin mayores dificultades la petición de actualización al servidor. Sin embargo, cuando la segunda intenta hacer lo mismo, descubre que el registro que había leído originalmente ha desaparecido, y en ese momento se aborta la operación mediante una excepción. Esta excepción pasa primero por el evento *OnPostError*, aunque en este caso lo mejor es releer el registro, si no se ha modificado la clave primaria, y volver a repetir la operación.

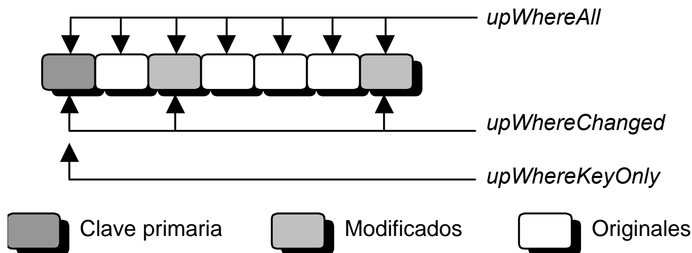
La suposición básica tras esta aparentemente absurda filosofía es que es poco probable que dos aplicaciones realmente traten de modificar el mismo registro a la vez. Piense, por ejemplo, en un cajero automático. ¿Qué posibilidad existe de que tratemos de sacar dinero al mismo tiempo que un buen samaritano aumenta nuestras existencias económicas? Siendo de breve duración este tipo de operaciones, ¿cuán probable es que la compañía de teléfonos y la de electricidad se topen de narices al saquearnos a principios de mes? Sin embargo, la razón de más peso para adoptar una filosofía optimista con respecto al control de concurrencia es que de esta manera disminuye el tiempo en que el bloqueo está activo sobre un registro, disminuyendo por consiguiente las restricciones de acceso sobre el mismo. Ahora este tiempo depende del rendimiento del sistema, no de la velocidad con que el usuario teclea sus datos después de la activación del modo de edición.

El modo de actualización

Cuando una tabla está utilizando el modo optimista de control de concurrencia, este mecanismo se configura de acuerdo a la propiedad *UpdateMode* de la tabla en cuestión. Esta propiedad nos dice qué algoritmo utilizará Delphi para localizar el registro original correspondiente al registro modificado. Los posibles valores son los siguientes:

Valor	Significado
upWhereAll	Todas las columnas se utilizan para buscar el registro a modificar.
upWhereChanged	Solamente se utilizan las columnas que pertenecen a la clave primaria, más las columnas modificadas.
upWhereKeyOnly	Solamente se utilizan las columnas de la clave primaria.

El valor por omisión es *upWhereAll*. Este valor es nuestro seguro de vida, pues es el más restrictivo de los tres. Es, en cambio, el menos eficiente, porque la petición de búsqueda del registro debe incluir más columnas y valores de columnas.



Aunque *upWhereKeyOnly* parezca una alternativa más atractiva, el emplear solamente las columnas de la clave puede llevarnos en el caso más general a situaciones en que dos aplicaciones entran en conflicto durante la modificación de un registro. Piense, por ejemplo, que estamos modificando el salario de un empleado; la clave primaria de la tabla de empleados es su código de empleado. Por lo tanto, si alguien está modificando en otro lugar alguna otra columna, como la fecha de contratación, las actualizaciones realizadas por nuestra aplicación pueden sobrescribir las actualizaciones realizadas por la otra aplicación. Si nuestra aplicación es la primera que escribe, tendremos un empleado con una fecha de contratación correcta y el salario sin corregir; si somos los últimos, el salario será el correcto (¡suerte que tiene el chico!), pero la fecha de contratación no estará al día.

Sin embargo, el valor *upWhereChanged* puede aplicarse cuando queremos permitir actualizaciones simultáneas en diferentes filas de un mismo registro, que no modifiquen la clave primaria; esta situación se conoce como *actualizaciones ortogonales*, y volveremos a mencionarlas en el capítulo sobre bases de datos remotas. Supongamos que nuestra aplicación aumenta el salario al empleado Ian Marteens. Si el valor de *UpdateMode* es *upWhereAll*, la instrucción SQL que lanza el BDE es la siguiente:

```
update  Employee
set     Salary = 1000000       -- Me lo merezco
where   EmpNo = 666
  and  FirstName = 'Ian'
  and  LastName = 'Marteens'
  and  Salary = 0              -- Este es el salario anterior
  and  PhoneExt = '666'
  and  HireDate = '1/1/97'
```

Por supuesto, si alguien cambia cualquier dato del empleado 666, digamos que la extensión telefónica, la instrucción anterior no encontrará registro alguno y se producirá un fallo de bloqueo optimista. En cambio, con *upWhereChanged* la instrucción generada sería:

```
update  Employee
set     Salary = 1000000       -- Me lo merezco
where   EmpNo = 666
  and  Salary = 0              -- Este es el salario anterior
```

Lo único que se exige ahora es que no hayan cambiado a nuestras espaldas el código (nuestra identidad) o el salario (¡a ver si se pelean ahora por aumentarnos la paga!). Este tipo de configuración aumenta, por lo tanto, las posibilidades de acceso concurrente. Debe, sin embargo, utilizarse con cuidado, en particular cuando existen relaciones de dependencia entre las columnas de una tabla. El siguiente ejemplo es un caso extremo: se almacena en una misma fila la edad y la fecha de nacimiento; está claro que no tiene sentido permitir la modificación concurrente de ambas.

Pongamos ahora un ejemplo más real: una relación entre salario y antigüedad laboral, de manera que dentro de determinado rango de antigüedad solamente sean posibles los salarios dentro de cierto rango. Desde un puesto, alguien aumenta la antigüedad, pero desde otro puesto alguien disminuye el salario. Ambas operaciones darían resultados correctos por separado, pero al sumarse sus efectos, el resultado es inadmisible. Sin embargo, la solución es sencilla: si existe la regla, debe haber algún mecanismo que se encargue de verificarla, ya sea una restricción **check** o un *trigger*. En tal caso, la primera de las dos modificaciones triunfaría sin problemas, mientras que la segunda sería rechazada por el servidor.

La relectura del registro actual

Hay otro detalle que no he mencionado con respecto a los fallos de bloqueos optimistas, y que podemos comprobar mediante SQL Monitor. Cuando se produce uno de estos fallos, el BDE relee sigilosamente el registro actual, a nuestras espaldas. Volvamos al ejemplo de la actualización de dos filas. Dos usuarios centran su atención simultáneamente en el empleado Marteens. El primero cambia su fecha de contrato al 1 de Enero del 96, mientras que el segundo aumenta su salario a un millón de dólares anuales. El segundo usuario es más rápido, y lanza primero el *Post*. Para complicar las cosas, supongamos que la tabla de empleados está configurada con la opción *upWhereAll*. A estas alturas debemos estar firmemente convencidos de que la actualización de la fecha de contrato va a fallar:

```
update  Employee
set     HireDate = '1/1/96'    -- Cambiando la fecha
where   EmpNo = 666
  and   FirstName = 'Ian'
  and   LastName = 'Marteens'
  and   Salary = 0             -- ¡Ya no, gracias a Dios!
  and   PhoneExt = '666'
  and   HireDate = '1/1/97'
```

Por supuesto, el usuario recibe una excepción. Sin embargo, observando la salida del SQL Monitor descubrimos que la aplicación lanza la siguiente instrucción:

```
select EmpNo, FirstName, LastName, Salary, PhoneExt, HireDate
from   Employee
where  EmpNo = 666             -- Sólo emplea la clave primaria
```

La aplicación ha releído nuestros datos, pero los valores actuales no aparecen en pantalla. ¿Para qué ha hecho esto, entonces? El primer usuario, el que insiste en corregir nuestra antigüedad, es un individuo terco. La fila activa de su tabla de empleados sigue en modo *dsEdit*, por lo cual puede repetir la operación *Post*. Esta es la nueva sentencia enviada:

```
update  Employee
set     HireDate = '1/1/96',  -- Cambiando la fecha
        Salary = 0            -- ¡No es justo!
where   EmpNo = 666
  and   FirstName = 'Ian'
  and   LastName = 'Marteens'
  and   Salary = 1000000      -- Nos pilló
  and   PhoneExt = '666'
  and   HireDate = '1/1/97'
```

Es decir, al releer el registro, la aplicación está en condiciones de reintentar la grabación del registro que está en modo de edición. Además, se produce algo negativo: esta última grabación machaca cualquier cambio realizado concurrentemente desde otro puesto. Tenemos entonces un motivo adicional para utilizar *upWhereChanged*, pues como es lógico, este tipo de situaciones no se producen en este modo de actualización.

> Por desgracia, no tenemos propiedades que nos permitan conocer los valores actuales del registro durante el evento *OnPostError*. Si revisamos la documentación, encontraremos propiedades *OldValue*, *CurValue* y *NewValue* para cada campo. Sí, estas propiedades son precisamente las que nos hacen falta para diseñar manejadores de errores inteligentes ... pero solamente podemos utilizarlas si están activas las actualizaciones en caché, o si estamos trabajando con clientes Midas. Tendremos que esperar un poco más.

El método *Edit* sobre tablas cliente/servidor lanza una instrucción similar, para leer los valores actuales del registro activo. Puede suceder que ese registro que tenemos en nuestra pantalla haya sido eliminado por otro usuario, caso en el cual se produce el error correspondiente, que puede atraparse en el evento *OnEditError*. Este comportamiento de *Edit* puede aprovecharse para implementar un equivalente de *Refresh*, pero que solamente afecte al registro activo de un conjunto de datos:

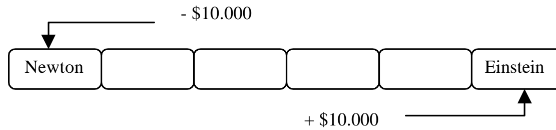
```
Table1.Edit;
Table1.Cancel;
```

Es necesario llamar a *Cancel* para restaurar la tabla al modo *dsBrowse*.

Las propiedades "ácidas" de una transacción

Hasta aquí hemos analizado el comportamiento de Delphi para garantizar la consistencia de las operaciones de actualización. Desgraciadamente, este mecanismo solamente vale para actualizaciones aisladas, que involucran un solo registro de la base de datos a la vez. Cuando la consistencia de la base de datos depende del éxito de actualizaciones sincronizadas, las técnicas estudiadas se nos quedan cortas.

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[26].

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.

Transacciones SQL y en bases de datos locales

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 ter-

[26] 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...

mina. Todos los sistemas SQL ofrecen las siguientes instrucciones para marcar el principio y fin de una transacción:

```
start transaction
commit work
rollback work
```

La primera instrucción 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.

La implementación de transacciones para tablas locales (dBase y Paradox) es responsabilidad del BDE. Las versiones de 16 bits del BDE no ofrecen esta posibilidad. A partir de la versión 3.0 del BDE que apareció con Delphi 2, se soportan las llamadas *transacciones locales*. Esta implementación es un poco limitada, pues no permite deshacer operaciones del lenguaje DDL (**create table**, o **drop table**, por ejemplo), y la independencia entre transacciones es bastante pobre. Tampoco pueden activarse transacciones sobre tablas de Paradox que no tengan definida una clave primaria. Y hay que tener en cuenta la posibilidad de que al cerrar una tabla no se puedan deshacer posteriormente los cambios realizados en la misma, aunque aún no se haya confirmado la transacción.

Transacciones implícitas y explícitas

Como ya el lector se habrá dado cuenta, el modo de trabajo habitual de Delphi 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, las cosas son distintas.

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;
```

Después de una llamada a *Rollback* es aconsejable realizar una operación *Refresh* sobre todas las tablas abiertas de la base de datos, para releer los datos y actualizar la panta-

lla. Considerando que la base de datos lleva cuenta de los conjuntos de datos activos, se puede automatizar esta operación:

```
procedure CancelarCambios(ADatabase: TDatabase);
var
   I: Integer;
begin
   ADatabase.Rollback;
   for I := ADatabase.DatasetCount - 1 downto 0 do
      ADatabase.Datasets[I].Refresh;
end;
```

La propiedad *InTransaction*, disponible en tiempo de ejecución, nos avisa si hemos iniciado alguna transacción sobre la base de datos activa. Esta es una propiedad disponible a partir de Delphi 2.

```
procedure TForm1.TransaccionClick(Sender: TObject);
begin
   IniciarTransaccion1.Enabled := not Database1.InTransaction;
   ConfirmarTransaccion1.Enabled := Database1.InTransaction;
   CancelarTransaccion1.Enabled := Database1.InTransaction;
end;
```

La transferencia bancaria que mencionamos antes puede 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
   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.

> Si está trabajando con InterBase, quizás le interese modificar el parámetro *DRIVER FLAGS* en el controlador. 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.

Entrada de datos y transacciones

Se puede aprovechar el carácter atómico de las transacciones para automatizar el funcionamiento de los diálogos de entrada de datos que afectan simultáneamente a varias tablas de una misma base de datos, del mismo modo que lo hemos logrado con las actualizaciones sobre una sola tabla. Si el lector recuerda, para este tipo de actualizaciones utilizábamos la siguiente función, que llamábamos desde la respuesta al evento *OnCloseQuery* del cuadro de diálogo:

```
function PuedoCerrar(AForm: TForm; ATable: TDataSet): Boolean;
begin
   Result := True;
   if AForm.ModalResult = mrOk then
      ATable.Post
   else if not ATable.Modified
      or (Application.MessageBox('¿Desea abandonar los cambios?',
         'Atención', MB_ICONQUESTION + MB_YESNO) = IdYes) then
      ATable.Cancel
   else
      Result := False;
end;
```

La generalización del algoritmo de cierre del cuadro de diálogo es inmediata. Observe el uso que se hace de *CheckBrowseMode*, para garantizar que se graben los cambios pendientes.

```
function PuedoCerrarTrans(AForm: TForm;
   Tables: array of TDBDataSet): Boolean;
var
   I: Integer;
begin
   Result := True;
   if AForm.ModalResult = mrOk then
   begin
      for I := 0 to High(Tables) do Tables[I].CheckBrowseMode;
      Tables[0].Database.Commit;
   end
   else if Application.MessageBox('¿Desea abandonar los cambios?',
      'Atención', MB_ICONQUESTION + MB_YESNO) = IdYes then
   begin
      for I := 0 to High(Tables) do Tables[I].Cancel;
      Tables[0].Database.Rollback;
   end
   else
      Result := False;
end;
```

Esta función se debe llamar desde el evento *OnCloseQuery* del diálogo de la siguiente manera:

```
procedure TdlgPedidos.FormCloseQuery(Sender: TObject;
   var CanClose: Boolean);
begin
   with modDatos do
      CanClose := PuedoCerrarTrans(Self, [tbPedidos, tbDetalles]);
end;
```

Ahora hay que llamar explícitamente a *StartTransaction* antes de comenzar la edición o inserción:

```
procedure TPrincipal.AltaPedidos(Sender: TObject);
begin
   with modDatos do
   begin
      tbPedidos.Database.StartTransaction;
      tbPedidos.Append;
      dlgPedidos.ShowModal;
   end;
end;
```

Es un poco incómodo tener que iniciar explícitamente la transacción cada vez que se active el cuadro de diálogo. También hemos perdido la posibilidad de detectar fácilmente si se han realizado modificaciones en algunas de la tablas involucradas; es posible que algunos de los cambios realizados hayan sido enviados con el método *Post* a la base de datos, como sucede frecuentemente en la edición *master/detail*. De esta forma, si activamos por error el formulario de entrada de datos y pulsamos la tecla ESC inmediatamente, obtenemos una inmerecida advertencia sobre las consecuencias de abandonar nuestros datos a su suerte.

Más adelante estudiaremos las *actualizaciones en caché*, un mecanismo que nos ayudará, entre otras cosas, a resolver éstos y otros inconvenientes menores.

Aislamiento de transacciones

Las propiedades atómicas de las transacciones valen tanto para sistemas multipuestos como para 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 cumplirse 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.

En cuanto al segundo requerimiento, ya hemos hablado acerca del mismo 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).

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. 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.

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.

Delphi tiene previstos estos tres niveles de aislamiento de transacciones, que se configuran en la propiedad *TransIsolation* de los objetos de tipo *TDatabase*. Los valores posibles para esta propiedad son:

Constante	Nivel de aislamiento
tiDirtyRead	Lee cambios sin confirmar
tiReadCommited	Lee solamente cambios confirmados
tiRepeatableRead	Los valores leídos no cambian durante la transacción

El valor por omisión de *TransIsolation* es *tiReadCommited*. 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 *tiReadCommited*. Por otra parte, 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.

Aislamiento de transacciones mediante 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 utilizan técnicas basadas en bloqueos. Ya hemos visto cómo se utilizan los bloqueos para garantizar el acceso exclusivo a registros en los sistemas con control de concurrencia pesimista. Ahora veremos cómo se adapta este mecanismo de sincronización a la implementación de transacciones.

La idea básica es que una transacción va colocando bloqueos en los registros sobre los que actúa, y estos bloqueos no se liberan hasta que la transacción termina, de un modo u otro. El algoritmo más sencillo consiste en pedir un bloqueo para cada registro que se va a modificar. Este bloqueo, una vez concedido, debe impedir el acceso al registro afectado, tanto para leer como para modificar su valor. La consecuencia inmediata de esto es que una vez modificado un registro, ninguna otra transacción tiene acceso al valor modificado hasta que la transacción termine con **commit** ó **rollback**. De esta manera, no sólo se impiden modificaciones superpuestas sino que también evitamos las lecturas sucias.

Para resolver el problema de las lecturas repetibles necesitamos, 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

Cuando una transacción va a leer un registro, coloca primeramente 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.

> La implementación de las lecturas repetibles por el SQL Link de Oracle es algo diferente al esquema explicado. Oracle saca una versión sólo lectura de los datos leídos en la transacción, y no permite actualizaciones a la aplicación que solicita este nivel de aislamiento.

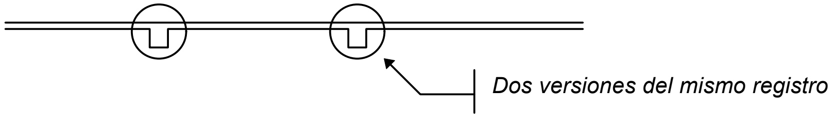
El jardín de los senderos que se bifurcan

InterBase resuelve estos problemas 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.

Volvemos 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.



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.

De nuevo los optimistas

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. Ya hemos visto una clasificación similar para las estrategias utilizadas en la edición de registros. Habíamos analizado las ventajas y desventajas de cada una: las técnicas optimistas eran mejores en el sentido de que provocaban menos tráfico de red, y menos restricciones en el acceso a registros. Pero desde el

punto de vista del usuario podía ser un poco incómodo descubrir, al final de una larga y tediosa operación, que sus actualizaciones entraban en conflicto con las de otro usuario.

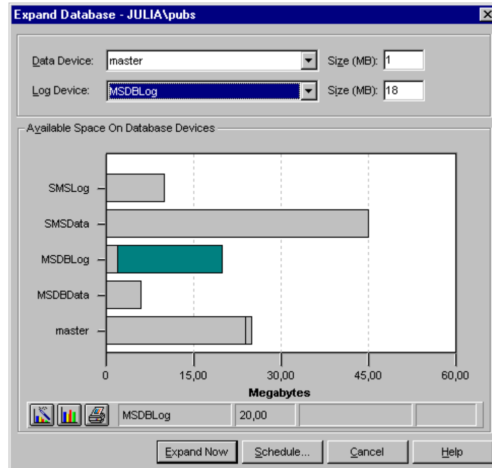
Sin embargo, cuando se trata de optimismo y pesimismo en relación con la implementación del aislamiento entre transacciones, las ventajas de una estrategia optimista son realmente abrumadoras. Los sistemas basados en bloqueos han sido diseñados y optimizados con la mente puesta en un tipo de transacciones 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!

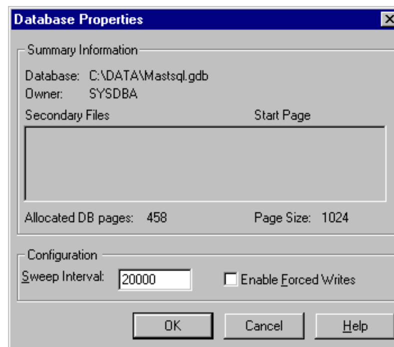
En un sistema basado en bloqueos las transacciones OLTP y DSS tienen una difícil coexistencia. En la práctica, un sistema de este tipo debe "desconectar" la base de datos para poder efectuar la copia de seguridad ("Su operación no puede efectuarse en estos momentos", me dice la verde pantalla de mi cajero). 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.

Hay otro aspecto a tener en cuenta al evaluar estas técnicas, y es el modo en el que se garantiza la atomicidad de las operaciones. Lo más frecuente es encontrar implementaciones basadas en *registros de transacciones* (*transaction logs*). Estos registros de transacciones son ficheros en los cuales se graban secuencialmente las operaciones necesarias para deshacer las transacciones no terminadas. Se trata de una estructura com-

plicada de mantener. Si la base de datos falla durante una transacción, este fichero debe "ejecutarse" para restablecer un estado válido de la base de datos. La figura siguiente muestra el diálogo que aumenta el tamaño máximo reservado para el registro de transacciones de MS SQL Server.



En contraste, 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 tran-

sacciones 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.

Capítulo

32

Actualizaciones en caché

LAS ACTUALIZACIONES EN CACHÉ SON UN RECURSO de las versiones de 32 bits del BDE para aumentar el rendimiento de las transacciones en entornos cliente/servidor. Los conjuntos de datos de Delphi vienen equipados con una propiedad, *CachedUpdates*, que decide si los cambios efectuados en el conjunto de datos son grabados inmediatamente en la base de datos o si se almacenan en memoria del ordenador cliente y se envían en bloque al servidor, a petición del programa cliente, en un momento dado.

En este capítulo estudiaremos las características básicas de las actualizaciones en caché, y cómo se pueden aplicar a la automatización de los procesos de entrada de datos. Al final, veremos cómo aprovechar esta técnica para mejorar la actualizabilidad de consultas en entornos cliente/servidor y para minimizar el impacto sobre el usuario de los bloqueos optimistas.

¿Caché para qué?

¿Qué nos aporta este intrincado mecanismo? En primer lugar, mediante este recurso una transacción que requiera interacción con el usuario puede hacerse más corta. Y, como explicamos en el capítulo anterior, una transacción mientras más breve, mejor. Por otra parte, las actualizaciones en caché pueden disminuir drásticamente el número de paquetes enviados por la red. Cuando no están activas las actualizaciones en caché, cada registro grabado provoca el envío de un paquete de datos. Cada paquete va precedido de cierta información de control, que se repite para cada envío. Además, estos paquetes tienen un tamaño fijo, y lo más probable es que se desaproveche parte de su capacidad. También se benefician aquellos sistemas SQL que utilizan internamente técnicas pesimistas de bloqueos para garantizar las lecturas repetibles. En este caso, los bloqueos impuestos están activos mucho menos tiempo, durante la ejecución del método *ApplyUpdates*. De este modo, se puede lograr en cierto modo la simulación de un mecanismo optimista de control de concurrencia.

Las ventajas mencionadas se aplican fundamentalmente a los entornos cliente/servidor. Pero también es correcto el uso de actualizaciones en caché para bases de

datos locales. Esta vez la razón es de simple conveniencia para el programador, y tiene que ver nuevamente con la modificación e inserción de objetos complejos, representados en varias tablas. Como vimos con anterioridad, se pueden utilizar las transacciones para lograr la atomicidad de estas operaciones. Pero para programar una transacción necesitamos iniciarla y confirmarla o deshacerla explícitamente, mientras que, como veremos, una actualización en caché no necesita ser iniciada: en cada momento existe sencillamente un conjunto de actualizaciones realizadas desde la última operación de confirmación. Todo lo cual significa menos trabajo para el programador.

En particular, si intentamos utilizar transacciones sobre bases de datos locales, tenemos que enfrentarnos al límite de bloqueos por tabla (100 para dBase y 255 para Paradox), pues las transacciones no liberan los bloqueos hasta su finalización. Si en vez de utilizar directamente las transacciones locales utilizamos actualizaciones en caché, la implementación de las mismas por el BDE sobrepasa esta restricción escalando automáticamente el nivel de los bloqueos impuestos.

La historia, sin embargo, no acaba aquí. Al seguir estando disponibles los datos originales de un registro después de una modificación o borrado, tenemos a nuestro alcance nuevas posibilidades: la selección de registros de acuerdo a su estado de actualización y la cancelación de actualizaciones registro a registro. Incluso podremos implementar algo parecido a los borrados lógicos de dBase.

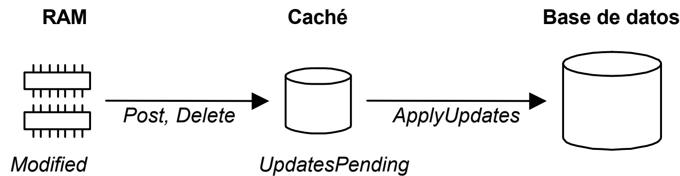
> Me permitiré una breve disgresión lingüística sobre nuestro neologismo "caché". Como sucede con muchas de las palabras incorporadas al castellano que están relacionadas con la informática, esta incorporación ha sido incorrecta. El problema es que pronunciamos "caché", acentuando la palabra en la última sílaba. En inglés, sin embargo, esta palabra se pronuncia igual que *cash*, y su significado es esconder; en particular, puede significar un sitio oculto donde se almacenan provisiones. Es curioso que, en francés, existe la expresión *cache-cache* que se pronuncia como en inglés y quiere decir "el juego del escondite" (*cacher* es ocultar).

Activación de las actualizaciones en caché

La activación del mecanismo se logra asignando el valor *True* a la propiedad *CachedUpdates* de las tablas o consultas. El valor de esta propiedad puede cambiarse incluso estando la tabla activa. Como las actualizaciones en caché utilizan internamente transacciones para su confirmación, si se efectúan sobre tablas locales, la propiedad *TransIsolation* de la correspondiente base de datos debe valer *tiDirtyRead*.

Se puede conocer si existen actualizaciones en la caché, pendientes de su confirmación definitiva, utilizando la propiedad de tipo lógico *UpdatesPending* para cada tabla o

consulta. Observe que la propiedad *UpdatesPending* solamente 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 *TTable* que se refiera a la misma tabla física, los cambios realizados en la primera tabla 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.

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 TDBDataSets);
```

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 TDataSet.ApplyUpdates;
procedure TDataSet.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 descarta las actualizaciones pendientes en caché. ¿Por qué necesitamos un protocolo de dos fases? El problema es que, si realizamos actualizaciones sobre varias tablas, y pretendemos grabarlas atómicamente, tenemos 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 falibles a la primera fase, la llamada a *ApplyUpdates*; por el contrario, *CommitUpdates* no debe fallar nunca, a pesar de Murphy.

La división en dos fases la aprovecha el método *ApplyUpdates* de la clase *TDatabase*. Para aplicar las actualizaciones pendientes de una lista de tablas, la base de datos inicia una transacción e intenta llamar a los métodos *ApplyUpdates* individuales de cada conjunto de datos. Si falla alguno de éstos, 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 := 0 to High(Tablas) do
      Tablas[I].ApplyUpdates;         // Pueden fallar
   Commit;
except
   Rollback;
   raise;                             // Propagar la excepción
end;
for I := 0 to High(Tablas) do
   Tablas[I].CommitUpdates;           // Nunca fallan
```

Es recomendable llamar siempre al método *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. No obstante, es posible aprovechar estos procedimientos de más bajo nivel en circunstancias especiales, como puede suceder cuando queremos coordinar actualizaciones en caché sobre dos bases de datos diferentes.

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]);
```

Marcha atrás

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.

El siguiente procedimiento muestra una forma sencilla de descartar cambios en una lista de conjuntos de datos:

```
procedure DescartarCambios(const DataSets: array of TDataSets);
var
   I: Integer;
begin
   for I := 0 to High(DataSets) do
      DataSets[I].CancelUpdates;
end;
```

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 propiedad *UpdateStatus* de un conjunto de datos indica el tipo de la última actualización realizada sobre el registro activo. *UpdateStatus* puede tomar los siguientes valores:

Valor	Significado
usUnmodified	El registro no ha sufrido actualizaciones
usModified	Se han realizado modificaciones sobre el registro
usInserted	Este es un registro nuevo
usDeleted	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. Por ejemplo:

```
procedure TForm1.DBGrid1DrawColumnCell(Sender: TObject;
   const Rect: TRect; DataCol: Integer; Column: TColumn;
   State: TGridDrawState);
begin
   with Sender as TDBGrid do
   begin
      case DataSource.DataSet.UpdateStatus of
         usModified: Canvas.Font.Style := [fsBold];
         usInserted: Canvas.Font.Style := [fsItalic];
         usDeleted: Canvas.Font.Style := [fsStrikeOut];
      end;
      DefaultDrawColumnCell(Rect, DataCol, Column, State);
   end;
end;
```

También puede aprovecharse la propiedad para deshacer algún tipo de cambios en la tabla:

```
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;
```

En el siguiente epígrafe veremos un método más sencillo de descartar actualizaciones selectivamente.

El filtro de tipos de registros

¿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
rtModified	Mostrar los registros modificados
rtInserted	Mostrar los nuevos registros
rtDeleted	Mostrar los registros eliminados
rtUnModified	Mostrar los registros no modificados

Inicialmente, esta propiedad contiene el siguiente valor, que muestra todos los tipos de registros, con excepción de los borrados:

```
[rtModified, rtInserted, rtUnModified]
```

Si por algún motivo asignamos el conjunto vacío a esta propiedad, el valor de la misma se restaura a este valor por omisión:

```
Table1.UpdateRecordTypes := [];
if Table1.UpdateRecordTypes = [rtModified,
   rtInserted, rtUnModified] then
   ShowMessage('UpdateRecordTypes restaurado al valor por omisión');
```

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 automáticamente el registro recuperado desaparece 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; esto queda como ejercicio para el lector.

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.

Entonces necesitamos un menú. Estas son las opciones que incluiremos:

Caché	**V**er
Grabar	**O**riginales
Cancelar	**M**odificados
Cancelar **a**ctual	**N**uevos
	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.miAplicarClick(Sender: TObject);
begin
   // 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 de comentario 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*:

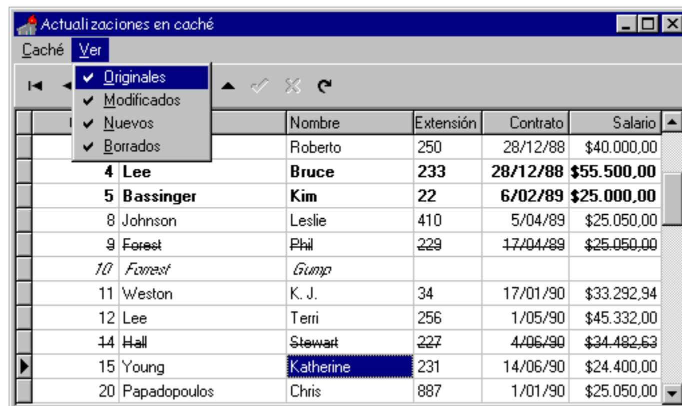
```
procedure TForm1.miCacheClick(Sender: TObject);
begin
   miAplicar.Enabled := Table1.UpdatesPending;
   miCancelar.Enabled := Table1.UpdatesPending;
   miCancelarActual.Enabled := Table1.UpdateStatus <> usUnModified;
end;
```

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 que hemos analizado antes, para visualizar el estado de actualización de cada registro mostrado.

El Gran Final: edición y entrada de datos

Podemos automatizar los métodos de entrada de datos en las aplicaciones que tratan con objetos complejos, del mismo modo que ya lo hemos hecho con la edición de objetos representables en una sola tabla. En el capítulo sobre transacciones habíamos desarrollado la función *PuedoCerrarTrans*, que cuando se llamaba desde la respuesta al evento *OnCloseQuery* de la ventana de entrada de datos, se ocupaba de guardar o cancelar automáticamente las grabaciones efectuadas sobre un conjunto de tablas. Esta era la implementación lograda entonces:

```
function PuedoCerrarTrans(AForm: TForm;
   Tables: array of TDBDataSet): Boolean;
var
   I: Integer;
begin
   Result := True;
   if AForm.ModalResult = mrOk then
   begin
      for I := 0 to High(Tables) do Tables[I].CheckBrowseMode;
      Tables[0].Database.Commit;
   end
   else if Application.MessageBox('¿Desea abandonar los cambios?',
      'Atención', MB_ICONQUESTION + MB_YESNO) = IdYes then
   begin
      for I := 0 to High(Tables) do Tables[I].Cancel;
      Tables[0].Database.Rollback;
   end
   else
      Result := False;
end;
```

El uso de esta rutina se complicaba, porque antes de ejecutar el cuadro de diálogo había que iniciar manualmente una transacción, además de poner la tabla principal en modo de inserción o edición. Teníamos el inconveniente adicional de no saber si se habían realizado modificaciones durante la edición, al menos sin programación adicional; esto causaba que la cancelación del diálogo mostrase cada vez un incómodo mensaje de confirmación.

En cambio, si utilizamos actualizaciones en caché obtenemos los siguientes beneficios:

- No hay que llamar explícitamente a *StartTransaction*.

- Combinando las propiedades *Modified* y *UpdatesPending* de los conjuntos de datos involucrados en la operación, podemos saber si se han realizado modificaciones, ya sea a nivel de campo o de caché.
- El uso de *StartTransaction* disminuye la posibilidad de acceso concurrente, pues por cada registro modificado durante la transacción, el sistema coloca un bloqueo que solamente es liberado cuando cerramos el cuadro de diálogo. Con las actualizaciones en caché, la transacción se inicia y culmina durante la respuesta a los botones de finalización del diálogo.

Esta es la función que sustituye a la anterior:

```
function PuedoCerrar(AForm: TForm;
   const Tables: array of TDBDataSet): Boolean;
var
   I: Integer;
   Actualizar: Boolean;
begin
   Result := True;
   // Verificamos si hay cambios en caché
   Actualizar := False;
   for I := 0 to High(Tables) do
      if Tables[I].UpdatesPending or Tables[I].Modified then
         Actualizar := True;
   // Nos han pedido que grabemos los cambios
   if AForm.ModalResult = mrOk then
      Tables[0].Database.ApplyUpdates(Tables)
   // Hay que deshacer los cambios
   else if not Actualizar
      or (Application.MessageBox('¿Desea abandonar los cambios?',
         'Atención', MB_ICONQUESTION + MB_YESNO) = IdYes) then
      for I := 0 to High(Tables) do
      // CancelUpdates llama a Cancel si es necesario
         Tables[I].CancelUpdates
   // El usuario se arrepiente de abandonar los cambios
   else
      Result := False;
end;
```

Se asume que todas las tablas están conectadas a la misma base de datos; la base de datos en cuestión se extrae de la propiedad *Database* de la primera tabla de la lista.

Si, por ejemplo, el formulario de entrada de datos *entPedidos* realiza modificaciones en las tablas *tbPedidos*, *tbDetalles*, *tbClientes* y *tbArticulos* (*orders*, *items*, *customer* y *parts*) pertenecientes al módulo de datos *modDatos* y conectadas a la misma base de datos, podemos asociar el siguiente método al evento *OnCloseQuery* de la ventana en cuestión:

```
procedure entPedidos.entPedidosCloseQuery(Sender: TObject;
   var CanClose: Boolean);
begin
   with modDatos do
      CanClose := PuedoCerrar(Self,
         [tbPedidos, tbDetalles, tbClientes, tbArticulos]);
end;
```

El orden en que se pasan las tablas a esta función es importante. Tenga en cuenta que para grabar una línea de detalles en el servidor tiene que existir primeramente la cabecera del pedido, debido a las restricciones de integridad referencial. Por lo tanto, la porción de la caché que contiene esta cabecera debe ser grabada antes que la porción correspondiente a las líneas de detalles.

Combinando la caché con grabaciones directas

Resumiendo lo que hemos explicado a lo largo de varios capítulos, tenemos en definitiva tres técnicas para garantizar la atomicidad de las actualizaciones de objetos complejos:

- La edición en memoria de los datos del objeto, y su posterior grabación durante una transacción.
- La edición con controles *data-aware*, activando una transacción al comenzar la operación.
- El uso de actualizaciones en caché.

De entrada, trataremos de descartar la primera técnica siempre que sea posible. La segunda técnica nos permite evitar la duplicación de verificaciones, y nos libera del problema de la transferencia de datos a las tablas. Pero no es aconsejable usar transacciones durante la edición, pues los bloqueos impuestos se mantienen todo el tiempo que el usuario necesita para la edición. Las actualizaciones en caché evitan el problema de los bloqueos mantenidos durante largos intervalos, además de que el usuario tampoco tiene que duplicar restricciones. El uso automático de transacciones durante la grabación de la caché nos asegura la atomicidad de las operaciones. Pero el comportamiento exageradamente "optimista" de las actualizaciones en caché puede plantearle dificultades al usuario y al programador.

Utilizaré como ejemplo el sistema de entrada de pedidos. Supongamos que las tablas de pedidos y las líneas de detalles tienen la caché activa. Como estas tablas se utilizan para altas, no hay problemas con el acceso concurrente a los registros generados por la aplicación. Ahora bien, si nuestra aplicación debe modificar la tabla de inventario, *tbArticulos*, al realizar altas, ya nos encontramos en un aprieto. Si utilizamos actualizaciones en caché, no podremos vender Coca-Cola simultáneamente desde dos terminales, pues estas dos terminales intentarán modificar el mismo registro con dos versiones diferentes durante la grabación de la caché, generando un error de bloqueo optimista. En cambio, si no utilizamos caché, no podemos garantizar la atomicidad de la grabación: si, en la medida que vamos introduciendo y modificando líneas de detalles, actualizamos el inventario, estas grabaciones tienen carácter *definitivo*. La solución completa tampoco es realizar estas modificaciones en una transacción *aparte*,

posterior a la llamada a *ApplyUpdates*, pues al ser transacciones independientes no se garantiza el éxito o fracaso simultáneo de ambas.

A estas alturas, el lector se habrá dado cuenta de que, si estamos trabajando con un sistema SQL, la solución es muy fácil: implementar los cambios en la tabla de artículos en un *trigger*. De este modo, la modificación de esta tabla ocurre al transferirse el contenido de la caché a la base de datos, durante la misma transacción. Así, todas la grabaciones tienen éxito, o ninguna.

¿Y qué pasa con los pobres mortales que están obligados a seguir trabajando con Paradox, dBase, FoxPro y demás engendros locales? Para estos casos, necesitamos ampliar el algoritmo de escritura de la caché, de manera que se puedan realizar grabaciones directas durante la transacción que inicia *ApplyUpdates*. Delphi no ofrece soporte directo para esto, pero es fácil crear una función que sustituya a *ApplyUpdates*, aplicada a una base de datos. Y una de las formas de implementar estas extensiones es utilizando punteros a métodos:

```
procedure ApplyUpdatesEx(const DataSets: array of TDBDataSet;
   BeforeApply, BeforeCommit, AfterCommit: TNotifyEvent);
var
   I: Integer;
   DB: TDatabase;
begin
   DB := DataSets[0].Database;
   DB.StartTransaction;
   try
      if Assigned(BeforeApply) then BeforeApply(DB);
      for I := 0 to High(DataSets) do
         DataSets[I].ApplyUpdates;
      if Assigned(BeforeCommit) then BeforeCommit(DB);
      DB.Commit;
   except
      DB.Rollback;
      raise;
   end;
   for I := 0 to High(DataSets) do
      DataSets[I].CommitUpdates;
   if Assigned(AfterCommit) then AfterCommit(DB);
end;
```

Los punteros a métodos *BeforeApply*, *BeforeCommit* y *AfterCommit* se han declarado como pertenecientes al tipo *TNotifyEvent* por comodidad y conveniencia. Observe que cualquier escritura que se realice en los métodos *BeforeApply* y *BeforeCommit* se realiza dentro de la misma transacción en la que se graba la caché. La llamada a *AfterCommit*, en cambio, se realiza después del bucle en que se vacía la caché, utilizando el método *CommitUpdates*. Esto lo hemos programado así para evitar que una excepción producida en la llamada a este evento impida el vaciado de la caché.

> He diferenciado entre *BeforeApply* y *BeforeCommit* por causa de un comportamiento algo anómalo de las actualizaciones en caché: cuando se ha llamado a *ApplyUpdates*, pero todavía no se ha vaciado la caché de la tabla con *CommitUpdates*, los registros modificados aparecen dos veces dentro del cursor de la tabla. Si necesitamos un método que recorra alguna de las tablas actualizadas, como el que veremos dentro de poco, debemos conectarlo a *BeforeApply* mejor que a *BeforeCommit*.

Si estamos utilizando el evento *OnCloseQuery* de la ficha para automatizar la grabación y cancelación de cambios, tenemos que extender la función *PuedoCerrar* para que acepte los punteros a métodos como parámetros. He incluido también la posibilidad de realizar la entrada de datos en modo continuo, como se ha explicado en el capítulo sobre actualizaciones. Esta es la nueva versión:

```
function PuedoCerrar(AForm: TForm;
   const Tables: array of TDBDataSet;
   ModoContinuo: Boolean;
   BeforeApply, BeforeCommit, AfterCommit: TNotifyEvent): Boolean;
var
   I: Integer;
   Actualizar: Boolean;
begin
   Result := True;
   // Verificamos si hay cambios en caché
   Actualizar := False;
   for I := 0 to High(Tables) do
      if Tables[I].UpdatesPending or Tables[I].Modified then
         Actualizar := True;
   // Nos han pedido que grabemos los cambios
   if AForm.ModalResult = mrOk then
   begin
      ApplyUpdatesEx(Tables, BeforeApply, BeforeCommit,
         AfterCommit);
      if ModoContinuo then
      begin
         Result := False;
         Tables[0].Append;
      end;
   end
   // Hay que deshacer los cambios
   else if not Actualizar
      or (Application.MessageBox('¿Desea abandonar los cambios?',
         'Atención', MB_ICONQUESTION + MB_YESNO) = IdYes) then
      for I := 0 to High(Tables) do
         Tables[I].CancelUpdates
   // El usuario se arrepiente de abandonar los cambios
   else
      Result := False;
end;
```

Para implementar la entrada de datos continua se ha supuesto que la primera tabla del vector es la tabla principal, a partir de la cual se desarrolla todo el proceso de actualización.

Ahora volvemos al ejemplo que motivó estas extensiones. Supongamos que tenemos una ficha de entrada de pedidos y queremos actualizar el inventario, en la tabla *tbArticulos*, una vez grabado el pedido. La solución consiste en declarar un método público en el módulo de datos que realice los cambios en la tabla de artículos de acuerdo a las líneas de detalles del pedido activo:

```
procedure TmodDatos.ActualizarInventario(Sender: TObject);
begin
   tbDetalles.First;
   while not tbDetalles.EOF do
   begin
      if tbArticulos.Locate('PartNo', tbDetalles['PartNo'], []) then
      begin
         tbArticulos.Edit;
         tbArticulos['OnOrder'] := tbArticulos['OnOrder'] +
            tbDetalles['Qty'];
         tbArticulos.Post;
      end;
      tbDetalles.Next;
   end;
end;
```

Luego, en la ficha de altas de pedidos modificamos la respuesta al evento *OnCloseQuery* de esta forma:

```
procedure entPedidos.entPedidosCloseQuery(Sender: TObject;
   var CanClose: Boolean);
begin
   with modDatos do
      CanClose := PuedoCerrar(Self, [tbPedidos, tbDetalles], True,
         ActualizarInventario, nil, nil);
end;
```

Hemos especificado el puntero vacío **nil** como valor para los parámetros *BeforeCommit* y *AfterCommit*. Más adelante veremos cómo este último parámetro puede aprovecharse para realizar la impresión de los datos del pedido una vez que se ha confirmado su entrada.

> Los sistemas cliente/servidor también pueden obtener beneficios de esta técnica. En el mismo ejemplo de altas de pedidos podemos utilizar para actualizar el inventario un procedimiento almacenado que se ejecutaría después de haber grabado la caché, en vez de apoyarnos en el uso de *triggers*. Una ventaja adicional es que minimizamos la posibilidad de un *deadlock*, o abrazo mortal.

Prototipos y métodos virtuales

Realmente, el uso de punteros a métodos es cuando menos engorroso, pero es necesario mientras *PuedoCerrar* sea una función, no un método. Una forma de simplificar

esta metodología de trabajo es definiendo un prototipo o plantilla de formulario de entrada de datos, que se utilice como clase base para todas las altas y modificaciones de objetos complejos. En este caso, *PuedoCerrar* puede definirse como un método de esta clase, y los punteros a métodos sustituirse por métodos virtuales.

Partamos de un nuevo formulario, al cual configuraremos visualmente como un cuadro de diálogo, y al cual añadiremos los siguientes métodos en su declaración de clase:

```
type
   TwndDialogo = class(TForm)
      // …
   protected
      procedure BeforeApply(DB: TDatabase); virtual;
      procedure BeforeCommit(DB: TDatabase); virtual;
      procedure AfterCommit(DB: TDatabase); virtual;
      procedure ApplyUpdatesEx(const DataSets: array of TDBDataSet);
      function PuedoCerrar(const DataSets: array of TDBDataSet;
         ModoContinuo: Boolean): Boolean;
      // …
   end;
```

En principio, los tres métodos virtuales se definen con cuerpos vacíos:

```
procedure TwndDialogo.BeforeApply(DB: TDatabase);
begin
end;

procedure TwndDialogo.BeforeCommit(DB: TDatabase);
begin
end;

procedure TwndDialogo.AfterCommit(DB: TDatabase);
begin
end;
```

La nueva implementación de *ApplyUpdatesEx* ejecuta a estos métodos:

```
procedure TwndDialogo.ApplyUpdatesEx(
   const DataSets: array of TDBDataSet);
var
   I: Integer;
   DB: TDatabase;
begin
   DB := DataSets[0].Database;
   DB.StartTransaction;
   try
      BeforeApply(DB);
      for I := 0 to High(DataSets) do
         DataSets[I].ApplyUpdates;
      BeforeCommit(DB);
      DB.Commit;
   except
      DB.Rollback;
```

```
      raise;
    end;
    for I := 0 to High(DataSets) do
      DataSets[I].CommitUpdates;
    AfterCommit(DB);
  end;
```

Finalmente, *PuedoCerrar* se simplifica bastante:

```
function TwndDialogo.PuedoCerrar(
  const DataSets: array of TDBDataSet;
  ModoContinuo: Boolean): Boolean;
var
  I: Integer;
  Actualizar: Boolean;
begin
  Result := True;
  // Verificamos si hay cambios en caché
  I := High(DataSets);
  while (I >= 0) and not DataSets[I].UpdatesPending
    and not DataSets[I].Modified do Dec(I);
  Actualizar := (I >= 0);
  // Nos han pedido que grabemos los cambios
  if ModalResult = mrOk then
  begin
    ApplyUpdatesEx(DataSets);
    if ModoContinuo then
    begin
      Result := False;
      DataSets[0].Append;
    end;
  end
  // Hay que deshacer los cambios
  else if not Actualizar
    or (Application.MessageBox('¿Desea abandonar los cambios?',
      'Atención', MB_ICONQUESTION + MB_YESNO) = IdYes) then
    for I := 0 to High(DataSets) do
      DataSets[I].CancelUpdates
  // El usuario se arrepiente de abandonar los cambios
  else
    Result := False;
end;
```

Partiendo de este prototipo, el programador puede derivar de él los formularios de entrada de datos, redefiniendo los métodos virtuales cuando sea necesario.

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. Antes, en el capítulo 31, hemos visto cómo podíamos modificar el algo-

ritmo de grabación mediante la propiedad *UpdateMode* de una tabla. Bien, ahora podemos ir más lejos, y no solamente con las tablas.

Esto es especialmente útil cuando estamos trabajando con consultas SQL 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. En tal caso, deberíamos modificar ese nombre en todos los registros de clientes que lo 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*:

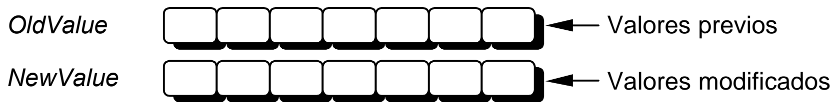
```
type
   TUpdateRecordEvent = procedure(DataSet: TDataSet;
      UpdateKind: TUpdateKind;
      var UpdateAction: TUpdateAction) of object;
```

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
uaFail	Anula la aplicación de las actualizaciones, lanzando una excepción
uaAbort	Aborta la operación mediante la excepción silenciosa
uaSkip	Ignora este registro en particular
uaRetry	No se utiliza en este evento
uaApplied	Actualización exitosa

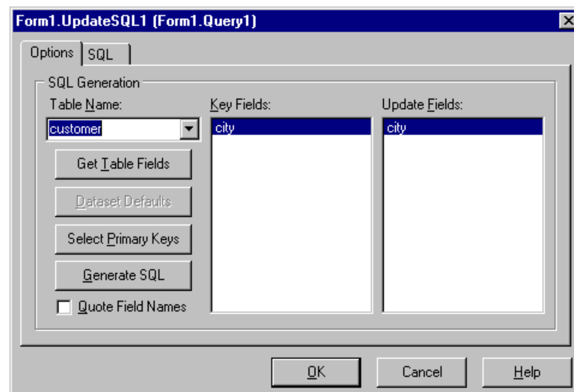
Dentro de la respuesta a este evento, podemos combinar cuantas operaciones de actualización necesitemos, siempre que no cambiemos la fila activa de la tabla que se actualiza. Cuando estamos trabajando con este evento, tenemos acceso a un par de

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. Esta 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, podemos hacer doble clic sobre él para ejecutar un editor de propiedades, que nos ayudará a generar el código de las tres sentencias SQL:



El editor del componente *TUpdateSQL* necesita que le especifiquemos cuál es la tabla que queremos actualizar, pues en el caso de un encuentro tendríamos varias posibilidades. 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:

```
// Borrado
delete from Customer
where  City = :OLD_City

// Modificación
update Customer
set    City = :City
where  City = :OLD_City

// Altas
insert into Customer(City)
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 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);
   // O, alternativamente:
   // UpdateSQL1.SetParams(UpdateKind);
   // UpdateSQL1.ExecSql(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 mediante 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, FDataSet);
   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.

Detección de errores durante la grabación

Cuando se producen errores durante la confirmación de las actualizaciones en caché, éstos no se reportan por medio de los eventos ya conocidos, *OnPostError* y *OnDelete-Error*, sino a través del nuevo evento *OnUpdateError*:

```
procedure TDataModule1.Table1UpdateError(DataSet: TDataSet;
   E: EDatabaseError; UpdateKind: TUpdateKind;
   var UpdateAction: TUpdateAction);
```

La explicación es sencilla: como la grabación del registro se difiere, no es posible verificar violaciones de claves primarias, de rangos, y demás hasta que no se apliquen las actualizaciones.

Como se puede ver, la filosofía de manejo de este evento es similar a la del resto de los eventos de aviso de errores. Se nos indica el conjunto de datos, *DataSet* y la excepción que está a punto de producirse. Se nos indica además qué tipo de actualización se estaba llevando a cabo: si era una modificación, una inserción o un borrado (*UpdateKind*). Por último, mediante el parámetro *UpdateAction* podemos controlar el comportamiento de la operación. Los valores son los mismos que para el evento *OnUpdateRecord*, pero varía la interpretación de dos de ellos:

Valor	**Nuevo significado**
uaRetry	Reintenta la operación sobre este registro
uaApplied	¡No lo utilice en este evento!

A diferencia de lo que sucede con los otros eventos de errores, en este evento tenemos la posibilidad de ignorar el error o de corregir nosotros mismos el problema realizando la actualización. La mayor parte de los ejemplos de utilización de este evento tratan los casos más sencillos: ha ocurrido alguna violación de integridad, la cual es corregida y se reintenta la operación. Sin embargo, el caso más frecuente de error de actualización en la práctica es el fallo de un bloqueo optimista: alguien ha cambiado el registro original mientras nosotros, reflexionando melancólicamente como Hamlet, decidíamos si aplicar o no las actualizaciones. El problema es que, en esta situación, no tenemos forma de restaurar el registro *antiguo* para reintentar la operación. Recuerde que, como explicamos al analizar la propiedad *UpdateMode*, el BDE realiza la actualización de un registro de una tabla SQL mediante una instrucción SQL en la que se hace referencia al registro mediante sus valores anteriores. Para generar la sentencia de actualización, se utilizan los valores almacenados en los campos, en sus propiedades *OldValue*. Estas propiedades son sólo para lectura de modo que, una vez que nos han cambiado el registro original, no tenemos forma de saber en qué lo han transformado.

Sin embargo, en casos especiales, este problema tiene solución. Supongamos que estamos trabajando con la tabla *parts*, el inventario de productos. El conflicto que surge con mayor frecuencia es que alguien vende 10 Coca-Colas, a la par que nosotros intentamos vender otras 15. Las cantidades vendidas se almacenan en la columna *OnOrder*. Evidentemente, se producirá un error si el segundo usuario realiza su transacción después que hemos leído el registro de las Coca-Colas, pero antes de que hayamos grabado nuestra operación. El problema se produce por un pequeño desliz de concepto: a mí no me interesa decir que se han vendido 100+15=115 unidades, sino que se han vendido otras 15. Esta operación puede llevarse a cabo fácilmente mediante una consulta paralela, *Query1*, cuya instrucción SQL sea la siguiente:

```
update Parts
set    OnOrder = OnOrder + :NewOnOrder - :OldOnOrder
where  PartNo = :PartNo
```

En estas circunstancias, la respuesta al evento *OnUpdateError* de la tabla de artículos puede ser la siguiente:

```
procedure TForm1.Table1UpdateError(DataSet: TDataSet;
   E: EDatabaseError; UpdateKind: TUpdateKind;
   var UpdateAction: TUpdateAction);
begin
   if (UpdateKind = ukModify) and (E is EDBEngineError) then
      with EDBEngineError(E).Errors[0] do
         if ErrorCode <> DBIERR_OPTRECLOCKFAILED then
         begin
            Query1.ParamByName('PartNo').AsFloat :=
               Table1.FieldByName('PartNo').OldValue;
            Query1.ParamByName('OldOnOrder').AsInteger :=
               Table1.FieldByName('OnOrder').OldValue;
            Query1.ParamByName('NewOnOrder').AsInteger :=
               Table1.FieldByName('OnOrder').NewValue;
            Query1.ExecSql;
            UpdateAction := uaSkip;
         end;
end;
```

Hemos tenido cuidado en aplicar la modificación únicamente cuando el error es el fallo de un bloqueo optimista; podíamos haber sido más cuidadosos, comprobando que la única diferencia entre las versiones fuera el valor almacenado en la columna *OnOrder*. Un último consejo: después de aplicar los cambios es necesario vaciar la caché, pues los registros que provocaron conflictos han quedado en la misma, gracias a la asignación de *uaSkip* al parámetro *UpdateAction*.

Capítulo

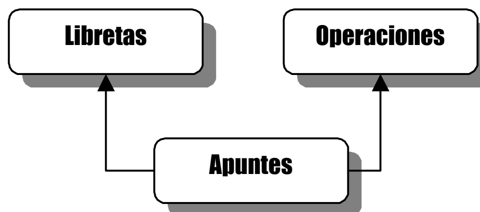
33

Libretas de bancos

NO SE TRATA DE ASALTAR UN BANCO, aunque es probable que cuando termine esta aplicación le venga el deseo de entrar a saco en alguno[27]. Me gustaría finalizar esta parte del libro con un ejemplo sencillo que muestre como integrar muchos de los conceptos que hemos explicado. Nuestro ejemplo trata sobre la administración de un conjunto de libretas de ahorro, en las cuales se van realizando apuntes, mientras se mantiene actualizado el saldo de las mismas.

Descripción del modelo de datos

El modelo de datos contendrá tres tablas: una para las libretas de ahorros, con los datos del titular, la descripción de la libreta, y con determinados valores consolidados auxiliares, como el saldo acumulado y el número de apuntes. Otra tabla servirá para la descripción de las operaciones en las libretas: descripción alfabética de la operación, código numérico y tipo de operación (si es un ingreso o un reintegro). La tercera tabla contendrá todos los apuntes realizados en todas las libretas, asociando apunte con libreta mediante un código de referencia a la libreta. Las relaciones de integridad referencial entre estas tres tablas son las siguientes:



Antes de definir las tablas, es conveniente crear un par de dominios, para representar valores de dinero y valores lógicos:

---

[27] Las noticias de asaltos a bancos son en cierto sentido similares al conocido chiste de "Hombre muerde a perro".

```
create domain boolean as
       char(1) not null
       check (value in ('S', 's', 'N', 'n'));

create domain money as
       integer default 0;
```

La definición de las tablas es muy sencilla:

```
create table Operaciones (
       Codigo         int not null,
       Descripcion    varchar(30) not null,
       Ingreso        boolean,

       primary key    (Codigo),
       unique         (Descripcion)
);

create table Libretas (
       Codigo         int not null,
       Titular varchar(30) not null,
       Descripcion    varchar(50) not null,
       Saldo          money,
       NumApuntes     int default 0,

       primary key    (Codigo)
);
create index TitularLibreta on Libretas(Titular);

create table Apuntes (
       Codigo         int not null,
       Libreta        int not null,
       Numero         int not null,
       Fecha          date default "Now" not null,
       CodOp          int not null,
       Importe        money not null,
       Saldo          money,

       primary key    (Codigo),
       unique         (Libreta, Numero),
       foreign key    (Libreta) references Libretas(Codigo),
       foreign key    (CodOp) references Operaciones(Codigo)
);
create index FechaApunte on Apuntes(Fecha);
```

Debemos distinguir entre el código de un apunte (*Codigo*) y su número (*Numero*). El primero servirá como clave primaria de la tabla de apuntes, mientras que el segundo será un valor consecutivo dentro de cada libreta, pero que puede repetirse a nivel global. En realidad, hubiéramos podido designar al par libreta y número como la clave primaria de *Apuntes*, pero hubiera sido más complicado, fundamentalmente porque los números de apuntes van a ser asignados en el servidor, y el lector ya conoce las implicaciones negativas que tiene esta decisión desde el punto de vista de las herramientas clientes.

Creación de vistas

Otra forma en que la programación en el servidor puede ahorrarnos trabajo es definiendo vistas para la presentación de datos. En esta aplicación puede sernos de suma utilidad poder resumir las operaciones por fechas, y desglosarlas en gastos e ingresos. Recuerde que nuestros apuntes indican un código de operación con más detalles que solamente conocer si son gastos e ingresos.

Estas son las dos vistas que crearemos en la base de datos:

```
create view Ingresos(Tipo, Fecha, Total) as
       select  'Ingresos', Fecha, sum(Importe)
       from    Apuntes
       where   CodOp in
               (select Codigo
                from   Operaciones
                where  Ingreso in ('S', 's'))
       group   by Fecha;

create view Gastos(Tipo, Fecha, Total) as
       select  'Gastos  ', Fecha, sum(Importe)
       from    Apuntes
       where   CodOp in
               (select Codigo
                from   Operaciones
                where  Ingreso in ('N', 'n'))
       group   by Fecha;
```

¿Se ha dado cuenta de los dos espacios al final de la cadena *'Gastos'*? Es que tenemos en mente combinar ambas vistas con el operador **union**. Y este operador exige que los tipos de datos de las dos relaciones que se unen sean idénticos, incluyendo el tamaño máximo de las cadenas de caracteres.

Desafortunadamente, InterBase no permite utilizar el operador **union** para concatenar directamente las dos vistas anteriores. Pero si necesitamos esta unión siempre podremos lanzar un cursor desde un cliente que realice dicha operación.

Manteniendo actualizados los saldos

Volvemos a enfrentarnos al problema de la asignación automática de códigos o claves artificiales. Dejaremos que los códigos de libretas y operaciones sean asignados por el usuario, pero asignaremos por programas la clave única de los apuntes: el campo *Codigo*. Esta vez, utilizaremos un generador para asignar un código único y global a todos los apuntes. Eso sí, mantendremos nuestra costumbre de pedir el código al servidor mediante un procedimiento almacenado y asignarlo en el cliente antes de enviar el registro para ser insertado:

```
create generator CodigoApunte;

create procedure ObtenerCodigo returns (Codigo int) as
begin
        Codigo = gen_id(CodigoApunte, 1);
end
```

Antes de insertar un apunte realizaremos varias operaciones. Para mantener legible el código, he aprovechado que InterBase permite definir varios *triggers* para una misma operación y utilizar la cláusula **position** para indicar el orden de disparo. El primer *trigger* verifica que el signo del importe coincida con el tipo de operación, ingreso o reintegro:

```
create trigger TipoOperacion for Apuntes
        active before insert position 1 as
declare variable TipoOp char;
begin
        select Ingreso
        from   Operaciones
        where  Codigo = new.CodOp
        into   :TipoOp;
        if (:TipoOp in ('S', 's')) then
        begin
                if (new.Importe < 0) then
                        new.Importe = - new.Importe;
        end
        else if (new.Importe > 0) then
                new.Importe = - new.Importe;
end
```

El siguiente *trigger* se dispara a continuación, actualizando los datos de la libreta y corrigiendo el saldo posterior de la operación y el número consecutivo de apunte:

```
create trigger MantenerSaldo for Apuntes
        active before insert position 2 as
begin
        /* Buscar el número de apunte y el saldo acumulado */
        select Saldo, NumApuntes
        from   Libretas
        where  Codigo = new.Libreta
        into   new.Saldo, new.Numero;
        new.Saldo = new.Saldo + new.Importe;
        new.Numero = new.Numero + 1;
        /* Actualizar la libreta */
        update Libretas
        set    Saldo = new.Saldo, NumApuntes = new.Numero
        where  Codigo = new.Libreta;
end
```

Otra operación que puede ser interesante es la que corrige el importe de un apunte. En la vida real, cuando detrás de la ventanilla anotan mal un apunte, tienen que realizar otro apunte a modo de compensación. Pero nosotros podemos ser más flexibles y ofrecer esta opción a los usuarios. Los apuntes compensatorios sólo sirven para introducir conceptos de operación artificiales, y para desfigurar las estadísticas.

En principio, pudiéramos expresar mediante *triggers* de modificación la regla de propagación a través de los saldos. Pero esta implementación sería bastante engorrosa, pues tendríamos que cuidarnos de las llamadas recursivas al *trigger*. Por lo tanto, crearemos un procedimiento almacenado para encapsular la operación:

```
create procedure ModificarApunte(Clave int, Imp int) as
declare variable Lib int;
declare variable Num int;
declare variable ImpAnt int;
begin
       select Libreta, Numero, Importe
       from   Apuntes
       where  Clave = :Clave
       into   :Lib, :Num, :ImpAnt;
       update Apuntes
       set    Importe = :Imp
       where  Clave = :Clave;
       update Apuntes
       set    Saldo = Saldo + :Imp - :ImpAnt
       where  Libreta = :Lib
         and  Numero >= :Num;
       update Libretas
       set    Saldo = Saldo + :Imp - :ImpAnt
       where  Codigo = :Lib;
end
```

Libretas de banco y MS SQL Server

El siguiente listado contiene la definición de tablas, índices, procedimientos y *triggers* para Transact-SQL. Por comodidad, al principio del listado se utiliza una base de datos *MastSql*, que puede crearse siguiendo las instrucciones del capítulo 24.

```
use MastSQL
go

// Creación de tablas e índices

create table Operaciones (
       Codigo         int not null,
       Descripcion    varchar(30) not null,
       Ingreso        char not null default 'S',

       primary key    (Codigo),
       unique         (Descripcion),
       check          (Ingreso in ('S', 's', 'N', 'n'))
)
go

create table Libretas (
       Codigo         int not null,
       Titular        varchar(30) not null,
       Descripcion    varchar(30) not null,
       Saldo          int not null default 0,
```

```
        NumApuntes      int not null default 0,

        primary key     (Codigo),
        unique          (Titular, Descripcion)
)
go

create table Apuntes (
        Codigo          int not null,
        Libreta         int not null,
        Numero          int not null default -1,
        Fecha           datetime not null default (getdate()),
        CodOp           int not null,
        Importe         int not null,
        Saldo           int null default null,

        primary key     (Codigo),
        unique          (Libreta, Numero),
        foreign key     (Libreta) references Libretas(Codigo),
        foreign key     (CodOp) references Operaciones(Codigo)
)

create index FechaApunte on Apuntes(Fecha)
go

// Códigos automáticos para la tabla de apuntes

create table Codigos (
        Codigo          int not null
)
insert into Codigos values (1)
go

create procedure ObtenerCodigo @cod int output as
begin
        select  @cod = Codigo
        from    Codigos holdlock
        update  Codigos
        set     Codigo = Codigo + 1
end
go

// Mantenimiento del saldo

create trigger MantenerSaldo on Apuntes for insert as
begin
        declare @Libreta int, @Importe int,
                @UltSaldo int, @UltApunte int, @TipoOp char
        select  @Libreta = Libreta, @Importe = Importe
        from    inserted
        select  @TipoOp = Ingreso
        from    Operaciones
        where   Codigo = (select CodOp from inserted)
        if (@TipoOp in ('S', 's') and @Importe < 0 or
                @TipoOp in ('N', 'n') and @Importe > 0)
                select @Importe = - @Importe
        select  @UltSaldo = Saldo, @UltApunte = NumApuntes
        from    Libretas holdlock
        where   Codigo = @Libreta
        update  Libretas
```

```
            set     Saldo = Saldo+@Importe, NumApuntes = NumApuntes+1
            where   Codigo = @Libreta
            update  Apuntes
            set     Numero = @UltApunte + 1, Importe = @Importe,
                  Saldo = @UltSaldo + @Importe
            where   Codigo = (select Codigo from inserted)
    end
    go
```

Como es de suponer, existen diferencias importantes. El *trigger* para la actualización de los saldos se ejecuta después de realizada la grabación. Por este motivo tendremos que suministrar un valor por omisión para el número del apunte ... o renunciar a la unicidad de la combinación libreta/número.

Ahora, en Oracle

El siguiente listado contiene la definición de los objetos de la base de datos, esta vez en el dialecto PL/SQL de Oracle:

```
connect system/manager;

-- Definición de las tablas -------------------------------------

-- Conceptos
create table Conceptos (
       Codigo          int not null,
       Descripcion     varchar(30) not null,
       Ingreso         char(1),
       primary key     (Codigo),
       unique          (Descripcion),
       check           (Ingreso in ('S', 's', 'N', 'n'))
);

-- Libretas
create table Libretas (
       Codigo          int not null,
       Titular         varchar(30) not null,
       Descripcion     varchar(50) not null,
       Saldo           int default 0,
       NumApuntes      int default 0,
       primary key     (Codigo)
);
create index TitularLibreta on Libretas(Titular);

-- Apuntes
create table Apuntes (
       Clave           int not null,
       Libreta         int not null,
       Numero          int not null,
       Fecha           date default sysdate not null,
       CodConcepto     int not null,
       Importe         int not null,
```

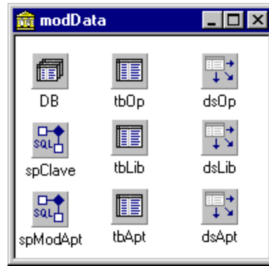
```
        Saldo          int,
        primary key    (Clave),
        unique         (Libreta, Numero),
        foreign key    (Libreta) references Libretas(Codigo),
        foreign key    (CodConcepto) references Conceptos(Codigo)
);
create index FechaApunte on Apuntes(Fecha);

-- Vistas

create view Ingresos(Tipo, Fecha, Total) as
        select "Ingresos", Fecha, sum(Importe)
        from   Apuntes
        where  CodConcepto in
               (select Codigo
                from   Conceptos
                where  Ingreso in ('S', 's'))
        group  by Fecha;

create view Gastos(Tipo, Fecha, Total) as
        select "Gastos  ", Fecha, sum(Importe)
        from   Apuntes
        where  CodConcepto in
               (select Codigo
                from   Conceptos
                where  Ingreso in ('N', 'n'))
        group   by Fecha;

-- Claves secuenciales
create table Claves (
        Proxima        integer
);

insert into Claves values(1);

-- Triggers y procedimientos almacenados -----------------------

-- Tabla de apuntes

create or replace procedure ObtenerClave (Clave out int) as
begin
        select Proxima
        into   Clave
        from   Claves;
        update Claves
        set    Proxima = Proxima + 1;
end;
/

create or replace procedure ModificarApunte(unaClave int,
   unImp int) as
        Lib int;
        Num int;
        ImpAnt int;
begin
        select Libreta, Numero, Importe
        into   Lib, Num, ImpAnt
        from   Apuntes
        where  Clave = unaClave;
        update Apuntes
```

```
        set     Importe = unImp
        where   Clave = unaClave;
        update  Apuntes
        set     Saldo = Saldo + unImp - ImpAnt
        where   Libreta = Lib
          and   Numero >= Num;
        update  Libretas
        set     Saldo = Saldo + unImp - ImpAnt
        where   Codigo = Lib;
end;
/

create or replace trigger NuevoApunte
        before insert on Apuntes
        for each row
declare
        TipoOp char;
begin
        -- Corregir importe de acuerdo al tipo de operación
        select  Ingreso
        into    TipoOp
        from    Conceptos
        where   Codigo = :new.CodConcepto;
        if (TipoOp in ('S', 's')) then
                if (:new.Importe < 0) then
                        :new.Importe := - :new.Importe;
                end if;
        else
                if (:new.Importe > 0) then
                        :new.Importe := - :new.Importe;
                end if;
        end if;
        -- Buscar el número de apunte y el saldo acumulado
        select  Saldo, NumApuntes
        into    :new.Saldo, :new.Numero
        from    Libretas
        where   Codigo = :new.Libreta;
        :new.Saldo := :new.Saldo + :new.Importe;
        :new.Numero := :new.Numero + 1;
        -- Actualizar la tabla de libretas
        update  Libretas
        set     Saldo = :new.Saldo, NumApuntes = :new.Numero
        where   Codigo = :new.Libreta;
end;
/

-- FIN ---------------------------------------------------------
```

El módulo de datos

El desarrollo de la aplicación comenzará por la implementación del módulo de datos de la misma. El primer objeto será, por supuesto, un componente de tipo *TDatabase*, con el cual accederemos a la base de datos. La configuración del componente de-

penderá en lo fundamental del formato que se esté utilizando para los datos: InterBase, Oracle o MS SQL Server.



Una vez que tenemos la referencia a la base de datos traemos tres tablas al módulo: *tbOp* (operaciones), *tbLib* (libretas) y *tbApt* (apuntes), y sus correspondientes fuentes de datos. Establecemos una relación *master/detail* desde la tabla de libretas hacia la tabla de apuntes:

Propiedad	Valor
Name	*tbApt*
DatabaseName	*APT*
TableName	*Apuntes*
MasterSource	*dsLib*
MasterFields	*Codigo*
IndexFieldNames	*Libreta;Numero*

Preste atención a la propiedad *IndexFieldNames*, pues se han concatenado los campos *Libreta* y *Numero*, para que la tabla de detalles quede ordenada efectivamente por el número de los apuntes.

Definimos también, en la tabla de apuntes, un campo de referencia a la tabla de operaciones. Como el modelo de datos de la aplicación es sencillo y pequeño, no utilizaremos una tabla separada para las referencias, que es lo que recomiendo en otros casos. En la tabla de apuntes, por su parte, definiremos un campo calculado *Operación*, para traducir el "sí" y el "no" del campo *Ingreso* en algo más legible:

```
procedure TmodData.tbOpCalcFields(DataSet: TDataSet);
begin
   if not tbOpIngreso.IsNull then
      if UpperCase(tbOpIngreso.Value) = 'S' then
         tbOpOperacion.Value := 'Ingreso'
      else
         tbOpOperacion.Value := 'Reintegro';
end;
```

Traemos un par de componentes *TStoredProc* al módulo, *spCodigo* y *spModApt*, y hacemos que se refieran a los procedimientos almacenados *ObtenerCodigo* y *Modificar-Apunte*. El primer procedimiento almacenado será utilizado para asignar la clave primaria de un apunte antes de grabarlo, en el evento *BeforePost*:

```
procedure TmodData.tbAptBeforePost(DataSet: TDataSet);
begin
   spClave.ExecProc;
   tbAptClave.Value := spClave.Params[0].AsInteger;
end;
```

Note que no se pregunta si la operación *Post* se dispara para una inserción o modificación, pues en esta aplicación no se modificarán apuntes directamente, sino por medio de procedimientos almacenados.

Los últimos tres eventos que interceptaremos están relacionados con la inicialización del campo fecha de los apuntes durante la inserción. Es muy conveniente que, cuando estemos tecleando apuntes, se mantenga para la fecha del nuevo apunte la fecha del último apunte insertado. Esto lo conseguimos declarando una variable *Last-Date* en el módulo de datos:

```
type
   TmodData = class(TDataModule)
      // …
   private
      LastDate: Variant;
      // …
   end;
```

La variable ha sido declarada de tipo *Variant* con el propósito de que pueda almacenar el valor *Null*. Estos son, finalmente, los eventos que utilizan a esta variable:

```
procedure TmodData.modDataCreate(Sender: TObject);
begin
   LastDate := Null;
end;

procedure TmodData.tbAptNewRecord(DataSet: TDataSet);
begin
   if VarIsNull(LastDate) then
      tbAptFecha.Value := Date
   else
      tbAptFecha.Value := LastDate;
end;

procedure TmodData.tbAptAfterPost(DataSet: TDataSet);
begin
   LastDate := tbAptFecha.Value;
end;
```

Gestión de libretas

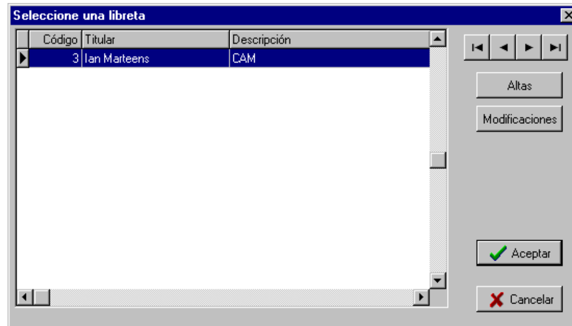
Pasamos ahora al desarrollo visual de la aplicación. Esta vez no utilizaremos el modelo MDI para la organización de las ventanas. La ventana principal será una ventana SDI, que mostrará los apuntes correspondientes a determinada libreta; no olvide que la tabla de apuntes está en relación *master/detail* con la de libretas. Antes de mostrar la ventana principal de la aplicación necesitaremos un cuadro de diálogo que nos permita gestionar libretas (crear o modificar) y seleccionar una de ellas antes de seguir trabajando. Por lo tanto, comenzaremos con la creación de estos formularios.

El primero será un cuadro de diálogo, al cual bautizaremos como *dlgLibretas*. En este formulario será donde teclearemos los datos de una nueva libreta y permitiremos la modificación de determinados datos de una libreta existente. Traemos cuadros de edición para cada uno de los campos de la tabla *tbLib*, y nos cuidamos bien de desactivar los correspondientes al saldo y al número de apuntes, pues los valores de estos campos deberán mantenerse automáticamente.

El único código asociado al diálogo serán las respuestas a los métodos *OnCreate*, para desactivar la modificación del código durante las ediciones, y *OnCloseQuery*:

```
procedure TdlgLibretas.FormCreate(Sender: TObject);
begin
   if modData.tbLib.State = dsEdit then
   begin
      DBEdit1.Enabled := False;
      DBEdit1.TabStop := False;
      DBEdit1.ParentColor := True;
      DBEdit1.ReadOnly := True;
   end;
end;

procedure TdlgLibretas.FormCloseQuery(Sender: TObject;
   var CanClose: Boolean);
begin
   if ModalResult = mrOk then
      modData.tbLib.Post
   else
      modData.tbLib.Cancel;
end;
```

Creamos ahora otro cuadro de diálogo, *TwndLibretas*, que servirá para la selección de una libreta, y que podrá ejecutar a *TdlgLibretas* para crear y modificar libretas:



La rejilla ha sido configurada como sólo para lectura, y se han activado las opciones *dgRowSelect* y *dgAlwaysShowSelection*. Aunque no se muestran en la imagen, la rejilla contiene también columnas para el saldo y el número de apuntes. La ejecución del cuadro de altas y modificaciones se realiza en el método *Mantenimiento*, que es la respuesta compartida por los botones de *Altas* y *Modificaciones*:

```
procedure TwndLibretas.Mantenimiento(Sender: TObject);
begin
   with modData.tbLib do
      if Sender = bnAltas then Append else Edit;
   with TdlgLibretas.Create(Application) do
   try
      ShowModal;
   finally
      Free;
   end;
end;
```

Al presentarse el diálogo en pantalla, es necesario actualizar sus datos, pues el número de apuntes y el saldo acumulado se actualizan en el servidor, sin que la tabla de libretas en el cliente tenga noticia de estos cambios:

```
procedure TwndLibretas.FormShow(Sender: TObject);
begin
   modData.tbLib.Refresh;
end;
```

Una doble pulsación sobre la rejilla debe tener el mismo efecto que una pulsación sobre el botón *Aceptar*:

```
procedure TwndLibretas.DBGrid1DblClick(Sender: TObject);
begin
   bnOk.Click;
end;
```

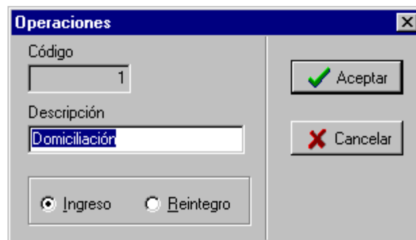
Finalmente, el siguiente método de clase encapsula la creación y visualización del cuadro de diálogo:

```
class function TwndLibretas.Select: Boolean;
var
   BM: TBookmarkStr;
begin
   with Create(Application) do
   try
      BM := modData.tbLib.Bookmark;
      Result := ShowModal = mrOk;
      if not Result then
         modData.tbLib.Bookmark := BM;
   finally
      Free;
   end;
end;
```

Observe el uso de la marca para regresar a la libreta previamente seleccionada cuando se cancela el cuadro de diálogo.

Códigos de operaciones

Muy similares serán los formularios de gestión de la tabla de operaciones. El primero en crearse será el diálogo de altas y modificaciones, *TdlgOper*:

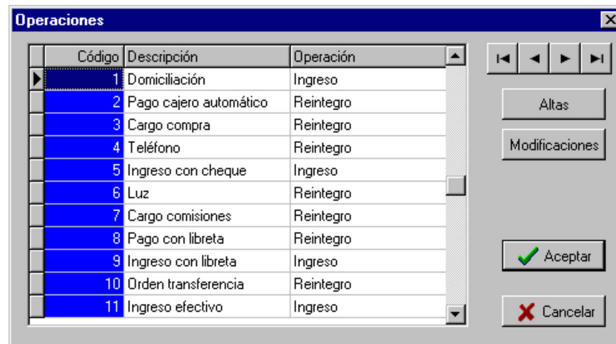


Al igual que en el diálogo de libretas, interceptamos los eventos *OnCreate* y *OnCloseQuery*:

```
procedure TdlgOper.FormCreate(Sender: TObject);
begin
   if modData.tbOp.State = dsEdit then
   begin
      DBEdit1.ReadOnly := True;
      DBEdit1.Enabled := False;
      DBEdit1.TabStop := False;
      DBEdit1.ParentColor := True;
   end;
end;

procedure TdlgOper.FormCloseQuery(Sender: TObject;
   var CanClose: Boolean);
```

```
var
   Ins: Boolean;
begin
   if ModalResult = mrOk then
   begin
      Ins := modData.tbOp.State = dsInsert;
      modData.tbOp.Post;
      if Ins then
      begin
         modData.tbOp.Append;
         SelectFirst;
         CanClose := False;
      end;
   end
   else
      modData.tbOp.Cancel;
end;
```

La siguiente imagen muestra la ventana de visualización de operaciones, a la cual llamaremos *TwndOper*:



Este es el código asociado al formulario:

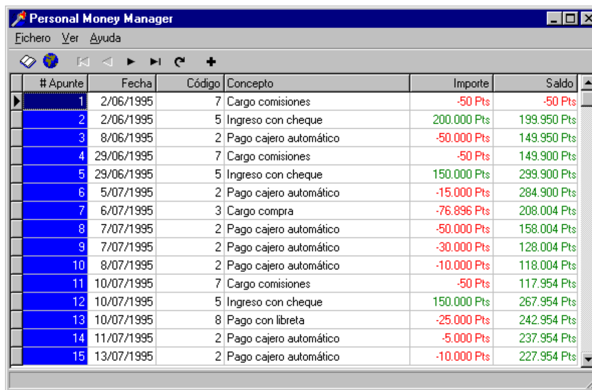
```
procedure TwndOper.Modificaciones(Sender: TObject);
begin
   if Sender = bnAltas then
      modData.tbOp.Append
   else
      modData.tbOp.Edit;
   with TdlgOper.Create(Application) do
   try
      ShowModal;
   finally
      Free;
   end;
end;

procedure TwndOper.DBGrid1DblClick(Sender: TObject);
begin
   bnOk.Click;
end;
```

```
class function TwndOper.Select: Boolean;
var
   BM: TBookmarkStr;
begin
   with Create(Application) do
   try
      BM := modData.tbOp.Bookmark;
      Result := ShowModal = mrOk;
      if not Result then
         modData.tbOp.Bookmark := BM;
   finally
      Free;
   end;
end;
```

Como puede observar, el método *Select* es casi idéntico al del diálogo de libretas. Le propongo al lector que piense en cómo aprovechar la herencia visual para diseñar un prototipo común para *wndOper* y *wndLib*.

La ventana principal

Ya podemos plantearnos la creación de la ventana principal. El aspecto final de la misma se muestra en la siguiente imagen:



# Apunte	Fecha	Código	Concepto	Importe	Saldo
1	2/06/1995	7	Cargo comisiones	-50 Pts	-50 Pts
2	2/06/1995	5	Ingreso con cheque	200.000 Pts	199.950 Pts
3	8/06/1995	2	Pago cajero automático	-50.000 Pts	149.950 Pts
4	29/06/1995	7	Cargo comisiones	-50 Pts	149.900 Pts
5	29/06/1995	5	Ingreso con cheque	150.000 Pts	299.900 Pts
6	5/07/1995	2	Pago cajero automático	-15.000 Pts	284.900 Pts
7	6/07/1995	3	Cargo compra	-76.896 Pts	208.004 Pts
8	7/07/1995	2	Pago cajero automático	-50.000 Pts	158.004 Pts
9	7/07/1995	2	Pago cajero automático	-30.000 Pts	128.004 Pts
10	8/07/1995	2	Pago cajero automático	-10.000 Pts	118.004 Pts
11	10/07/1995	7	Cargo comisiones	-50 Pts	117.954 Pts
12	10/07/1995	5	Ingreso con cheque	150.000 Pts	267.954 Pts
13	10/07/1995	8	Pago con libreta	-25.000 Pts	242.954 Pts
14	11/07/1995	2	Pago cajero automático	-5.000 Pts	237.954 Pts
15	13/07/1995	2	Pago cajero automático	-10.000 Pts	227.954 Pts

El primer paso es mostrar la ventana de gestión de libretas antes de mostrar la ventana principal en la pantalla:

```
procedure TwndMain.FormShow(Sender: TObject);
begin
   if not TwndLibretas.Select then
      Application.Terminate;
end;
```

El menú de la ventana será el siguiente:

Fichero	**V**er	**A**yuda
Libretas...	**S**aldo	**A**cerca de...
Operaciones...	**A**nálisis	
Modificar importe		
Configurar impresora...		
Imprimir		
Salir		

Los comandos *Libretas* y *Operaciones* serán compartidos por un par de botones en la barra de herramientas que situaremos en la ventana. Las respuestas son extremadamente simples:

```
procedure TwndMain.bnLibretasClick(Sender: TObject);
begin
   TwndLibretas.Select;
end;

procedure TwndMain.bnOperacionesClick(Sender: TObject);
begin
   TwndOper.Select;
end;
```

El siguiente método, que se ejecuta en respuesta al evento *OnDrawColumnCell* de la rejilla, se ocupa de mostrar en verde o en rojo los valores del saldo y del importe de cada apunte, según el valor de la columna sea positivo o negativo:

```
procedure TwndMain.DBGrid1DrawColumnCell(Sender: TObject;
   const Rect: TRect; DataCol: Integer; Column: TColumn;
   State: TGridDrawState);
begin
   with Sender as TDBGrid, Canvas.Font do
   begin
      if CompareText(Column.FieldName, 'Saldo') = 0 then
         if modData.tbAptSaldo.Value > 0 then
            Color := clGreen
         else
            Color := clRed
      else if CompareText(Column.FieldName, 'Importe') = 0 then
         if modData.tbAptImporte.Value > 0 then
            Color := clGreen
         else
            Color := clRed;
      DefaultDrawColumnCell(Rect, DataCol, Column, State);
   end;
end;
```

En la siguiente sección crearemos un cuadro de diálogo para la entrada de los apuntes, que denominaremos *TdlgApt*. Este cuadro de diálogo debe ejecutarse en respuesta a la pulsación del botón de inserción del navegador:

```
procedure TwndMain.DBNavigator2Click(Sender: TObject;
   Button: TNavigateBtn);
```

```
begin
   if Button = nbInsert then
      with TdlgApuntes.Create(Application) do
      try
        ShowModal;
      finally
         Free;
      end;
end;
```

Y una doble pulsación sobre la rejilla debe tener el mismo efecto:

```
procedure TwndMain.DBGrid1DblClick(Sender: TObject);
begin
   DBNavigator2.BtnClick(nbInsert);
end;
```

Entrada de apuntes

Si queremos que alguien utilice nuestra maravillosa aplicación, tendremos que esmerarnos con el cuadro de entrada de apuntes; lo sé por experiencia personal. Debemos hacer que la introducción de datos sea lo más cómoda posible, teniendo en cuenta que es la parte de la aplicación que más se utilizará.

Este es el aspecto del cuadro de diálogo:

La grabación y cancelación de cambios se realiza en respuesta a los eventos *OnClick* del botón de grabación y *OnCloseQuery*, y es una variación del esquema que hemos estado utilizando a lo largo del libro:

```
procedure TdlgApuntes.FormCloseQuery(Sender: TObject;
   var CanClose: Boolean);
begin
   modData.tbApt.Cancel;
end;

procedure TdlgApuntes.bnSaveClick(Sender: TObject);
begin
   modData.tbApt.Post;
   modData.tbApt.Append;
   SelectFirst;
end;
```

Para facilitar la entrada de datos numéricos, transformaremos la acción de la tecla INTRO, como hicimos en la aplicación de entrada de pedidos. Primero, asignamos *True* a la propiedad *KeyPreview*, para después interceptar el evento *OnKeyPress* del formulario:

```
procedure TdlgApuntes.FormKeyPress(Sender: TObject;
   var Key: Char);
begin
   if (Key = #13) and not (ActiveControl is TButton) then
   begin
      SelectNext(ActiveControl, True, True);
      Key := #0;
   end;
end;
```

También ayudaremos al usuario cuando se encuentre sobre el cuadro de edición de fechas, *DBEdit1* en nuestro formulario. Queremos que las teclas + y – sirvan para pasar al día siguiente y al día anterior, respectivamente. La clave es manejar el evento *OnKeyPress*, esta vez del control *DBEdit1*:

```
procedure TdlgApuntes.DBEdit1KeyPress(Sender: TObject;
   var Key: Char);
begin
   with modData.tbAptFecha do
      case Key of
         '+': Value := Value + 1;
         '-': Value := Value - 1;
      else
         Exit;
      end;
   Key := #0;
end;
```

Como habrá observado el lector, el formulario tiene un par de casillas de verificación, a las cuales hemos bautizado *cbMillares* y *cbCodigo*. Cuando la primera esté activa, los valores menores de 1000 pesetas que se tecleen en el cuadro de edición *DBEdit3* (el importe), se multiplican automáticamente por 1000:

```
procedure TdlgApuntes.DBEdit3Exit(Sender: TObject);
begin
   with modData.tbAptImporte do
      if not IsNull and cbMillares.Checked
         and (Abs(Value) < 1000) then
         Value := Value * 1000;
end;
```

Por último, el control *cbCodigo* se utiliza para activar o desactivar la edición de la operación por su código. Para determinados usuarios, puede ser más cómodo teclear el código de la operación que realizar una búsqueda incremental en el combo de búsqueda. Esta es la respuesta al evento *OnClick* del control *cbCodigo*:

```
procedure TdlgApuntes.cbCodigoClick(Sender: TObject);
```

```
begin
   with DBEdit2 do
   begin
      TabStop := cbCodigo.Checked;
      Enabled := cbCodigo.Checked;
      ReadOnly := not cbCodigo.Checked;
      if ReadOnly then
         ParentColor := True
      else
         Color := clWindow;
   end;
end;
```

El lector puede añadir sus propias mejoras al formulario.

Corrigiendo el importe de un apunte

Sin embargo, y a pesar de todas las facilidades que podamos incluir para la entrada de apuntes, es normal que el usuario pueda equivocarse tecleando un apunte, sobre todo en lo que respecta al importe. Yo, por ejemplo, tiendo a pagar menos y a cobrar más. En circunstancias reales, cuando el banco se equivoca se introducen apuntes compensatorios. Esta solución no es la mejor para los objetivos de la aplicación, pues este tipo de apuntes complica la obtención de estadísticas. Por fortuna, esta situación la habíamos previsto, cuando creamos el procedimiento almacenado *ModificarApunte* en la base de datos.

Necesitamos un nuevo cuadro de diálogo, al cual le daremos el nombre de *dlgModApt*. La siguiente imagen muestra el aspecto visual del mismo:

Los cuadros de edición que aparecen desactivados en la parte superior del diálogo, son controles *DBEdit* enlazados a los campos de la tabla *tbApt*, y sirven de referencia para la modificación del apunte. En la creación del formulario, se inicializa el texto del control *Edit1* (el nuevo importe) con el valor actual del importe del apunte a modificar:

```
procedure TdlgModApt.FormCreate(Sender: TObject);
begin
   Edit1.Text := modData.tbAptImporte.AsString;
end;
```

Cuando se vaya a cerrar el cuadro de diálogo, se ejecuta el procedimiento almacenado *spModApt*, que hemos incluido en el módulo de datos:

```
procedure TdlgModApt.FormCloseQuery(Sender: TObject;
   var CanClose: Boolean);
begin
   if ModalResult <> mrOk then Exit;
   // Verificar que coincidan los signos
   if (StrToInt(Edit1.Text) > 0) <>
      (modData.tbAptImporte.Value > 0) then
      DatabaseError('El signo del importe es incorrecto');
   // Ejecutar el procedimiento almacenado
   with modData.spModApt do
   begin
      ParamByName('CLAVE').AsInteger := modData.tbAptClave.Value;
      ParamByName('IMP').AsInteger := StrToInt(Edit1.Text);
      ExecProc;
      modData.tbApt.Refresh;
   end;
end;
```

Este cuadro de diálogo se lanza desde la ventana principal, en respuesta al comando *Modificar importe* del menú *Fichero*:

```
procedure TwndMain.ModificarImporte1Click(Sender: TObject);
begin
   with TdlgModApt.Create(Application) do
   try
      ShowModal;
   finally
      Free;
   end;
end;
```

Como detalle final, el comando anteriormente mencionado se activa o desactiva en la respuesta al evento *OnClick* del elemento de menú *Fichero1*:

```
procedure TwndMain.Fichero1Click(Sender: TObject);
begin
   ModificarImporte1.Enabled := not modData.tbApt.IsEmpty;
end;
```

Y es que no tiene sentido modificar algo que no se tiene.

> En el capítulo 35 volveremos a ver este ejemplo, que utilizaremos para la creación de gráficos y estadísticas.

6

Los detalles finales

- Impresión de informes con QuickReport
- Gráficos de decisión
- Bibliotecas de Enlace Dinámico
- Servidores de Internet
- Conjuntos de datos clientes
- El Modelo de Objetos Componentes: la teoría
- El Modelo de Objetos Componentes: ejemplos
- Bases de datos remotas
- Creación de instalaciones

Parte

Capítulo

34

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. Era un generador de informes bastante bueno, con posibilidades de impresión de informes por columnas, *master/detail*, tabulares, gráficos, etc. Sin embargo, no tuvo la acogida deseable por parte de los programadores. ¿La razón?: un *runtime* bastante grande e incómodo que distribuir, demasiado ineficiente en tiempo y espacio (estamos hablando de los tiempos de Delphi 1, sobre 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. Posteriormente se descubrió que era muy fácil ocultar esta pantalla, pero estoy seguro de que ésta fue una de las causas no confesadas que motivaron la aparición de toda una variedad de sistemas de informes alternativos.

La historia del producto

En junio de 1995, un programador noruego llamado Allan Lochert colocó en la Internet un pequeño y eficiente sistema de impresión de informes de libre distribución, al que bautizó QuickReport. Era un producto simple y elegante, basado en el principio "lo pequeño es bello"; el código fuente solamente ocupaba 5000 líneas. El sistema fue creciendo y depurándose, llamando la atención de Borland, que lo incluyó como alternativa a ReportSmith en Delphi 2. Cuando apareció la versión 2.01 de Delphi, la que introdujo los componentes para Internet, el QuickReport que se incluyó era una versión con ligeras mejoras al producto original: configuración del tamaño de papel, marcos para bandas, etc. Adicionalmente, era posible comprar aparte la versión profesional de QuickReport, que contenía el código fuente y los componentes para 16 y 32 bits.

Delphi 3 y QuickReport 2 aparecieron a la par. El nuevo QuickReport había sido reprogramado de arriba abajo, cambiando totalmente su apariencia y añadiéndose más potencia. La filosofía del producto seguía siendo la misma, y era fácil, para alguien familiarizado con la versión anterior, crear un informe con la nueva. También era posible convertir de forma automática un informe de la versión 1 a la 2, pero en ciertos casos especiales, la conversión debía complementarse manualmente.

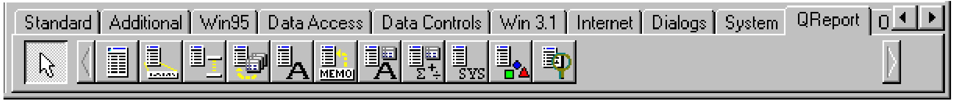
Por desgracia, junto a la inclusión de nuevas características aparecieron visitantes no deseados: me refiero a *bugs*. Durante el año y poco que duró Delphi 3, QuSoft (la empresa responsable de QuickReport) sacó parches designados alfabéticamente, desde la A hasta la K. A veces un parche no sólo corregía errores, sino que introducía otros nuevos. Para colmo de males, ReportSmith se dejó de vender con Delphi. Borland pasó el producto a la compañía Strategic Reporting, aunque continuó incluyendo en la Paleta de Componentes el componente *TReport*, que permite establecer una conexión con el motor de impresión de ReportSmith. Este componente está inicialmente escondido, y hay que utilizar el diálogo de propiedades de la Paleta para mostrarlo.

Ahora con Delphi 4, tenemos la versión 3 del QuickReport. Solamente el tiempo y la experiencia nos dirá si es una versión estable y fiable. Esperemos que sí (presunción de inocencia).

En este capítulo estudiaremos solamente las versiones 2 y 3, pues la versión 1 ha quedado obsoleta. Cuando tenga que referirme a alguna versión en específico, utilizaré las siglas QR1, QR2 y QR3 por brevedad. He aquí la Paleta de Componentes de QR3:

La página de QR2 en la Paleta tenía este aspecto:

Aunque estoy mostrando las Paleta de Delphi 3 y 4, recuerde que este producto puede instalarse también en Delphi 1 y Delphi 2. Y para que pueda comparar, esta era la página original de QR1, con menos componentes:



La filosofía del producto

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 en Delphi, utilizando las propias herramientas de diseño de Delphi. Los componentes de QuickReport se colocan en un formulario de Delphi, 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, en Delphi 3 y 4 el código del motor puede utilizase desde un paquete; *qrpt30.dpl* ó *qrpt40.bpl*, según la versión.

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.

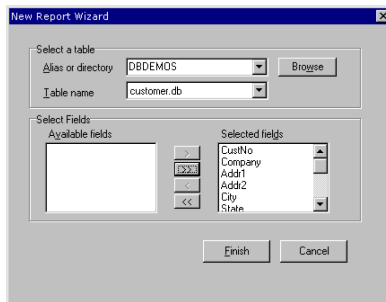
Claro está, también existen inconvenientes para este estilo de creación de informes. Ya hemos mencionado el primero: no hay una retroalimentación inmediata de las acciones de edición. El segundo inconveniente es más sutil. Con un sistema de informes independiente se pueden incluir *a posteriori* informes adicionales, que puede diseñar el propio usuario de la aplicación. Esto no puede realizarse directamente con QuickReport, a no ser que montemos un sofisticado mecanismo basado en DLLs, o

un formato de intercambio diseñado desde cero. En el capítulo 40 ofrecemos una alternativa interesante.

Plantillas y expertos para QuickReport

QuickReport trae plantillas para acelerar la creación de informes. Estas plantillas son tres: una para listados de una sola tabla, una para informes *master/detail*, y otra para la impresión de etiquetas. En Delphi 2, 3 y 4 estas plantillas se encuentran en la página *Forms* del Depósito de Objetos. Cuando se instala QuickReport en Delphi 1, hay que registrar manualmente las plantillas en la Galería. No voy a explicar directamente el trabajo con estas plantillas pues, a partir de la explicación que haré de los componentes de impresión, puede deducirse fácilmente qué se puede hacer con ellas.

Delphi nos ofrece un experto para la generación de listados simples. Este experto está en la página *Business* del Depósito. 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, ya sea mediante un alias o un directorio, si se trata de una base de datos local. Además, debemos elegir qué campos de la tabla seleccionada queremos mostrar:

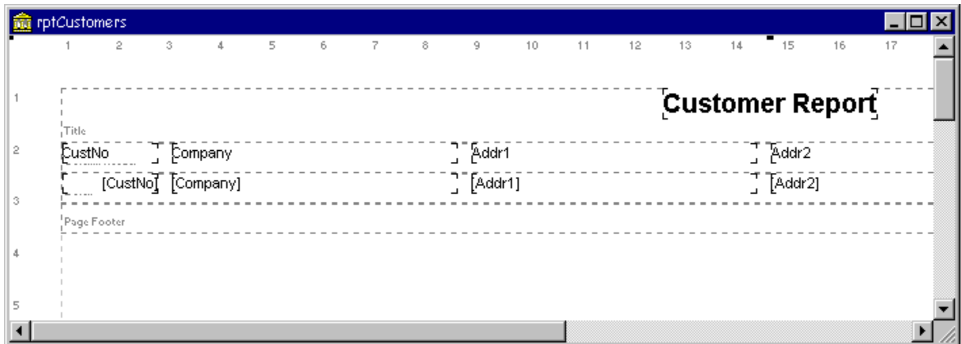


El resultado es un formulario con una tabla y con los componentes necesarios de QuickReport. Si quiere visualizar el informe generado, pulse el botón derecho del ratón sobre el componente *TQuickRep* y ejecute el comando *Preview*.

> El generador de informes que venía con Delphi 3 fallaba constantemente. La secuencia de pantallas era ligeramente diferente a la del experto de Delphi 4, y la tabla que creaba la dejaba cerrada, por lo que el comando *Preview* no funcionaba inmediatamente.

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;
```

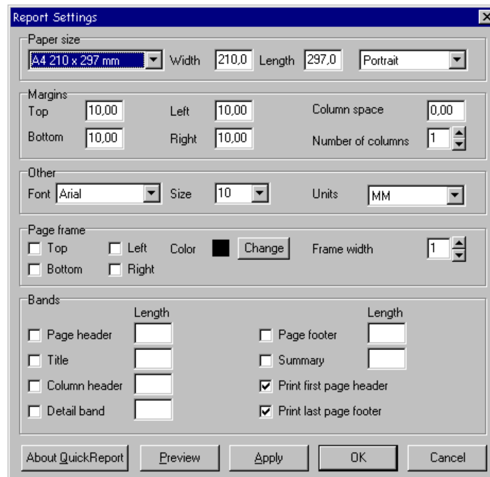
La propiedad *ReportTitle* del informe se utiliza como título de la ventana de previsualización predefinida.

> 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
PaperSize, Length, Width	Tamaño del papel
Orientation	Orientación de la página
LeftMargin, RightMargin	Ancho de los márgenes horizontales
TopMargin, BottomMargin	Ancho de los márgenes verticales
Ruler	Mostrar regla en tiempo de diseño
Columns	Número de columnas
ColumnSpace	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 componente. 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 los valores posibles de esta propiedad:

Tipo de banda	Objetivo
rbTitle	Se imprime una sola vez, al principio del informe
rbSummary	Se imprime una sola vez, al terminar el informe
rbPageHeader	Se imprime en cada página, en la cabecera
rbPageFooter	Se imprime en cada página, al final de la misma
rbColumnHeader	Si la página se divide en columnas, al comienzo de cada una
rbDetail	Se imprime para cada registro de la tabla principal
rbSubDetail	Se imprime para cada registro de una tabla dependiente
rbGroupHeader	Se imprime cuando se detecta un cambio de grupo
rbGroupFooter	Se imprime cuando termina la impresión de un grupo
rbChild	Banda hija: se imprime siempre después de su banda madre
rbOverlay	Se sobreimprime sobre cada página

El orden de impresión de los diferentes tipos de bandas, para un listado simple, es el siguiente:

rbPageHeader: En todas las páginas
rbTitle: En la primera página
rbColumnHeader: En cada columna, si las hay
rbDetail: Una banda por cada fila de la tabla
rbSummary: Al final del informe
rbPageFooter: 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 Componente, 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 la segunda línea de dirección está asignada. 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...
TQRLabel	Un texto fijo de una línea
TQRMemo	Un texto fijo con varias líneas
TQRRichEdit	Un texto fijo en formato RTF
TQRImage	Una imagen fija, en uno de los formatos de Delphi
TQRShape	Una figura geométrica simple
TQRSysData	Fecha actual, número de página, número de registro, etc.
TQRDBText	Un texto extraído de una columna
TQRDBRichEdit	Un texto RTF extraído de una columna
TQRExpression	Una expresión que puede referirse a columnas
TQRDBImage	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 horizontal definida.

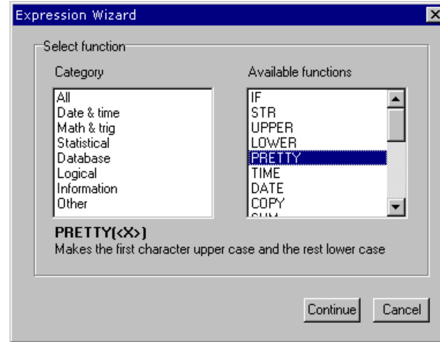
El componente más frecuentemente empleado es *TQRDBText*, que imprime el contenido de un campo de un conjunto de datos. Hay que configurar sus propiedades *DataSet* y *DataField*. Tiene la ventaja de que aprovecha automáticamente el formato de visualización del campo asociado, algo que no hace el componente 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 cabe utilizar un componente *TQrExpr*, configurando su propiedad *Expression* del siguiente modo:

```
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 sim-

ple, 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 estadística.

> 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.

Definiendo nuevas funciones

Una interesante posibilidad es la de poder extender el conjunto de funciones admitidas por el evaluador. Para esto basta con definir una clase derivada de la clase *TQREvElementFunction*, definida en la unidad *QRExpr*, y registrarla. Pueden definirse funciones simples y de conjuntos (al estilo de *Sum*, *Avg* y *Max*), con parámetros y sin parámetros.

Como ejemplo, definiremos una función que devuelva el año de una fecha como si fuera una cadena de caracteres. Aunque podemos imprimir el año de una fecha utilizando máscaras de impresión, esta función puede sernos útil para informes agrupados, que veremos en la siguiente sección, en los que el criterio de agrupamiento es precisamente por años. Creamos una unidad, y definimos la siguiente clase (puede ser en la sección de implementación):

```
uses QrExpr, SysUtils;

type
   TQREvYear = class(TQREvElementFunction)
   public
      function Calculate: TQREvResult; override;
   end;
```

Esta es la implementación de la función *Calculate*:

```
function TQREvYear.Calculate: TQREvResult;
begin
   if (ArgList.Count = 1) and (Argument(0).Kind = resString) then
   begin
      Result.Kind := resString;
      Result.StrResult := FormatDateTime('yyyy',
         TDateTime(StrToDate(Argument(0).StrResult)));
   end
   else
      Result.Kind := resError;
end;
```

Estamos comprobando el número de argumentos recibidos mediante la propiedad *ArgList*, y obteniendo el valor del mismo mediante la función *Argument*, que interpreta los valores almacenados en *ArgList*. El evaluador de expresiones de QuickReport pasa los valores de fecha como cadenas de caracteres. Para obtener el resultado con total seguridad e independencia del formato, convertimos primero la cadena a fecha, y después volvemos a darle formato, con la función *FormatDateTime*, una vieja conocida.

La clase creada debe registrarse en la sección de inicialización de la unidad:

```
initialization
   RegisterQRFunction(
      TQREvYear,                     // La clase de la función
      'YEAR',                        // El nombre de la función
      'YEAR(X)|Returns the year of a date as a string',
      'The Marteens'' Factory',      // El fabricante
      '1S');                         // Grupo y tipo de argumentos
end.
```

Si vamos a utilizar mucho esta función, quizás nos interese que aparezca en la lista de funciones en el Editor de Expresiones. Aunque este paso no es necesario, podemos crear un *package* de tiempo de diseño para que registre la función en el entorno de Delphi. De todos modos, basta con incluir la unidad en algún sitio de nuestro proyecto, y editar manualmente la propiedad *Expression* del componente en que la vamos a emplear.

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 un 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 ' + STR(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:

```
SUM(Total)
```

De este modo, cuando termine cada grupo se imprimirá el total de todas las compras realizadas por los clientes de ese grupo.

Informe con grupos

Close

Informe con grupos

Empresa		Total
Más de 250000		
Sight Diver		$261.575,80
	Total:	$261.575,80
Más de 175000		
VIP Divers Club		$193.079,75
American SCUBA Supply		$183.094,40
	Total:	$376.174,15
Más de 150000		
Blue Sports		$165.245,45
	Total:	$165.245,45

Page 1 of 2

> 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	**Freeport**
Unisco	
Blue Sports	**Giribaldi**

Cayman Divers World Unlimited **Grand Cayman**
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:

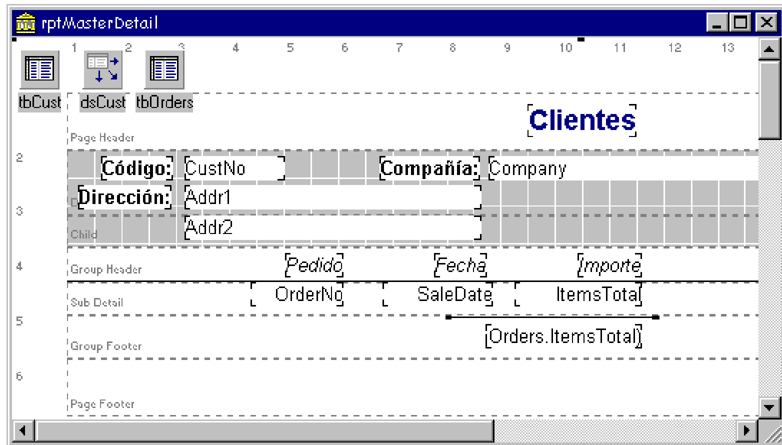
```
select C.CustNo, C.Company, O.OrderNo, O.SaleDate, O.ItemsTotal
from   Customer C, Orders O
where  C.CustNo = O.CustNo
```

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
Master	Apunta a un *TQuickReport* o a otro *TQRSubDetail*
DataSet	El conjunto de datos de detalles
Bands	Permite crear rápidamente la cabecera y el pie
HeaderBand	Una banda de tipo *rbGroupHeader*
FooterBand	Una banda de tipo *rbGroupFooter*

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 componente *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*. Esta clase contiene propiedad *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

QR2 introduce el componente *TQRCompositeReport*, que sirve para imprimir consecutivamente varios informes. Este componente tiene 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
   with QRCompositeReport1.Reports do
   begin
      Add(Form2.QuickReport1);
      Add(Form3.QuickReport1);
      Add(Form4.QuickReport1);
   end;
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

QuickReport permite la definición de un diálogo o ventana de previsualización creado por el programador. 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 añadírsele 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*.

Después, hay que interceptar el evento *OnPreview* del informe de esta manera:

```
procedure TInforme.QuickRep1Preview(Sender: TObject);
begin
   Visualizador.QRPreview1.QRPrinter := TQRPrinter(Sender);
   Visualizador.ShowModal;
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*, además del objeto *QRPrinter* global.

> Hasta Delphi 3, las ventanas de previsualización debían mostrarse de forma no modal. Esto ha sido corregido en QuickReport 3 para Delphi 4.

Existen innumerables 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
   with Form2.QuickRep1 do
   begin
      PrinterSetup;
      if Tag = 0 then
         Print;
   end;
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 Delphi, 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 QR: 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 **nil**, 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.

Ya hemos visto, en el capítulo sobre bases de datos y sesiones, cómo extraer información sobre los alias disponibles, las tablas que existen dentro de los mismos, y los campos que contiene cada tabla. De todos modos, mostraré el código asociado al mantenimiento de esta información. 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.

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 un procedimiento que permite enviar un *buffer* con códigos binarios directamente a la impresora, haciendo caso omiso de la interfaz de alto nivel del controlador. El procedimiento realiza llamadas al API de Windows, y utiliza el siguiente procedimiento auxiliar para transformar valores de retorno de error en nuestras queridas excepciones:

```
procedure PrinterError;
begin
   raise Exception.Create('Operación inválida con la impresora');
end;
```

No puedo entrar en explicaciones sobre cada una de las funciones utilizadas en el procedimiento, pues esto sobrepasa los objetivos del presente libro. He aquí el procedimiento:

```
procedure Imprimir(PrinterName: string;
   const Data; Count: Cardinal); overload;
type
   TDocInfo1 = record
      DocName, OutputFile, DataType: PChar;
   end;
var
   hPrinter: THandle;
   DocInfo: TDocInfo1;
   BytesWritten: Cardinal;
begin
   // Llenar estructura con la información sobre el documento
   DocInfo.DocName := 'Document';
   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,
               BytesWritten) 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;
   if BytesWritten <> Count then PrinterError;
end;
```

Esta otra versión del procedimiento ha sido adaptada para que imprima una secuencia de caracteres y códigos contenidos en una cadena en la impresora:

```
procedure Imprimir(PrinterName: string; const Data: string);
   overload;
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* al procedimiento *Imprimir*. *TMemoryStream* fue estudiado en el capítulo sobre tipos de datos.

Capítulo

35

Gráficos de decisión

ENTRE LOS COMPONENTES INTRODUCIDOS por Delphi 3, 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, mostraremos 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 la aplicación desarrollada en el capítulo 33, que trata sobre la gestión de libretas de banco.

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!

No voy a describir totalmente el proceso de desarrollo de una aplicación que calcule e imprima biorritmos. Mi interés es mostrar 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)
```

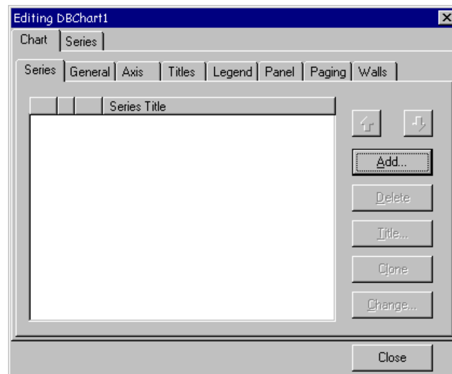
La proporción determina el ángulo que ocupa cada trozo de la tarta, y la etiqueta sirve ... pues para eso ... para etiquetar el trozo. En cambio, un gráfico lineal puede contener tripletas en vez de pares:

```
(etiqueta, valor X, valor Y)
```

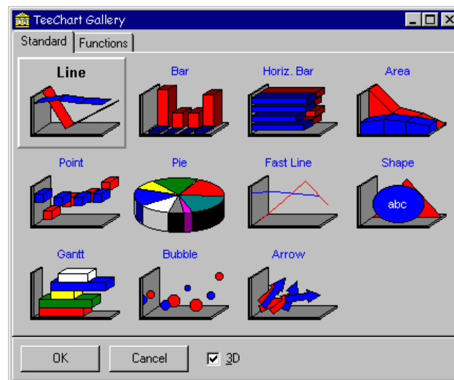
Ahora se ha incluido un valor X para calcular distancias en el eje horizontal. Si un gráfico lineal contiene tres puntos, uno para enero, uno para febrero y otro para diciembre, se espera que los puntos de enero y febrero estén más alejados del punto correspondiente a diciembre.

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 siguiente cuadro de diálogo:



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:

```
procedure Add(YValue: Double; const ALabel: string; AColor: TColor);
```

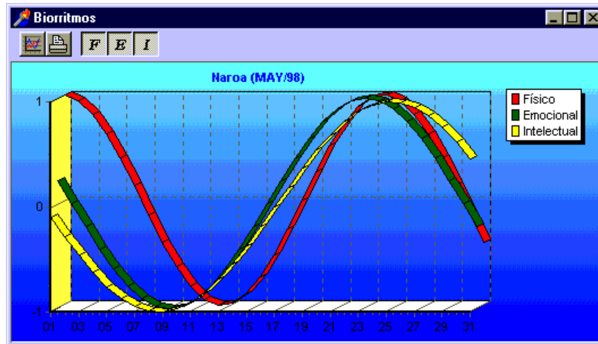
Utilizaremos este método, en vez de *AddXY*, porque nuestros puntos irán espaciados uniformemente. En el tercer parámetro podemos utilizar la constante especial de color *clTeeColor*, para que el componente decida qué color utilizar.

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 bio-

rritmos, y el número de meses que debe abarcar. Todo esto lo dejo en sus manos expertas. Me limitaré a mostrar la función que rellena el gráfico a partir de estos datos:

```
procedure TwndMain.FillChart(Nombre: string;
   Nacimiento, PrimerDia: TDateTime; Meses: Integer);
var
   UltimoDia: TDateTime;
   Y, M, D: Word;
   I, Dias: Integer;
   Pi2: Double;
   Etiqueta, MS1, MS2: string;
begin
   if Nombre = '' then
      Nombre := 'BIORRITMOS';
   Pi2 := 2 * Pi;
   DecodeDate(PrimerDia, Y, M, D);
   while Meses > 0 do
   begin
      Dec(Meses);
      Inc(M);
      if M > 12 then
      begin
         M := 1;
         Inc(Y);
      end;
   end;
   UltimoDia := EncodeDate(Y, M, 1) - 1;
   MS1 := UpperCase(FormatDateTime('mmm/yy', PrimerDia));
   MS2 := UpperCase(FormatDateTime('mmm/yy', UltimoDia));
   if MS1 <> MS2 then MS1 := MS1 + '-' + MS2;
   Chart1.Title.Text.Text := Nombre + ' (' + MS1 + ')';
   Series1.Clear;
   Series2.Clear;
   Series3.Clear;
   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;
```

El resultado de la ejecución del programa puede verse a continuación:



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;
```

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

Volvemos a 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:

- Evolución del saldo.
- Rejilla de análisis.
- 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:

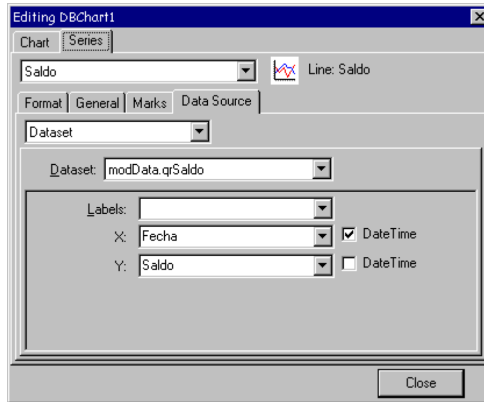
```
select Fecha, Saldo
from   Apuntes
where   Libreta = :Codigo
order  by Fecha
```

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.

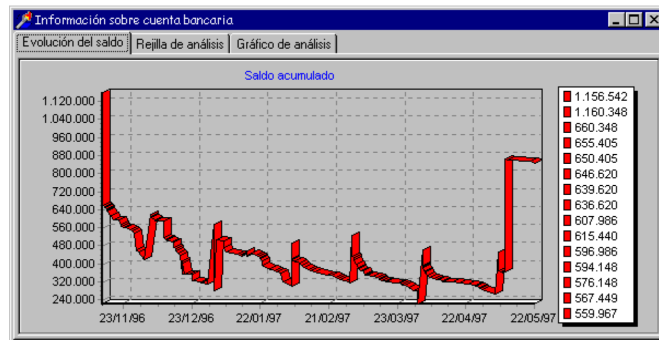
Ahora vamos a la primera página del formulario principal, seleccionamos la página *Data Controls* de la Paleta de Componentes, y traemos un *TDBChart*, cambiándole su alineación *Align* a *alClient*. ¿Por qué *TDBChart*? Porque en este caso, los datos pueden obtenerse directamente de la base de datos.

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:



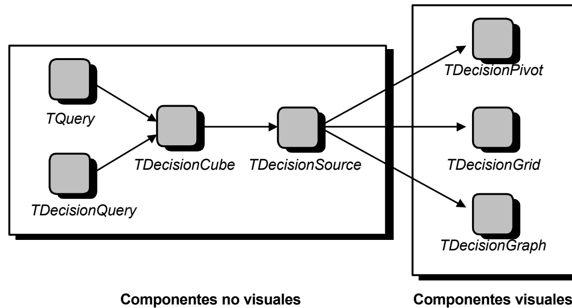
A continuación, podemos configurar detalles del gráfico, como el color del fondo, el título, etc. Puede también intentar añadir una serie que represente el saldo promedio, utilizando la página *Functions* de la galería de series. El gráfico final puede apreciarse en la siguiente figura:



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*:

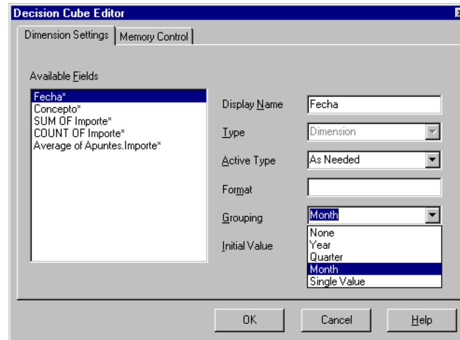
```
select A.Fecha, C.Concepto, sum(A.Importe), count(A.Importe)
from   Apuntes A, Conceptos C
where  A.Concepto = C.Codigo and
       A.Libreta = :Codigo
group  by A.Fecha, C.Concepto
```

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						
Concepto	nov, 1996	dic, 1996	ene, 1997	feb, 1997	mar, 1997	abr, 1997	may, 1997
Abono intereses	3.806,00 Pts	2.532,00 Pts					
Abono ints. 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	-92.689,00 Pts	-130.943,00 Pts	-63.989,00 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/lib			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			-49.486,00 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

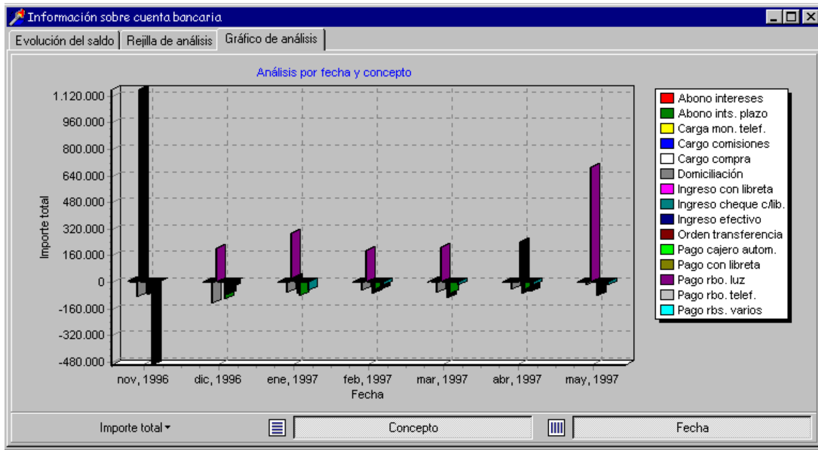
Como se observa en la imagen, en la dimensión horizontal aparece la fecha, mientras que la vertical corresponde a los distintos tipos de operaciones. Utilizando directamente la rejilla podemos intercambiar las dimensiones, mezclarlas, ocultarlas y restaurarlas. Mostramos a continuación, como ejemplo, el resultado de contraer la dimensión de la fecha; aparecen los gastos totales agrupados por conceptos:

Información sobre cuenta bancaria

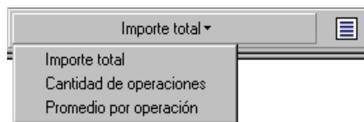
Evolución del saldo | Rejilla de análisis | Gráfico de análisis

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 Pt

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 alineará 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:

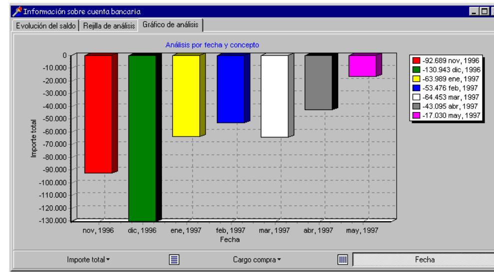


Mediante el pivote se puede cambiar el tipo de estadística que se muestra en el gráfico; para esto se utiliza el botón de la izquierda, que al ser pulsado muestra las funciones previstas durante el diseño de la consulta.



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.

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);
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*, que hay que incluir manualmente.

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, que funciona a la perfección en Delphi 3, provoca una excepción en Delphi 4.0. Este 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.

Capítulo

36

Bibliotecas de Enlace Dinámico

UNO DE LOS ELEMENTOS FUNDAMENTALES de la arquitectura de Windows como sistema operativo son las *Bibliotecas de Enlace Dinámico* (*Dynamic Link Libraries*), conocidas familiarmente como *DLLs*. No solamente son la forma en que se implementa el núcleo del sistema, sino que constituyen además el mecanismo básico de extensibilidad de Windows. Delphi, por supuesto, permite utilizar DLLs existentes y programar nuevas bibliotecas dinámicas. Las técnicas necesarias serán estudiadas en este capítulo.

Arquitectura básica

En los primeros tiempos, cuando el mundo aún olía a nuevo, si un programador quería utilizar una biblioteca de funciones de terceros, debía incluir físicamente el código de las rutinas dentro del ejecutable de su aplicación. En sistemas operativos primitivos como MS-DOS, esto no era un gran problema. Por el contrario, era percibido como una gran ventaja: "mi programa lleva todo incluido dentro de sí". Por supuesto, nadie se planteaba la ejecución simultánea de más de una aplicación. En sistemas operativos que soportan multitarea, cuando dos aplicaciones utilizan porciones de una misma biblioteca y están cargadas concurrentemente en un mismo ordenador, las rutinas comunes que ambas utilizan ocupan espacio duplicado en la memoria. Al tipo de enlace en el que el código de todas las rutinas necesarias se integra físicamente dentro del fichero ejecutable se le denomina *enlace estático*. Es lo que sucede normalmente con las unidades de Delphi, si no hemos activado el uso de paquetes.

La alternativa es utilizar el *enlace dinámico*: la biblioteca debe estar presente en tiempo de ejecución, y el código de las rutinas necesarias se incorpora a la aplicación durante la carga de la misma, o cuando la aplicación necesita una función por primera vez. La primera vez que vi aplicar esta técnica fue, curiosamente, con el sistema operativo UCSD-Pascal. El compilador de Pascal de este sistema no enlazaba estáticamente las unidades, sino que realizaba esta carga de forma dinámica. Por supuesto, existía una estrecha coordinación entre el diseño del lenguaje y el sistema operativo, algo impensable en nuestros turbulentos días.

La principal ventaja del enlace dinámico es que permite evitar cargar repetidamente la misma biblioteca si más de una aplicación hace uso simultáneo de ella. El segmento de código de la biblioteca puede ser compartido por varios programas a la vez. La biblioteca se descarga en el momento en que el último programa que la está utilizando se despide de este mundo cruel. Pero las bibliotecas dinámicas también abren el camino a la *extensibilidad* de las aplicaciones: un programa puede ejecutar una función no prevista durante su diseño.

Cuando Microsoft diseñó Windows, decidió hacer un uso intensivo de este recurso. La interfaz de programación de MS-DOS se realizaba mediante interrupciones, lo cual complicaba la programación enormemente. En contraste, el API de Windows consiste en llamadas a funciones. Ahora bien, ¿dónde hay que colocar estas funciones? ¿Quizás en bibliotecas estáticas? Imposible: los programas crecerían desmesuradamente, por lo cual el código de las bibliotecas debe colocarse fuera de los ejecutables. Y de este modo se resuelve también la duplicación de código que podría causar la concurrencia. Como resultado, el núcleo del sistema se programó en forma de bibliotecas de enlace dinámico. En las primeras versiones de Windows, éstas eran unas pocas: *kernel.exe*, para las funciones básicas, *user.exe*, para el manejo de ventanas, y *gdi.exe* para el motor de gráficos. Observe que, aunque todos estos ficheros eran DLLs, tenían la extensión *exe*. Posteriormente, en la medida en que fueron apareciendo nuevos servicios, como DDE, OLE, multimedia y comunicaciones, se incorporaron nuevas DLLs al repertorio básico. Pero lo más importante es que se estimuló la creación y uso de DLLs por los programadores: el novedoso recurso no quedaba reservado a los programadores del sistema operativo.

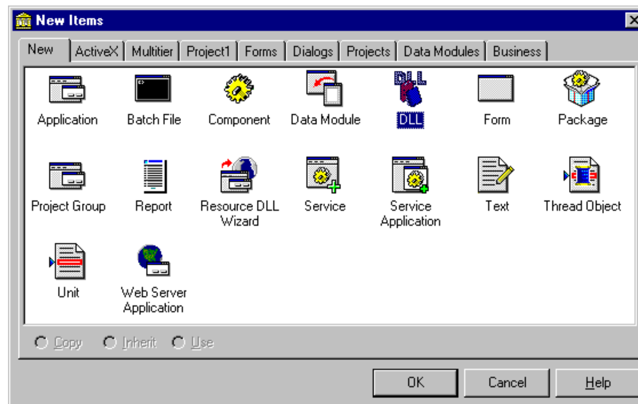
La aparición de versiones de Windows en modo protegido, el paso a una arquitectura de 32 bits y la sustitución de la concurrencia cooperativa por la apropiativa, provocaron cambios importantes en la implementación de la carga dinámica en Windows. Las versiones actuales de Windows utilizan una importante técnica conocida como *ficheros asignados en memoria* (*memory mapped files*) para compartir el mismo código de una DLL por varios programas. Esta técnica actúa directamente sobre la forma en que trabaja la memoria virtual del sistema. Al finalizar este capítulo presentaré un ejemplo de uso de los ficheros asignados en memoria.

En Windows 3.x, todas las instancias de una DLL compartían el mismo segmento de datos. De esta manera, las variables globales de la DLL podían utilizarse como forma de comunicación entre procesos que utilizaban una misma DLL. Esta técnica, sin embargo, se hizo inviable con la aparición de las versiones de 32 bits de Windows, pues no tenía en cuenta el modelo de multitarea apropiativa, sino el cooperativo. De haberse mantenido el modelo, dos aplicaciones podrían intentar concurrentemente modificar el valor de una misma variable, con todos los problemas de sincronización asociados. Por lo tanto, se decidió que en estas versiones, cada instancia de una DLL contara con un segmento de datos particular. Si dos instancias de una DLL deben

intercambiar información entre sí, deben hacerlo a través de ficheros asignados a memoria.

Proyectos DLL

Como hemos explicado casi al principio de este libro, las DLLs se crean en Delphi mediante proyectos especiales que comienzan con la palabra clave **library** en vez de **program**. El esqueleto sintáctico de este tipo de proyecto puede crearse mediante el Depósito de Objetos, mediante el icono nombrado *DLL*:



Aunque el proyecto generado no incluye unidades, un proyecto de DLL de Delphi tiene las mismas posibilidades de estructura que un proyecto de aplicación. Pueden añadirse incluso unidades que contengan formularios, y hacer uso de paquetes para evitar el enlace estático del código de la VCL. A continuación, mostramos la estructura sintáctica de un proyecto DLL:

```
library NombreDLL;

uses
        Lista de Unidades;

        Declaraciones

exports
        Lista de rutinas exportadas

begin
        Lista de instrucciones
end.
```

Exportación de funciones

El objetivo fundamental de una DLL es el *exportar* rutinas, de modo que puedan ser aprovechadas por aplicaciones u otras DLLs. No todas las funciones definidas en una DLL tienen por qué ser públicas. Ahora bien, las funciones exportadas deben estar preparadas para ser ejecutadas desde los entornos de programación más variados.

Lo primero a tener en cuenta es el convenio de traspaso de parámetros que se va a utilizar con la función. Hay que tener en cuenta que las rutinas exportadas en una DLL pueden ser ejecutadas desde otros lenguajes, cada uno con idiosincrasias diferentes. ¿Qué diferencia a un lenguaje de otro, en el momento de llamar a una rutina? Fundamentalmente estos dos puntos:

- El orden en que se pasan los parámetros.

  Los primeros compiladores de Pascal fueron diseñados pensando en una alta velocidad de compilación. Para lograr esto, los parámetros de las funciones se introducían en la pila en el mismo orden en que se escriben y en el que los encuentra el compilador: de izquierda a derecha. Si la llamada es *PropinarPaliza(Pepe, Paco)*, primero se introduce *Pepe* y luego *Paco*.
  Pero el lenguaje C permite definir funciones con cantidad variable de parámetros. El caso más conocido quizás sea *printf*, el equivalente al *Write* pascaliano de los tiempos de los monitores verdes. Para poder implementar este tipo de funciones, es necesario pasar los parámetros en orden inverso: si la llamada es *PropinarPaliza(Pepe, Paco, Juan*), el último en introducirse es *Pepe*. De esta forma el primer parámetro se encuentra situado en el tope de la pila, y sirve como punto de partida al compilador para encontrar a los demás.

- Quién extrae los parámetros de la pila.

  Las funciones con cantidad variable de parámetros tienen otro problema: como la función no puede saber de antemano cuántos parámetros le han sido suministrados, no puede extraer ella misma los parámetros de la pila. En C y C++, esto es responsabilidad de quien ejecuta la función. Por razones obvias, se ahorra espacio si la responsable de la extracción es la propia rutina, pues el código necesario no se repite por cada llamada. Además, los procesadores de Intel tienen instrucciones especiales que combinan el retorno de la función con la limpieza de la pila, con lo que también es más rápido el proceso de llamados al estilo Pascal.

Delphi admite asociar a las rutinas las siguientes directivas para especificar el convenio de traspaso de parámetros:

Directiva	Convenio
pascal	Los parámetros se introducen en la pila de izquierda a derecha. La propia función extrae los parámetros. Es el convenio por omisión en Delphi 1, compatible con muchos compiladores de C y C++.
register	Optimización en la que algunos parámetros se pasan en registros de la CPU. No es recomendable en DLLs.
cdecl	Los parámetros se introducen de derecha a izquierda. El que llama a la función es quien tiene que extraer los parámetros. Permite crear funciones con cantidad variable de parámetros, *á la C*.
stdcall	Cóctel a base de **cdecl** y **pascal**: los parámetros se pasan de derecha a izquierda, pero la propia función extrae los parámetros de la pila. Es el convenio preferido en DLLs, pues las funciones del API de Windows lo emplean.
safecall	Utilizado por COM, se encarga automáticamente de la propagación de excepciones. No utilizar fuera de este modelo de programación.

Además de especificar el convenio de paso de parámetros, si la DLL que programamos se va a compilar para 16 bits con Delphi 1, es necesario preparar las funciones y procedimientos seleccionados para su exportación. La culpa la tiene esta vez principalmente el segmento de datos de la DLL, que no tiene que coincidir con el de la aplicación o DLL que ejecuta sus funciones. Por lo tanto, es necesario que el código inicial de la rutina se encargue de este cambio de contexto; claro está, hay que generar también código especial para que cuando termine o se suspenda la ejecución de la rutina se restaure el contexto original. La declaración típica de una rutina que pensamos exportar en una DLL es similar a la siguiente:

```
procedure MiFuncion(Param1: Integer; Param2: PChar) stdcall; export;
```

Si se trata de una DLL de 32 bits, incluir una cláusula **export** no tiene efecto alguno, aunque tampoco hace daño.

Pero la cláusula **export** solamente "prepara" la rutina para que pueda ser llamada desde otro contexto. Para que la función o procedimiento aparezca en la tabla de enlace de la biblioteca, y pueda por lo tanto ser aprovechada por otros módulos, hace falta mencionarla en la cláusula **exports**, que es global a todo el código fuente de la biblioteca, y se sitúa justo antes del código de inicialización de la DLL. En su forma más sencilla, esta cláusula menciona una lista de nombres de funciones, separados por comas:

```
library Ejemplo;

procedure Procedimiento1(A, B: Integer); stdcall; export;
begin
   // …
end;

function Funcion2(S: PChar): Integer; stdcall; export;
begin
   // …
end;

exports
   Procedimiento1,
   Funcion2;
begin
   // Código de inicialización
end.
```

Es importante comprender que las funciones a exportar pueden haber sido definidas en las interfaces de otras unidades que forman parte del proyecto. Solamente por simplicidad la mayoría de los ejemplos de éste y otros libros insisten en definir las funciones a exportar en el mismo fichero *dpr*.

Al exportar una función tenemos la posibilidad de hacer público su *nombre* o su *índice*, o ambos. Esta es la información que se almacena en la tabla de enlace de la DLL para que los procesos que la utilizan puedan buscar las rutinas que necesitan. La búsqueda por índice es mucho más rápida y eficiente que la búsqueda por nombre; el aspecto negativo de los índices es que es más fácil equivocarse con un número que con un nombre, al importar la rutina para utilizarla desde otro módulo. También hay que tener cuidado con la exportación por nombre, pues el sistema operativo distingue entre mayúsculas y minúsculas en los nombres de funciones exportadas. Por otra parte, los índices de exportación deben comenzar a partir de 1.

La exportación por nombre o índice se efectúa mediante las directivas **index** y **name**, en la propia cláusula **exports**. Por ejemplo:

```
exports
   Procedimiento1 name 'Procedure1',
   Funcion2 index 1,
   Funcion3 name 'Function3' index 2;
```

En realidad, la directiva **name** debe emplearse cuando queremos exportar una rutina con un nombre diferente pues, si no se utilizan explícitamente **name** o **index**, la exportación procede por nombre, utilizando el identificador de la función o procedimiento.

Importación de funciones y unidades de importación

Ahora supondremos que tenemos una DLL a nuestra disposición, y que queremos utilizar sus funciones. Lo primero que necesitamos es la documentación con los nombres de las funciones, sus índices si es que los tienen, y los parámetros y tipos de retorno que requieren. Con excepción de los nombres, esta información no puede extraerse de forma automática a partir del fichero *dll*, de modo que tiene que ser suministrada por el fabricante.

Digamos, por ejemplo, que la DLL *unadll.dll* contiene un procedimiento exportado con el nombre *UnProcedimiento*, y que éste necesita un parámetro entero de 32 bits. En Delphi, antes de utilizar un identificador necesitamos que esté declarado, y en este caso lo logramos incluyendo, en algún punto anterior al uso del procedimiento, la siguiente declaración:

```
procedure UnProcedimiento(I: Integer);
   external 'unadll.dll' name 'UnProcedimiento';
```

En este caso, hemos realizado la importación por nombre. Al cargarse nuestra aplicación en memoria, Windows busca la DLL *unadll.dll*, primero en el directorio activo, y luego en los directorios especificados con la variable de entorno *PATH*. Si no se encuentra la DLL o la función que queremos enlazar dinámicamente, la carga del programa falla.

También se pueden importar rutinas por índice, o asignar otro nombre a una rutina para nuestro uso particular:

```
procedure MiProc(I: Integer);
   external 'unadll.dll' name 'UnProcedimiento';
function Funcion1(I: Integer): Integer;
   external 'unadll.dll' index 1;
```

En cualquier caso, la directiva **external** sustituye al cuerpo del procedimiento o función.

Cuando una DLL se utiliza con bastante frecuencia en Delphi, es conveniente agrupar las declaraciones externas de funciones en una unidad, de modo que no queden dispersas por todo el código de los proyectos. En la interfaz de la unidad se declara el prototipo de las rutinas, mientras que en la implementación éstas se importan desde la DLL. Posiblemente, en la sección de interfaz se incluyan también las declaraciones de constantes y tipos de datos que son utilizados por las funciones y procedimientos de la DLL. A este tipo de unidades se les denomina *import units*, o *unidades de importación*.

El caso más notable de unidad de importación en Delphi es la unidad *Windows*, que declara los tipos de datos, constantes y rutinas del API básico del sistema operativo. Esta unidad sustituyó, a partir de Delphi 2, a las unidades *WinTypes* y *WinProcs* de Delphi 1. Le recomiendo que abra el código fuente de este fichero desde Delphi y estudie la estructura global de la unidad.

Una unidad de importación no aporta código ni datos al ejecutable producido por un proyecto. La información suministrada en la unidad es utilizada durante la compilación para la verificación de tipos, y durante el enlace para generar la tabla de resolución de referencias externas.

Si usted compra un software de programación basado en DLLs, asegúrese de que incluya la unidad de importación correspondiente para Delphi, pues le va a ahorrar el trabajo de escribirla y los inevitables fallos en el proceso de conversión de tipos de datos. Recuerde que una DLL es independiente del lenguaje en que se va a utilizar; cuando la propaganda comercial anuncia: "¡Compatible con Delphi!" quiere decir en realidad que hay alguien con suficiente sentido común en la compañía que se ha acordado de nosotros y nos ha preparado la unidad de importación.

Tipos de datos en funciones exportadas

Tenemos toda la libertad del mundo para exportar rutinas con los tipos de datos que deseemos. Ahora bien, recuerde que si estos tipos son particulares de Delphi, nos costará esfuerzos para que la DLL pueda utilizarse desde otro lenguaje. Por ejemplo, si necesitamos recibir en un procedimiento una cadena de caracteres es preferible declarar el correspondiente parámetro con el tipo *PChar*, que es compatible con las cadenas de caracteres de C y C++.

Realmente, podemos también utilizar parámetros de tipo **string**. Pero si estamos en Delphi 2 ó 3, tenemos que tomar medidas especiales para que la DLL funcione con otros lenguajes. En particular, la primera unidad mencionada en la cláusula **uses** del proyecto de DLL debe ser *ShareMem*. Además, tenemos que distribuir con nuestra DLL la biblioteca dinámica *delphimm.dll*, que viene con Delphi.

Problemas similares ocurren con los tipos de clases. Los parámetros de clase son percibidos en otros lenguajes como simples punteros. Incluso, si utilizamos una rutina con parámetros de clase desde otro proyecto de Delphi, se nos pueden presentar dificultades con la identificación de tipos en tiempo de ejecución: el operador **is**, por ejemplo, puede no reconocer la pertenencia de un objeto a una clase determinada. Recuerde que este tipo de información se deduce a partir de los punteros almacenados en los objetos a sus VMTs. Como las VMTs residen en los segmentos de datos y la DLL y la aplicación tienen cada uno el suyo, existirán copias duplicadas y no idénticas en cada módulo, confundiendo al operador **is**.

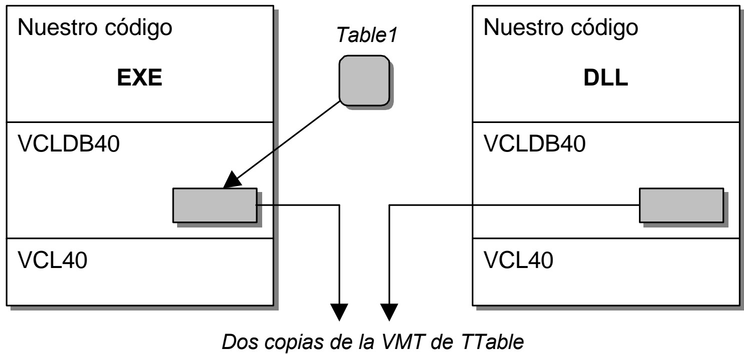
Voy a adelantarme y dar la solución del problema: si se le presenta este problema, debe utilizar *packages* tanto en el ejecutable como en la DLL que contiene la función. Vamos a mostrar en un sencillo ejemplo cómo pueden presentarse estas dificultades. Creemos, en primer lugar, una DLL sencilla que tome un parámetro de cualquier tipo de objeto. Y muy importante: asegúrese de que la DLL no está utilizando paquetes en tiempo de ejecución. Este es un posible ejemplo:

```
library DllPrj;
uses
   SysUtils, Classes, DB, DBTables;

function TestRTTI(D: TDataSet): string; stdcall;
begin
   if D is TTable then
      Result := 'Esto es una tabla'
   else if D is TQuery then
      Result := 'Esto es una consulta'
   else
      Result := '¿Qué me has pasado, colega?';
end;

exports
   TestRTTI;
begin  // No hay código de inicialización
end.
```

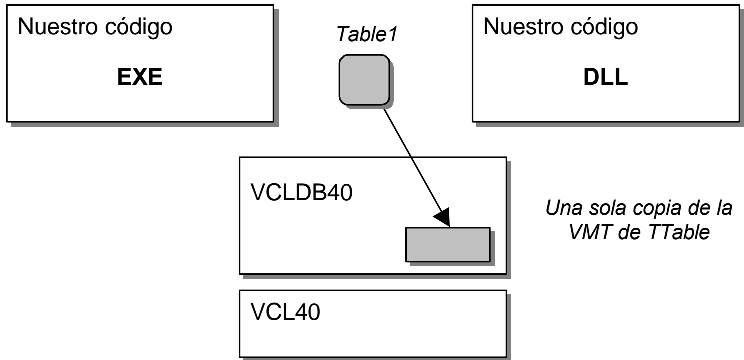
Ahora crearemos un ejecutable que llame a la función anterior. Compruebe también que esta aplicación tampoco está basada en paquetes de tiempo de ejecución, aunque esto no es necesario para hacerla fallar. La parte que nos interesa es la siguiente:

```
function TestRTTI(D: TDataSet): string; stdcall;
   external 'DllPrj.dll';

procedure TForm1.Button1Click(Sender: TObject);
begin
   ShowMessage(TestRTTI(Table1));
end;
```

El objeto *Table1* ha sido creado en tiempo de diseño y pertenece, como es de suponer, a la clase *TTable*. ¡Pero la DLL analiza el objeto con aire de sorpresa y nos pregunta enfadada qué nos traemos entre manos! Como ya hemos dicho, basta con compilar *ambos* proyectos con la opción de paquetes en tiempo de ejecución, para que se resuelva el problema mencionado. Este es un esquema de memoria de la situación que se presenta cuando ambos proyectos cargan con todo su código:



Por el contrario, esto es lo que sucede cuando ambos utilizan *packages*:



Código de inicialización y finalización

Sintácticamente, una DLL no es una unidad. En vez de utilizar una sección **initialization** para indicar el código inicial del módulo, se utiliza un bloque de instrucciones, parecido al de los programas, al final del fichero. Esas instrucciones se ejecutan cada vez que otro módulo carga la DLL para uso propio.

En cambio, es un poco más complicado incluir un código de finalización. En primer lugar, recuerde que las secciones **finalization** de las unidades incluidas en la DLL se ejecutarán normalmente al terminar el programa. A nivel global (de proyecto) la forma más sencilla de especificar un código de finalización consiste en aprovechar la cadena de procedimientos de finalización de Delphi, controlada por la variable global *ExitProc*, declarada en la unidad *System*. Esta variable debe apuntar a un procedimiento, y antes de terminar la ejecución de una aplicación en Delphi es utilizada por una instrucción parecida a la siguiente:

```
   while Assigned(ExitProc) do
      ExitProc;
   // …
```

La razón de este comportamiento es mantener una cadena de procedimientos de salida, para lo cual deben cooperar los procedimientos instalados. A continuación mostramos un fragmento de proyecto en el cual se define e instala un procedimiento de salida:

```
library XYZ;

// Una variable global, para mantener la cadena
var
   Proximo: Pointer;

procedure MiProcedimiento;
begin
   // Restaurar el procedimiento de salida antiguo
   ExitProc := Proximo;
   // … El código necesario va aquí …
end;

// El código de inicialización
begin
   Proximo := ExitProc;
   ExitProc := @MiProcedimiento;
end. // Fin de proyecto
```

El procedimiento *AddExitProc*, disponible desde Delphi 1, simplifica un poco todo este mantenimiento. Sin embargo, hay que tener mucho cuidado con los procedimientos de salida al estilo *ExitProc*, pues no son compatibles con los paquetes de Delphi. Podemos también usar el procedimiento alternativo *AddTerminateProc*, que es seguro, pero que requiere la inclusión de la VCL.

Otra forma de disponer de procedimientos de salida, es utilizar la variable de puntero *DllProc*. Esta variable debe apuntar a un procedimiento con la siguiente declaración:

```
procedure MiDllProc(Motivo: Integer);
```

Este procedimiento se llamará cuando la DLL se descargue desde otro proceso (el "motivo" es la constante *DLL_PROCESS_DETACH*), cuando el proceso padre inicie un hilo de ejecución paralela (*DLL_THREAD_ATTACH*) o cuando lo finalice (*DLL_THREAD_DETACH*).

Los programadores de C/C++ deben definir procedimientos *LibMain* y *WEP*, en Windows 16, y *DllEntryPoint*, para Windows 32. Delphi, en cambio, no permite definir directamente estos puntos de entrada y hay que utilizar las alternativas explicadas anteriormente.

Excepciones en bibliotecas dinámicas

Una función exportada por una DLL escrita en Delphi puede fallar al intentar cumplir su contrato. Normalmente, las aplicaciones desarrolladas con Delphi lanzan excepciones cuando suceden estos fallos. Sin embargo, nuestras DLLs pueden ser utilizadas desde otros módulos escritos en Dios sabe qué lenguajes. ¿Qué sucede si una función exportada termina su ejecución con una excepción, y está siendo llamada desde un programa desarrollado con Visual Basic?

Aunque Delphi permite disfrazar la excepción de modo que pueda ser tratada como una excepción del sistema operativo, la mayoría de los lenguajes existentes no está preparada para tal asunto. Por lo tanto (y para desgracia nuestra), lo más conveniente es que las funciones que exporta la DLL "absorban" todas las excepciones que lleguen hasta ellas, y las sustituyan por el anticuado y peligroso sistema de los códigos de error:

```
var
   FUltimoError: string;

procedure UltimoError(Buffer: PChar; MaxLen: Cardinal); stdcall;
begin
   StrPLCopy(Buffer, FUltimoError, MaxLen);
end;

function Usame(Parametro1: Integer): Boolean; stdcall;
begin
   Result := True;
   try
      LoQueSea(Parametro1);
   except
      on E: Exception do
      begin
         FUltimoError := E.Message;
         Result := False;
      end;
   end;
end;
```

Supondremos que *Usame* y *UltimoError* son rutinas exportadas por una DLL. La que en realidad le interesa al usuario es la primera, que además de intentar cumplir con algún misterioso cometido, devuelve un valor lógico para indicar si logró o no su propósito. En caso negativo, el programador que usa la DLL puede llamar a *UltimoCodigo* para recuperar el mensaje de error que le hubiera presentado la excepción. ¿Incómodo? En ningún momento he dicho lo contrario.

Por supuesto, si la DLL solamente se utilizará desde Delphi y C++ Builder puede olvidarse de todo lo dicho en esta sección y seguir programando como hasta ahora.

Carga dinámica de bibliotecas

La técnica de importación de rutinas anterior realiza el enlace de la biblioteca durante la carga de la aplicación. Es una forma muy cómoda de aprovechar una DLL, pues garantiza desde el mismo arranque de la aplicación el correcto funcionamiento en lo que se refiere al enlace de funciones externas, y conjuga agradablemente con la verificación estática de tipos de Delphi, C++ y los lenguajes modernos de programación. Tiene ciertas desventajas, en cambio: la carga de la aplicación se retrasa, al aumentar el tamaño de la tabla de relocalización y enlace, y debemos conocer de antemano, en tiempo de programación, qué funciones queremos utilizar y en qué DLL se encuentra cada una. Esto último, sobre todo, limita las promesas de extensibilidad de las bibliotecas dinámicas.

Este es el esquema básico para cargar dinámicamente y ejecutar un procedimiento contenido en una DLL de 32 bits:

```
procedure EjecutarFuncion(const DllName, FunctionName: string);
type
   TProc = procedure;
var
   Proc: TProc;
   Handle: THandle;
begin
   Handle := LoadLibrary(PChar(DllName));
   if Handle = 0 then
      raise Exception.Create('DLL no encontrada: ' + DllName);
   try
      Proc := GetProcAddress(Handle, PChar(FunctionName));
      if @Proc = nil then
         raise Exception.Create('Función no encontrada: ' +
            FunctionName);
      Proc;
   finally
      FreeLibrary(Handle);
   end;
end;
```

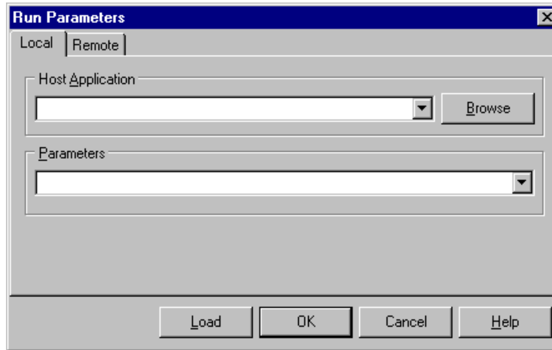
Observe, en primer lugar, que este procedimiento está limitado, debido a su diseño, a trabajar con rutinas sin parámetros. Si queremos trabajar con rutinas con parámetros debemos utilizar distintas versiones de *EjecutarProc* que definan de modo diferente el tipo interno *TProc*; esta es una limitación básica de Delphi y de la mayoría de los lenguajes de programación que conozco. En segundo lugar, tome nota de la correcta aplicación de las dos primeras reglas de Marteens para excepciones: si algo sale mal (contrato imposible de cumplir), lanzamos una excepción, y todo lo que pedimos (el *handle* de la biblioteca) debemos garantizar que es devuelto. Por último, está el detalle sintáctico de la arroba (@) frente al identificador *Proc* en la línea:

```
if @Proc = nil then // …
```

El hecho es que sin este operador de referencia, *Proc* puede interpretarse como una llamada a función sin parámetros. Por supuesto, *Proc* contiene la dirección de un procedimiento, no de una función, y el compilador señala, afortunadamente, el error.

Depuración de bibliotecas dinámicas

A partir de Delphi 3 podemos depurar el código de una biblioteca dinámica, dentro del propio Entorno de Desarrollo. Como una DLL no puede ejecutarse de forma independiente, necesitamos una aplicación matriz que utilice a la DLL para disparar el proceso de depuración. El nombre de esta aplicación se debe especificar mediante el comando de menú *Run|Parameters*:



La aplicación matriz (*Host application*) puede ser un programa de prueba sencillo que realice llamadas a las funciones exportadas que deseemos depurar. Los puntos de ruptura, por supuesto, deben colocarse en el código de la DLL.

Funciones de usuario en InterBase

Entre los sistemas que utilizan DLLs como forma de extender su funcionalidad está el propio servidor de InterBase. Como forma de ampliar el conjunto de funciones disponibles en SQL, los servidores de InterBase basados en Windows 95 y Windows NT admiten la creación de *funciones definidas por el usuario* (*User Defined Functions*, ó *UDF*). Estas son funciones definidas en DLLs que se deben registrar en el servidor, para poder 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 correspondencia entre tipos de datos de InterBase y de Delphi es sencilla: **int** se convierte en *Integer*, **smallint** en *SmallInt*, las cadenas de caracteres se pasan como punteros *PChar*, y así sucesivamente. En particular, las fechas se pasan en un tipo de registro con la siguiente declaración:

```
type
   TIBDate = record
      Days: LongInt;
      Frac: LongInt;
    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 diez milésimas de segundos transcurridas desde las doce de la noche. Con esta información en nuestras manos, es fácil programar la función *DiaSemana*:

```
function DiaSemana(var Date: TIBDate): PChar; cdecl;
const
   Tabla: array [0..6] of PChar = (
      'Miércoles', 'Jueves', 'Viernes', 'Sábado',
      'Domingo', 'Lunes', 'Martes');
```

```
begin
   Result := Tabla[Date.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. Observe la forma en que se devuelve la cadena de caracteres del resultado.

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, especialmente en InterBase 5, con las funciones que devuelven cadenas de caracteres generadas por la DLL. El problema es que estas funciones necesitan un *buffer* para devolver la cadena, que debe ser suministrado por la DLL. No se puede utilizar una variable global con este propósito, como en versiones anteriores de InterBase, debido a la nueva arquitectura multihilos. Ahora todas las conexiones de clientes comparten un mismo proceso en el servidor, y si varias de ellas utilizan una misma UDF, están accediendo a la función desde distintos hilos. Si utilizáramos una variable global, podríamos sobrescribir su contenido con mucha facilidad.

Por ejemplo, ésta es la implementación en Delphi de una función de usuario para convertir cadenas a minúsculas:

```
function Lower(S: PChar): PChar; cdecl;
var
   Len: Integer;
begin
   Len := StrLen(S);
   Result := SysGetMem(Len + 1);
   // SysGetMem asigna memoria con el formato del lenguaje C
   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 "imudf.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.

Bibliotecas de recursos e internacionalización

Normalmente, uno asocia las DLLs con técnicas para compartir código entre aplicaciones, o con la carga dinámica de funciones. Y un buen día descubrimos que pueden existir bibliotecas de enlace dinámico sin código. ¿Para qué sirven? Pues para compartir *recursos* entre programas, y para cargar recursos dinámicamente. Con la palabra *recursos* de la oración anterior me estoy refiriendo a los recursos típicos de Windows: mapas de bits, iconos, tablas de cadenas de caracteres, etc.

Las DLLs de recursos tienen múltiples aplicaciones prácticas, pero la que nos interesa en esta sección consiste en desarrollar versiones de nuestra aplicación para diferentes idiomas. Usted se preguntará: ¿y acaso mi aplicación se ejecutará algún día en Kuala Lumpur o en Calcuta? Probablemente no, pero si comprende cómo funciona este asunto comprenderá también cómo traducir esos mensajes en inglés de la VCL que tanto incordian a sus usuarios.

Pongámonos por un momento en la piel de Paco McFarland, el cotizado programador de Anaconda Software. Acaba de terminar su mejor aplicación, escrita de arriba abajo en el idioma de Jack The Ripper, pero ya le están pidiendo una copia para la sucursal de Anaconda en Barbate. Paco es un tío muy listo, y no escribe cosas como éstas:

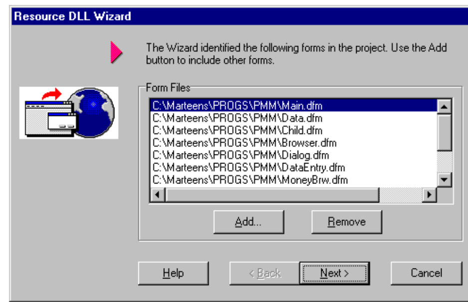
```
  if WeirdCondition then
  // Recuerde que Paco nació en Tucson, Arizona
     raise Exception.Create('What the heck is this?');
```

¡No, señor! El estilo de Paco es el siguiente:

```
resourcestring
   SWhatTheHeck = 'What the heck is this?';

// …

   if WeirdCondition then
      raise Exception.Create(SWhatTheHeck);
```

La declaración **resourcestring** añade una cadena a una tabla de cadenas de recursos administrada automáticamente por Delphi, y genera un identificador numérico para la misma. Para nosotros, los programadores, *SWhatTheHeck* es una constante de cadena normal. Pero Delphi, detrás del telón, se ocupa de leer el fichero de recursos acoplado a la aplicación durante la inicialización de esta unidad, y asigna la cadena leída a la variable que en realidad es *SWhatTheHeck*. Paco ha seguido esta técnica con todas las cadenas de mensajes que él utiliza en su aplicación, de modo que todas ellas residen en el fichero de recursos que se enlaza al ejecutable.

El señor McFarland carga la aplicación en el Entorno de Desarrollo, muestra el Depósito de Objetos y realiza un doble clic sobre el icono *Resource DLL Wizard*. La primera página de este asistente tiene el siguiente aspecto:



El asistente ha identificado todos los ficheros *dfm* que utiliza el proyecto y nos pide nuestra aprobación para copiarlos en la DLL que va a generar. Claro: no sólo hay que traducir las cadenas de mensajes, sino los títulos y etiquetas que aparecen en los formularios. Recuerde que los ficheros *dfm* se incluyen como un tipo especial de recurso dentro del ejecutable, para que los objetos de formularios puedan leer sus datos en tiempo de ejecución e inicializar sus propiedades publicadas.

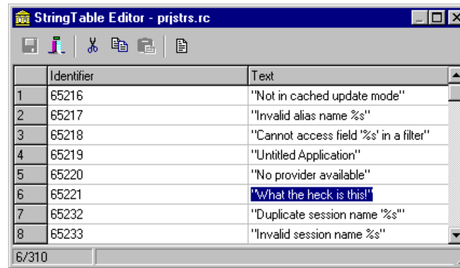
En la segunda página, se nos pide confirmación para el nombre del fichero donde se almacenará la tabla de cadenas de caracteres del proyecto, en formato fuente. Por omisión se utiliza el fichero *prjstrs.rc*. En la siguiente y última página se completan los datos para el asistente: el lenguaje al que se va a traducir, el nombre de la DLL a generar y el directorio donde se guardará el nuevo proyecto.

Ahora Paco tiene frente a sí un proyecto DLL con el mismo nombre que su aplicación; sin código fuente, pero plagado de directivas como las siguientes:

```
{$R 'Main.dfm' wndMain:TForm}
{$R 'Data.dfm' modData:TDataModule}
{$R 'Child.dfm' wndChild:TForm}
{$R 'Browser.dfm' wndBrowser:TForm(wndChild)}
```

Estas directivas instruyen al enlazador de Delphi para que incluya los ficheros *dfm* mencionados dentro de la DLL. Tenemos entonces a nuestra disposición copias de los *dfms* originales para modificar cualquier elemento de la interfaz que contenga un texto. Las modificaciones se realizan mediante el Inspector de Objetos, como si se tratara de un proyecto normal de Delphi, con la única salvedad de que no disponemos del código fuente que va asociado en los ficheros *pas*.

Las cadenas, por su parte, se editan a partir del fichero *rc*. La siguiente imagen muestra el editor gráfico de cadenas de recursos:



Una vez traducido todo lo que hay que traducir, Paco se estira, bosteza y compila la DLL. Ahora bien, el resultado no es un fichero de extensión *dll* sino *esn*, debido a la siguiente directiva de compilación generada por el asistente:

```
{$R esn}
```

Paco debe enviar a Barbate la misma versión del fichero ejecutable de la aplicación que se ejecuta en las oficinas de Tucson, pero debe incluir además la DLL del mismo nombre que el ejecutable y de extensión *esn*. Cuando la aplicación se ejecute en Barbate detectará que está trabajando con una versión de Windows en español (la *es* de la extensión) y con la alfabetización internacional (la *n* de la extensión). Por lo tanto, buscará sus recursos en un fichero de extensión *esn*, y si no lo encuentra tratará de utilizar la extensión *es* a secas. De esta forma, Paco hubiera podido generar una sola biblioteca de recursos para las sucursales de Barbate y de Aguascalientes, Méjico.

¿Qué provecho podemos sacar de todo lo anterior, nosotros que simplemente queremos mensajes en castellano? Le sugeriré tres alternativas, y en dos de ellas juegan un papel estrella las bibliotecas de recursos:

- Usted decide no utilizar paquetes en su aplicación.
  Busque en el directorio *source\vcl* de Delphi los ficheros *consts.pas*, *dbconsts.pas*, *bdeconsts.pas* y varios más que contienen declaraciones **resourcestring**. Traduzca las cadenas correspondientes a su dialecto natal y coloque el código fuente traducido en el directorio *lib* de Delphi. Cuando compile un proyecto, Delphi utilizará

la nueva versión de estas unidades, con los mensajes traducidos. Esta alternativa es la que más trabajo ahorra.

- Usted sigue decidiendo no utilizar paquetes en su aplicación.
  Haga lo que Paco McFarland. Escriba su aplicación en *spanglish* y cuando la tenga terminada, genere su DLL de recursos traducida. Problemas: cuando tenga que retocar su aplicación, necesitará retocar también la DLL, y este proceso tendrá que repetirlo para cada proyecto.
- Usted se hartó de no utilizar paquetes, y decide usarlos (o está obligado a ello). ¿Tiene un *Resource Workshop*[28], o alguna otra herramienta de edición de recursos a mano? En tal caso, cargue uno a uno los ficheros de paquetes *bpl*, extraiga los recursos y guárdelos en formato texto. Tradúzcalos entonces y genere usted sus propias DLL de recursos, con los mismos nombres de los paquetes, pero con extensiones *esn*. Cuando distribuya la aplicación, instale también junto a los paquetes las bibliotecas de recursos, en el directorio *windows\system*.

Ficheros asignados en memoria

Para cerrar el capítulo, demostraré como trabajar con ficheros asignados en memoria. Esta técnica tiene dos usos diferentes:

- Sirve para asociar directamente un fichero con memoria virtual. Usted dispone de un puntero al inicio de una zona de memoria con el contenido del fichero. La clave es que la lectura del fichero dentro de esta zona procede por demanda y de forma transparente para el programador. Además, cuando se necesita descartar páginas de esta memoria por razones de espacio, si la página no contiene modificaciones no es necesario grabarla en el archivo de memoria virtual del sistema, pues el sistema sabe que es la imagen de un sector del fichero original.
- También puede crearse un fichero asignado en memoria "sin fichero". Pasando un valor especial como identificador de fichero a la función del API de Windows correspondiente, se crea una zona de memoria compartida global. Cualquier otra aplicación puede hacer uso de la misma.

En vez de trabajar directamente con las funciones del API de Windows, es conveniente encapsularlas dentro de una clase como la siguiente:

```
type
  TMMFile = class
  protected
    FName: string;
```

[28] *Resource Workshop* venía incluido con las antiguas versiones de Borland C++ (hasta la 4.5x). Borland C++ 5.x integró la edición de recursos dentro del Entorno de Desarrollo ... y nos complicó la vida. El *RAD Pack* de Delphi, que se vendía en la época de Delphi 1, también contenía una copia de *RW*.

```
      FFileHandle: THandle;
      FHandle: THandle;
      FData: PChar;
      FSize: Cardinal;
   public
      constructor Create(const AName: string; ASize: Cardinal = 0);
      destructor Destroy; override;
      property Data: PChar read FData;
      property Size: Cardinal read FSize;
   end;
```

Si el programador no especifica un tamaño en el segundo parámetro de *Create* al construir el objeto, se crea el fichero asociado basándose en un fichero real cuyo nombre viene en el parámetro *AName*. En caso contrario, se crea una zona de memoria global compartida del tamaño indicado, y *AName* representa el nombre que identifica, de aplicación en aplicación, a dicha zona. En cualquiera de los dos casos, la propiedad *Data* apuntará a la zona de memoria obtenida, y *Size* será su tamaño.

La implementación del constructor es la siguiente:

```
resourcestring
   SCannotOpenFile = 'Cannot open file "%s"';
   SCannotCreateMapping = 'Cannot create mapping for file "%s"';
   SCannotMapFile = 'Cannot map file "%s" into memory';

constructor TMMFile.Create(const AName: string; ASize: Cardinal);

   procedure Error(const AMsg: string);
   begin
      raise Exception.CreateFmt(AMsg, [AName]);
   end;

begin
   inherited Create;
   if ASize > 0 then
   begin
      FFileHandle := $FFFFFFFF;
      FSize := ASize;
      FName := AName;
   end
   else
   begin
      FFileHandle := FileOpen(AName, fmOpenReadWrite);
      if FFileHandle = 0 then
         Error(SCannotOpenFile);
      FSize := GetFileSize(FFileHandle, nil);
      FName := '';
   end;
   FHandle := CreateFileMapping(FFileHandle, nil, PAGE_READWRITE, 0,
      FSize, PChar(FName));
   if FHandle = 0 then
      Error(SCannotCreateMapping);
   FData := MapViewOfFile(FHandle, FILE_MAP_WRITE, 0, 0, Size);
   if FData = nil then
      Error(SCannotMapFile);
   FName := AName;
end;
```

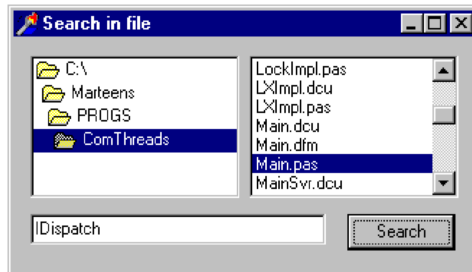
El destructor se encarga de deshacer todo lo realizado por el constructor:

```
destructor TMMFile.Destroy;
begin
   if FData <> nil then
      UnmapViewOfFile(FData);
   if FHandle <> 0 then
      CloseHandle(FHandle);
   if FFileHandle <> $FFFFFFFF then
      CloseHandle(FFileHandle);
   inherited Destroy;
end;
```

Una vez que tenemos una clase como ésta a nuestra disposición, es muy fácil idear aplicaciones para la misma. ¿Recuerda que en el capítulo sobre tipos de datos presenté una función *RabinKarp* para búsqueda dentro de cadenas?

```
function RabinKarp(const S1, S2: PChar; L1, L2: Integer): Integer;
   overload;
function RabinKarp(const S1, S2: string): Integer; overload;
```

Ahora podemos combinar ambas técnicas en un programa que busque un texto dado dentro de un fichero. He creado un formulario con un cuadro de directorios (*DirectoryListBox*) y un cuadro de lista de ficheros (*FileListBox*); ambos componentes están en la página *Win31* de la Paleta. Para sincronizar la acción de los mismos, se asigna *FileListBox1* en la propiedad *FileList* del cuadro de directorios. Debajo de ellos he añadido un cuadro de edición, para teclear el texto a buscar, y un botón para realizar la búsqueda. La parte básica del programa es la respuesta al evento *OnClick* de éste botón:




```
procedure TForm1.Button1Click(Sender: TObject);
begin
   with TMMFile.Create(FileListBox1.FileName) do
   try
      if RabinKarp(PChar(Edit1.Text), Data,
         Length(Edit1.Text), Size) = -1 then
         ShowMessage('Not found')
      else
         ShowMessage('Found!');
```

```
    finally
      Free;
    end;
  end;
```

Una advertencia final: si va a utilizar ficheros asignados en memoria como mecanismo para compartir memoria entre procesos, tenga en cuenta que necesitará de alguna técnica de sincronización. Es aconsejable que utilice *mutexes*, es decir, semáforos de exclusión mutua.

Capítulo

37

Servidores de Internet

PARA SER SINCEROS, TITULAR UN CAPÍTULO tal como lo hemos hecho revela un poco de presuntuosidad. Los temas relacionados con Internet son tantos que pueden redactarse libros completos (los hay). Tenemos el estudio de la infinidad de protocolos existentes, la comunicación mediante *sockets*, el envío y recepción de correo electrónico, el uso de controles ActiveX y las ActiveForms que nos permiten programar clientes de Internet inteligentes...

Sin embargo, este capítulo estará dedicado a un tema muy concreto: la programación de extensiones de servidores HTTP. Se trata de aplicaciones que se ejecutan en el servidor Web al que se conectan los navegadores de Internet, y que permiten el desarrollo de páginas con información dinámica. Tradicionalmente se han utilizado programas desarrollados para la interfaz CGI con este propósito. Delphi permite, a partir de la versión 3, desarrollar aplicaciones para la interfaz CGI, así como para las más modernas y eficientes interfaces ISAPI y NSAPI, de Microsoft y Netscape respectivamente. Pero lo más importante de todo es que podemos abstraernos del tipo concreto de interfaz final con la que va a trabajar la extensión. Para ilustrar estas técnicas, desarrollaremos una sencilla aplicación Web de búsqueda de información por palabras claves.

El modelo de interacción en la Web

El protocolo HTTP es un caso particular de arquitectura cliente/servidor. Un cliente, haciendo uso de algún navegador políticamente correcto, pide un "documento" a un servidor situado en determinado punto de la topología de la Internet; por ahora, no nos interesa saber cómo se produce la conexión física entre ambas máquinas. He puesto la palabra "documento" entre comillas porque, como veremos en breve, éste es un concepto con varias interpretaciones posibles.

Una vez que el servidor recibe la petición, envía al cliente el documento solicitado, si es que éste existe. Lo más importante que hay que comprender es lo siguiente: una vez enviado el documento, el servidor se desentiende totalmente del cliente. El servidor nunca sabrá si el cliente estuvo mirando la página uno o quince minutos, o si

cuando terminó siguió o no ese enlace que le habíamos recomendado al final de la página. Esta es la característica más abominable de Internet, inducida por las peculiaridades del hardware y los medios de transporte de la información, pero que complica gravemente la programación de aplicaciones para la red global. El navegante que utiliza Internet principalmente para bajarse imágenes bonitas, excitantes o jocosas, incluye en sus plegarias el que Internet 2 aumente su velocidad de transmisión. Nosotros, los Programadores De Negro, rezamos porque mejore su protocolo infernal.

Para solicitar un "documento" al servidor, el cliente debe indicar qué es lo que desea mediante una cadena conocida como URL: *Uniform Resource Locator*. La primera parte de una URL tiene el propósito de indicar en qué máquina vamos a buscar la página que deseamos, y qué protocolo de comunicación vamos a utilizar:

```
http://www.marteens.com/trucos/truco01.htm
```

El protocolo utilizado será HTTP; podría haber sido, por ejemplo, FILE, para abrir un documento local. La máquina queda identificada por el nombre de dominio:

```
www.marteens.com
```

El resto de la URL anterior describe un directorio dentro de la máquina, y un documento HTML situado dentro de este directorio:

```
trucos/truco01.htm
```

Este directorio es relativo al directorio definido como público por el servidor HTTP instalado en la máquina. En mi portátil, por ejemplo, el directorio público HTTP se llama *webshare*, y ha sido definido por el Personal Web Server de Microsoft, que es el servidor que utilizo para pruebas.

Más adelante veremos que una URL no tiene por qué terminar con el nombre de un fichero estático que contiene texto HTML, sino que puede apuntar a programas y bibliotecas dinámicas, y ahí será donde tendremos la oportunidad de utilizar a nuestro querido Delphi.

Aprenda HTML en 14 minutos

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*. Los códigos de HTML, o *etiquetas*, se encierran entre paréntesis angulares: `<>`. Una gran parte de las etiquetas consiste en pares de apertura y cierre, como `<TITLE>` y `</TITLE>`; la etiqueta de cierre utiliza el mismo nombre que la de apertura, pero precedido por una barra inclinada. Otras etiquetas, como `<HR>`, que dibuja una línea horizontal, realizan su cometido por sí mismas y no vienen por pares.

Existe un amplio repertorio de etiquetas que sirven para dar formato al texto de un documento. Entre éstas tenemos, por ejemplo:

Etiqueta	**Significado**
`<H1>, <H2> ...`	Encabezamientos
`<P>`	Marca de párrafo
`<UL>, <OL>`	Listas ordenadas y sin ordenar
`<IMG>`	Inclusión de imágenes
`<PRE>`	Texto preformateado

Una de las etiquetas más importantes es el *ancla* (*anchor*), que permite navegar a otra página. 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.inprise.com>Inprise
Corporation</A> para más información sobre Delphi.</P>
</BODY>
</HTML>
```

En el ejemplo anterior, la dirección del enlace que se ha suministrado es una URL completa, pero podemos incorporar enlaces locales: dentro del propio servidor donde se encuentra la página actual, e incluso a determinada posición dentro de la propia página. Existen más etiquetas de este tipo, como la etiqueta `<ADDRESS>`, que indica una dirección de correo electrónico. También se puede asociar un enlace a una imagen, colocando una etiqueta `<IMG>` dentro de un ancla.

El subconjunto de HTML antes descrito ofrece, en cierto sentido, las mismas características del sistema de ayuda de Windows. Es, en definitiva, una herramienta para definir sistemas de navegación a través de textos (vale, y gráficos también). Pero se necesita algo más si se desea una comunicación mínima desde el cliente hacia el servidor, y ese algo lo proporcionan los *formularios HTML*. Mediante estos formularios, el usuario tiene a su disposición una serie de controles de edición (no muy sofisticados, a decir verdad) que le permiten teclear datos en la pantalla del navegador y remitirlos explícitamente al servidor. Un formulario ocupa una porción de un documento HTML:

```
<HTML>
<HEAD><TITLE>Mi primer documento</TITLE></HEAD>
<BODY>
<H1>¡Hola, mundo!</H1>
<HR>
<FORM METHOD=GET ACTION=http://www.marteens.com/scripts/prg.exe>
<P>Nombre: <INPUT TYPE="TEXT" NAME="Nombre"></P>
<P>Apellidos: <INPUT TYPE="TEXT" NAME="Apellidos"></P>
<INPUT TYPE="SUBMIT" VALUE="Enviar">
</FORM>
</BODY>
</HTML>
```

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. El método casi siempre es 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.

Extensiones del servidor y páginas dinámicas

La etiqueta ACTION indica a qué URL se dirige la petición del cliente. En el ejemplo hemos utilizado esta cadena:

```
http://www.marteens.com/scripts/prg.exe
```

Ya lo habíamos anunciado: una URL no terminaba necesariamente con el nombre de un fichero HTML. En este caso, termina con el nombre de un programa: *prg.exe*. El propósito de este programa es generar dinámicamente el texto correspondiente a la página HTML que debe enviarse al cliente. Esta aplicación se ejecutará con una serie de parámetros, que se construirán tomando como base los valores suministrados por el cliente al formulario. Por supuesto, el algoritmo de generación de la página tendrá en cuenta estos parámetros.

A este tipo de programas creadores de páginas dinámicas se le conoce como 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. 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.

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 estas 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, de modo que una segunda petición de la misma extensión no necesite volver a cargarla.
- La comunicación es más rápida, porque el servidor ejecuta una función de la DLL, pasando los 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 está cargado una sola vez.

¿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 falla.

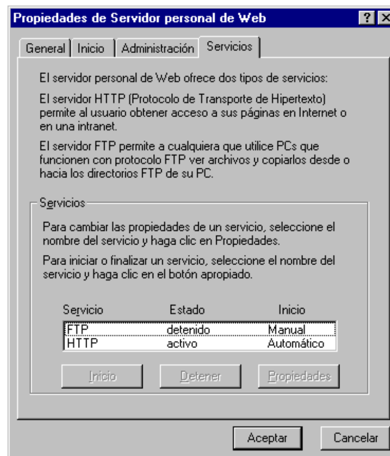
Nos queda pendiente explicar cómo se pasan los parámetros a la extensión del servidor. Supongamos que el usuario teclea, en el formulario que hemos utilizado como ejemplo, su nombre y apellidos, y pulsa el botón para enviar la solicitud. Como el método asociado al formulario es GET, la URL que se buscará tendrá la siguiente forma completa:

```
http://www.marteens.com/scripts/
   prg.exe?Nombre=Ian&Apellidos=Marteens
```

Si el método fuera POST, los datos que aparecen después del signo de interrogación se codificarían dentro del propio protocolo HTTP. Pero en el fondo se estaría pasando la misma información que en el otro caso.

¿Qué necesito para este seguir los ejemplos?

Para desarrollar extensiones de Web con Delphi va a necesitar un Delphi ... pero creo que esto ya se lo imaginaba. Además, hace falta un explorador de Internet, de la marca que sea y un servidor HTTP. Aquí es donde pueden complicarse un poco las cosas. Veamos, ¿tiene usted un Windows NT? Más bien, ¿tiene un BackOffice, de Microsoft? Entonces tendrá un Internet Information Server, o podrá instalarlo si todavía no lo ha hecho. ¿Y qué pasa si el lector pertenece a la secta que clava alfileres en el mapa en el sitio donde dice Redmond? Siempre puede utilizar el servidor de la competencia, el Netscape FastTrack.



Otra solución es utilizar Personal Web Server para Windows 95, un producto que puede conseguirse gratuitamente en la página Web de Microsoft. O, si tiene Windows 98, aprovechar que esta versión ya lo trae incorporado. La imagen anterior muestra una de las páginas del diálogo de propiedades del servidor, que puede ejecutarse desde el Panel de Control, o desde el icono en la Bandeja de Iconos, si se ha cargado este programa.

Si va a desarrollar extensiones ISAPI/NSAPI en vez de CGI, se encontrará con que necesitará descargar la DLL programada cada vez que quiera volver a compilar el proyecto, si es que la ha probado con un navegador. Lo que sucede es que el servidor de Internet deja a la DLL cargada en una caché, de forma tal que la próxima petición a esa URL pueda tener una respuesta más rápida. Mientras la DLL esté cargada, el fichero en disco estará bloqueado, y no podrá ser sustituido. Este comportamiento es positivo una vez que nuestra aplicación esté en explotación, pero es un engorro mientras la desarrollamos. Una solución es detener cada vez el servidor después de cada cambio, lo cual es factible solamente cuando lo estamos probando en modo local. La mejor idea es desactivar temporalmente la caché de extensiones en el servidor Web, modificando la siguiente clave del registro de Windows:

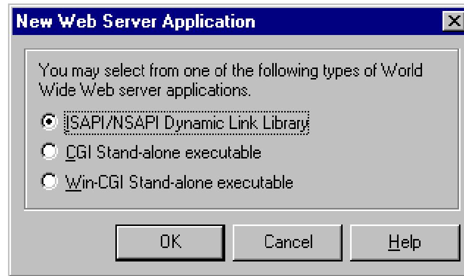
```
[HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\W3Svc\
   Parameters]
CacheExtensions="00"
```

El ejemplo mostrado en este capítulo ha sido desarrollado y depurado utilizando Personal Web Server de Microsoft para Windows 95.

Módulos Web

Podemos crear aplicaciones CGI fácilmente con Delphi: basta con desarrollar una aplicación de tipo *consola* y manejar directamente la entrada estándar y la salida estándar. Recuerde que 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. Desde la época de Delphi 2, podíamos generar este estilo de aplicaciones. 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 sencilla, como puede ser implementar un contador de accesos, no sería demasiado asumir todas estas tareas, pero las aplicaciones más complejas pueden írsenos 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
  HTTPApp,
  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 crea código para una DLL:

```
library Project1;
uses
  HTTPApp,
  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 en la cláusula **exports**? Sí, se trata de las funciones que el servidor Web espera que tenga nuestra extensión, y las llamará para la carga de la DLL, la generación de documentos dinámicos y para descargar finalmente la DLL cuando haga falta. Delphi implementa estas tres funciones por nosotros.

Ahora seguramente el lector saltará de alegría: le voy a contar cómo corregir un *bug* del asistente de generación de aplicaciones ISAPI/NSAPI de Borland. Resulta que existe una variable global en la unidad *System* que es utilizada por el administrador de memoria dinámica de Delphi para saber si puede utilizar o no el algoritmo más rápido, pero no reentrante, que asume la existencia de un solo subproceso dentro de la aplicación. Esta variable se llama *IsMultiThread*, y tenemos que añadir la siguiente inicialización en el fichero de proyecto:

```
IsMultiThread := True;
```

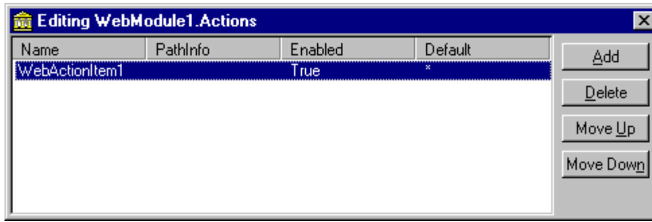
> Delphi 4 ha corregido el *bug*, inicializando *IsMultiThread* en las unidades de soporte de las extensiones ISAPI/NSAPI.

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 desciende 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.

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. 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*. Y 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 esta propiedad:



Cada objeto *WebActionItem* que se añade a esta colección posee las siguientes propiedades importantes:

Propiedad	Significado
PathInfo	El nombre del camino especificado en la URL, después del nombre de la aplicación
MethodType	El tipo de petición ante la cual reacciona el elemento
Default	Si es *True*, se dispara cuando no se encuentre una acción más apropiada
Enabled	Si está activo o no

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:

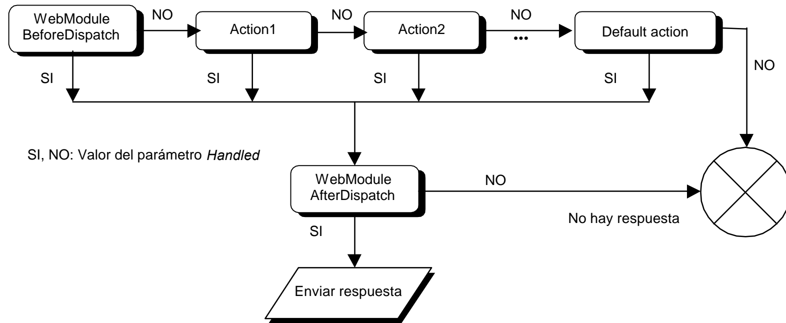
```
type
   THTTPMethodEvent = procedure(Sender: TObject;
      Request: TWebRequest; Response: TWebResponse;
      var Handled: Boolean) of object;
```

Toda la información acerca de la petición realizada por el cliente viene en el parámetro *Request* del evento, que analizaremos en la 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*. De este modo, 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*; 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.

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 para otros tipos de métodos menos comunes, 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`
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 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 HTML a la propiedad *Content* del parámetro *Response* del evento en cuestión. 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 empezar, veamos el componente *TPageProducer*, cuyas propiedades relevantes son las siguientes:

Propiedad	Significado
Dispatcher	El módulo en que se encuentra, o el componente *TWebDispatcher* al que se asocia
HTMLDoc	Lista de cadena que contiene texto HTML
HTMLFile	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. En cualquiera de estos dos casos, el ejemplo de respuesta a evento anterior se escribiría de este otro modo:

```
procedure TWebModule1.WebModule1WebActionItem1Action(
   Sender: TObject; Request: TWebRequest; Response: TWebResponse;
   var Handled: Boolean);
begin
   Response.Content := PageProducer1.Content;
end;
```

No obstante, si nos limitamos a lo que hemos descrito, 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;
```

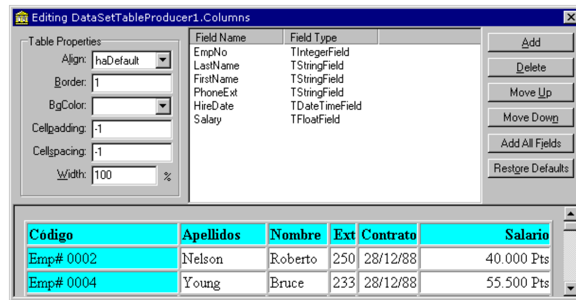
Generadores de tablas

Un caso particular de generadores de contenido son los generadores de tablas que trae Delphi incorporados: *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 Delphi. *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
DataSet	El conjunto de datos asociado
MaxRows	Número máximo de filas de datos generadas
Caption, CaptionAlignment	Permite añadir un título a la tabla
Header, Footer	Para colocar texto antes y después de la tabla
Columns, RowAttributes, ColumnAttributes	Formato de la tabla

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.

Mantenimiento de la información de estado

El problema principal de los servidores Web es su corta memoria. Usted le pide a uno de ellos: dame, por favor, la lista de los diez discos más vendidos, y el servidor le responderá con mil amores. Pero si se le 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.

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:

- 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".
- El usuario debe disponer a su vez de este identificador, lo que implica que el servidor debe pensar en algún de 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 (uh-oh)</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.

Existen muchas otras formas de mantener la información de estado relacionada con un cliente. Una de estas técnicas son las "famosas" *cookies*, que son plenamente soportadas por Delphi mediante propiedades y métodos de las clases *TWebRequest* y *TWebResponse*. 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, por lo que no entraré en detalles.

Un ejemplo: búsqueda de productos

Para poner en orden toda la información anterior, desarrollaremos un pequeño ejemplo de extensión de servidor Web. Con este servidor podremos buscar en una base de datos de productos de software mediante una lista de palabras reservadas. A continuación incluyo un fichero *script* para InterBase con la definición de la base de datos y las tablas que utilizaremos:

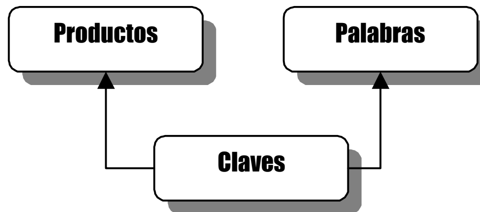
```
create database "C:\Data\Productos.GDB"
user "SYSDBA" password "masterkey" page_size 1024;
set autoddl on;

/*** Definición de las tablas ********************************/

create table Productos (
       Codigo         int not null,
       Descripcion    varchar(45) not null,
       Categoria      varchar(30) not null,
       UCat           varchar(30) not null,
       Comentario     blob sub_type 1,

       primary key    (Codigo),
       unique         (Descripcion)
);
create index CategoriaProducto on Productos(UCat);

create table Palabras (
       Codigo         int not null,
       Palabra        varchar(30) not null,

       primary key    (Codigo),
       unique         (Palabra)
);

create table Claves (
       Producto       int not null,
       Palabra        int not null,
```

```
        primary key    (Producto, Palabra),
        foreign key    (Producto) references Productos(Codigo)
                       on update cascade on delete cascade,
        foreign key    (Palabra) references Palabras(Codigo)
                       on update cascade on delete cascade
);

/*** Triggers y procedimientos almacenados *******************/

set term ^;

create generator GenProductos^

create procedure CodigoProducto returns (Codigo int) as
begin
        Codigo = gen_id(GenProductos, 1);
end^

create trigger NuevoProducto for Productos
        active before insert as
begin
        new.UCat = upper(new.Categoria);
end^

create procedure BorrarPalabras(Producto int) as
begin
        delete from Claves
        where Producto = :Producto;
end^

/* Palabras y asociaciones */

create generator GenPalabras^

create procedure Asociar(Producto int, Palabra varchar(30)) as
declare variable CodPal int;
begin
        Palabra = upper(Palabra);
        select  Codigo
        from    Palabras
        where   Palabra = :Palabra
        into    :CodPal;
        if (CodPal is null) then
        begin
                CodPal = gen_id(GenPalabras, 1);
                insert into Palabras
                values (:CodPal, :Palabra);
        end
        insert into Claves(Producto, Palabra)
        values (:Producto, :CodPal);
end^

set term ;^
```

La tabla fundamental es la de productos, que asocia a cada producto un código numérico, una descripción, comentarios y una categoría. La categoría es una cadena de

la cual se guardan dos versiones: una en mayúsculas (*UCat*) y la otra utilizando los mismos caracteres introducidos por el usuario. Este es un truco de InterBase para acelerar las búsquedas insensibles a mayúsculas y minúsculas: observe que el índice se ha creado sobre el campo *UCat*.

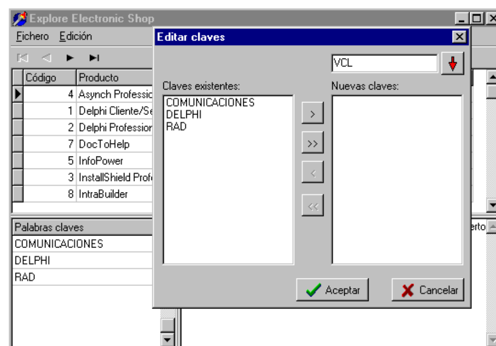
Paralelamente, existe una tabla con todas las palabras claves que están asociadas a los productos. Esta tabla se encarga de asociar un código numérico a cada una de las palabras. Finalmente, la tabla *Claves* almacena pares formados por un código de producto y un código de palabra. En general, cada producto puede estar asociado a varias palabras, y cada palabra puede estar vinculada con varios productos.



No voy a incluir la búsqueda por categorías, pues es una extensión sencilla al programa que vamos a desarrollar.

El motor de búsquedas

El paso siguiente consiste en crear una pequeña aplicación para poblar la base de datos de productos. La aplicación debe permitir introducir productos y sus descripciones, y asociarle posteriormente palabras claves para la búsqueda. Como es una aplicación muy sencilla, he decidido no incluir el código de la misma en el texto de libro, pero el usuario puede encontrar las fuentes del programa, profusamente comentadas, en el CD-ROM adjunto.



Como parte importante de la aplicación, sin embargo, se debe incluir la posibilidad de que el usuario pueda comprobar las asociaciones realizadas entre productos y palabras claves mediante una ventana de búsqueda. La parte fundamental de este mecanismo de búsqueda consiste en una función, de nombre *GenerateQuery*, que partiendo de una cadena con la lista de palabras a buscar separadas por espacio genera una instrucción **select**; esta consulta se copia dentro de un componente *Query*, que al abrirse muestra sus resultados en una rejilla de datos.

Esta función será utilizada tanto por la aplicación Web como por la utilidad que sirve para poblar la base de datos. Por lo tanto, se ha aislado, junto con la función auxiliar *Tokenize*, dentro de una unidad denominada *SearchEngine*, que deberemos incluir en el proyecto Web más adelante. Este es el código de ambas funciones:

```
procedure Tokenize(AText: string; AList: TStrings);
var
   I: Integer;
   S: string;
begin
   AText := AnsiUpperCase(Trim(AText));
   while AText <> '' do
   begin
      if AText[1] = '"' then
      begin
         Delete(AText, 1, 1);
         I := Pos('"', AText);
      end
      else
         I := Pos(' ', AText);
      if I = 0 then
      begin
         S := AText;
         AText := '';
      end
      else
      begin
         S := Copy(AText, 1, I - 1);
         Delete(AText, 1, I);
         AText := TrimLeft(AText);
      end;
```

```
      if S <> '' then AList.Add(S);
    end;
  end;

  procedure GenerateQuery(const AText: string; AQuery: TStrings);
  var
    I: Integer;
    Prefix: string;
    AList: TStrings;
  begin
    AList := TStringList.Create;
    try
      Tokenize(AText, AList);
      AQuery.Clear;
      AQuery.Add('SELECT * FROM PRODUCTOS');
      Prefix := 'WHERE ';
      for I := 0 to AList.Count - 1 do
      begin
        AQuery.Add(Prefix + 'CODIGO IN ');
        AQuery.Add('      (SELECT PRODUCTO');
        AQuery.Add('       FROM  CLAVES, PALABRAS');
        AQuery.Add('       WHERE CLAVES.PALABRA=PALABRAS.CODIGO');
        AQuery1Add('       AND PALABRAS.PALABRA = ' +
          QuotedStr(UpperCase(Trim(AList[I]))) + ')');
        Prefix := '  AND ';
    finally
      AList.Free;
    end;
  end;
```

Para comprender cómo funciona este procedimiento, supongamos que el usuario teclea el siguiente texto en el control de palabras a buscar:

```
"bases de datos" rad vcl
```

El primer paso que realiza *GenerateQuery* es descomponer la cadena anterior en las palabras o expresiones a buscar. El resultado intermedio es la siguiente lista de cadenas:

```
1.  bases de datos
2.  rad
3.  vcl
```

A partir de la lista anterior, se va creando una instrucción **select** en la que se añade una subconsulta sobre las tablas de claves y palabras por cada palabra tecleada por el usuario. La consulta generada es la siguiente:

```
select *
from   Productos
where  Codigo in
       (select Producto
        from   Claves, Palabras
        where  Claves.Palabra = Palabras.Codigo
          and  Palabras.Palabra = 'BASES DE DATOS')
  and  Codigo in
```

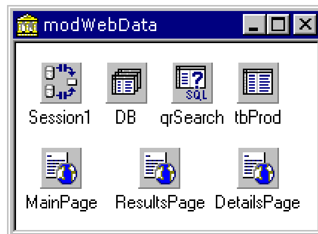
```
           (select Producto
            from   Claves, Palabras
            where  Claves.Palabra = Palabras.Codigo
              and  Palabras.Palabra = 'RAD')
      and  Codigo in
           (select Producto
            from   Claves, Palabras
            where  Claves.Palabra = Palabras.Codigo
              and  Palabras.Palabra = 'VCL')
```

Seguramente que el lector estará pensando en mejoras a este generador. Por ejemplo, incluir la búsqueda por categorías, permitir búsquedas parciales por prefijos de palabras, incluir otros operadores lógicos además de la conjunción.

Creando la extensión Web

Comenzaremos en este momento la creación de la extensión de servidor. Web. Ejecute el comando *File|New* para desplegar el Depósito de Objetos, y seleccione el icono *Web Server Application*. Luego elija una aplicación ISAPI/NSAPI ó CGI, según prefiera. Recuerde modificar el texto del proyecto, si elige ISAPI/NSAPI, como hemos explicado anteriormente, para indicar que esta aplicación lanzará múltiples subprocesos.

Cambie el nombre del módulo de datos a *modWebData*, y coloque los objetos que aparecen a continuación:

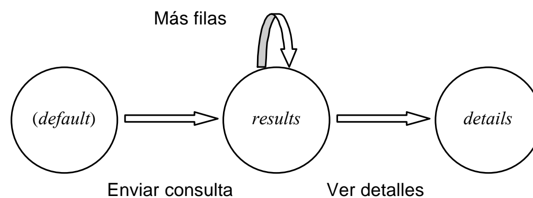


El primer objeto a situar es *Session1*, para que cada hilo concurrente del servidor corresponda a un usuario diferente y no tengamos problemas de sincronización. Debemos asignar *True* a su propiedad *AutoSessionName*, para evitar conflictos con las sesiones concurrentes; recuerde que cada cliente conectado va a crear una nueva instancia del módulo de datos. Después traemos la base de datos *BD*, la enganchamos al objeto de sesión mediante su propiedad *SessionName*, y configuramos sus parámetros de modo que podamos conectarnos a la base de datos, en la ubicación en que se encuentre en el servidor de Internet. Finalmente, he conectado una consulta, *qrSearch*, y una tabla, *tbProd*, a la base de datos, ambas inactivas en tiempo de diseño. La consulta tiene su instrucción SQL sin asignar, pues se generará mediante la función *GenerateQuery* que hemos definido antes. Por su parte, *tbProd* debe conectarse a la

tabla de productos de la base de datos, y se utilizará en la generación de la página de detalles de un producto.

El flujo de la aplicación será el siguiente:

1. El usuario se conecta a la página principal (acción por omisión). En esta página se le pedirá que teclee la cadena de búsqueda. Cuando pulse el botón de envío, se intentará activar la acción *results*, para ver los resultados de la búsqueda.
2. La página generada con la acción *results* mostrará la información de cabecera de los productos dentro de una tabla HTML. Limitaremos el número de filas por página, de modo que en determinados casos debe aparecer un botón al finalizar la página para recuperar más filas.
3. En la página de resultados, cada fila de productos tiene un enlace en la columna del nombre del producto para que ampliemos la información sobre el producto seleccionado. Esta información se mostrará en una tercera página, correspondiente a la acción *details*.



En concordancia con el flujo de páginas en la aplicación definimos tres acciones en el módulo, como puede verse en la siguiente imagen:

Editing modWebData.Actions

Name	PathInfo	Enabled	Default
Main		True	*
Results	/results	True	
Details	/details	True	

Ahora definiremos constantes y funciones auxiliares. Cuando generemos el texto HTML, tendremos que hacer referencias a otras páginas situadas dentro del servidor. Aunque podemos utilizar una URL relativa, prefiero siempre una URL completa. Pero esa dirección depende de si estamos probando la extensión en nuestro servidor local o en el servidor real de Internet. Por eso, he añadido una declaración de cadena de recursos, para indicar en qué dominio se encuentra nuestro programa:

```
resourcestring
  SDominio = 'http://ian.marteens.com';
```

La siguiente función utiliza la cadena definida para generar una URL:

```
function TmodWebData.GenURL(const APath: string): string;
begin
   Result := SDominio + '/scripts/ShopSrvr.dll/' + APath;
end;
```

También he apartado en métodos auxiliares la generación de cadenas HTML que utilizaremos con frecuencia. Por ejemplo, el encabezamiento de formularios:

```
function TmodWebData.GenFormHeader(const APath: string): string;
begin
   Result := '<FORM ACTION="' + GenURL(APath) + '" METHOD=GET>';
end;
```

Los tres objetos de clase *TPageProducer* serán analizados más adelante.

Generando la tabla de resultados

Seguimos definiendo métodos auxiliares en el módulo. La función *GenResults*, que mostraremos a continuación, genera el fragmento de código HTML con la tabla de productos encontrados a partir de la consulta del usuario. Esta función tiene un prerrequisito importante: la consulta *qrSearch* debe estar posicionada en el primer registro que queremos que aparezca en la tabla. El motivo es fácil de entender: una consulta puede generar muchos productos coincidentes, por lo cual se tratará de limitar la cantidad de productos por página, mediante la constante *MAXLINKS*, definida globalmente en el módulo:

```
const
   MAXLINKS = 5;
```

Uno de los parámetros que pasaremos en la URL será el último código de producto listado. Evidentemente, cuando el usuario realiza su consulta todavía no ha recibido ningún listado, por lo que este parámetro tendrá un valor por omisión: *-1*. Si un listado de productos sobrepasa el máximo de filas, colocaremos un botón de enlace al final del mismo, que ejecutará nuevamente la consulta, pero pasándole un parámetro *LastCode* que indique cual es el último código de producto incluido en la tabla. El siguiente listado muestra los detalles de la composición del documento HTML:

```
function TmodWebData.GenResults: string;
var
   I, LastCode: Integer;
begin
   if qrSearch.EOF then
   begin
      Result := '<HTML><BODY><P>' +
         'No se encontró ningún producto.</P></BODY></HTML>';
```

```
      Exit;
   end;
   Result := '<Table Width=100% Border=1>';
   AppendStr(Result, '<TR><TH BgColor="Blue">CODIGO</TH>' +
      '<TH BgColor="Blue">PRODUCTO</TH>' +
      '<TH  BgColor="Blue">CATEGORIA</TH></TR>');
   for I:= 1 to MAXLINKS do
   begin
      AppendStr(Result,
         '<TR><TD>' + IntToStr(qrSearch['CODIGO']) + '</TD>' +
         '<TD><A HREF="' + GenURL('details') +
         '?Codigo=' + IntToStr(qrSearch['CODIGO']) + '">' +
         qrSearch['DESCRIPCION'] + '</A></TD>' +
         '<TD>' + qrSearch['CATEGORIA'] + '</TD></TR>');
      qrSearch.Next;
      if qrSearch.EOF then
      begin
         LastCode := -1;
         Break;
      end;
      LastCode := qrSearch['CODIGO'];
   end;
   AppendStr(Result, '</Table>');
   if LastCode <> -1 then
   begin
      AppendStr(Result, GenFormHeader('results'));
      AppendStr(Result, '<INPUT TYPE="HIDDEN" NAME="LASTCODE" ' +
         'VALUE="' + IntToStr(LastCode) + '">');
      AppendStr(Result, '<INPUT TYPE="SUBMIT" ' +
         'VALUE="Más productos"></FORM>');
   end;
end;
```

Documentos HTML y sustitución de etiquetas

Implementaremos ahora la sustitución de etiquetas en los generadores de contenidos, asociando respuestas a sus eventos *OnHTMLTag*. Comenzaremos por el componente *MainPage*, que contiene el texto de la página principal en su propiedad *HTMLDoc*:

```
<HTML><BODY>
<H2>¡Bienvenido al Centro de Información de Productos!</H2>
<HR>
<#FORMHEADER>
<P>Teclee hasta 10 palabras para realizar la búsqueda:
<INPUT NAME="Keywords"></P>
<INPUT TYPE="HIDDEN" NAME="LASTCODE" VALUE="-1">
<INPUT TYPE="SUBMIT" VALUE="Buscar">
</FORM>
</BODY></HTML>
```

La página tiene una sola etiqueta transparente, `<#FORMHEADER>`, cuya sustitución es simple:

```
procedure TmodWebData.MainPageHTMLTag(Sender: TObject; Tag: TTag;
   const TagString: string; TagParams: TStrings;
   var ReplaceText: string);
begin
   if TagString = 'FORMHEADER' then
      ReplaceText := GenFormHeader('results');
end;
```

Los resultados de la búsqueda se mostrarán en la siguiente página, contenida en el componente *ResultsPage*:

```
<HTML>
<BODY>
<H2>Resultados de la búsqueda</H2>
<HR>
<#RESULTS>
</BODY>
</HTML>
```

Bien, la sustitución de la etiqueta `<#RESULTS>` es complicada, pero ya hemos recorrido más de la mitad del camino en la sección anterior:

```
procedure TmodWebData.ResultsPageHTMLTag(Sender: TObject; Tag: TTag;
   const TagString: string; TagParams: TStrings;
   var ReplaceText: string);
begin
   if TagString = 'RESULTS' then
      ReplaceText := GenResults;
end;
```

Finalmente, así se verá la información detallada acerca de un producto, mediante el componente *DetailsPage*:

```
<HTML><BODY>
<P><H3>Nombre del producto: </H3><#PRODUCTO></P>
<P><H4>Categoría: </H4><#CATEGORIA></P>
<H4>Comentarios:</H4>
<#COMENTARIOS>
</BODY></HTML>
```

Y este será el método que sustituirá las etiquetas correspondientes:

```
procedure TmodWebData.DetailsPageHTMLTag(Sender: TObject; Tag: TTag;
   const TagString: string; TagParams: TStrings;
   var ReplaceText: string);
var
   AList: TStringList;
   I: Integer;
begin
   if TagString = 'PRODUCTO' then
      ReplaceText := tbProdDescripcion.Value
   else if TagString = 'CATEGORIA' then
      ReplaceText := tbProdCategoria.Value
```

```
      else if TagString = 'COMENTARIOS' then
      begin
         AList := TStringList.Create;
         try
            AList.Assign(tbProdComentario);
            ReplaceText := '<BLOCKQUOTE>';
            for I := 0 to AList.Count - 1 do
               AppendStr(ReplaceText, '<P>' + AList[I] + '</P>');
            AppendStr(ReplaceText, '</BLOCKQUOTE>');
         finally
            AList.Free;
         end;
      end;
   end;
```

> He utilizado un componente *TPageProducer* en la generación de la página de detalles por compatibilidad con Delphi 3. Delphi 4 introduce el componente *TDataSetPageProducer*, similar a *TPageProducer*, pero que se asocia a un conjunto de datos, y puede sustituir automáticamente las etiquetas que correspondan a nombres de columnas con el valor de las mismas en la fila actual del conjunto de datos.

Respondiendo a las acciones

Por último, aquí están los métodos de respuesta a las acciones, que sirven para integrar todos los métodos desarrollados anteriormente:

```
procedure TmodWebData.modWebDataStartAction(Sender: TObject;
   Request: TWebRequest; Response: TWebResponse;
   var Handled: Boolean);
begin
   Response.Content := MainPage.Content;
end;

procedure TmodWebData.modWebDataResultsAction(Sender: TObject;
   Request: TWebRequest; Response: TWebResponse;
   var Handled: Boolean);
var
   LastCode: Integer;
begin
   Request.ExtractContentFields(Request.QueryFields);
   LastCode := StrToInt(Request.QueryFields.Values['LASTCODE']);
   GenerateQuery(Request.QueryFields.Values['Keywords'],
      qrSearch.SQL);
   qrSearch.Open;
   try
      if LastCode > 0 then
         qrSearch.Locate('CODIGO', LastCode, []);
      Response.Content := ResultsPage.Content;
   finally
      qrSearch.Close;
   end;
end;
```

```
procedure TmodWebData.modWebDataDetailsAction(Sender: TObject;
   Request: TWebRequest; Response: TWebResponse;
   var Handled: Boolean);
begin
   Request.ExtractContentFields(Request.QueryFields);
   tbProd.Open;
   try
      tbProd.Locate('CODIGO',
         Request.QueryFields.Values['Codigo'], []);
      Response.Content := DetailsPage.Content;
   finally
      tbProd.Close;
   end;
end;
```

Capítulo

38

Conjuntos de datos clientes

SER ORGANIZADOS TIENE SU RECOMPENSA, Y NO HAY que esperar a morirse e ir al Cielo para disfrutar de ella. Al parecer, los programadores de Borland (perdón, Inprise) hicieron sus deberes sobresalientemente al diseñar la biblioteca dinámica *dbclient.dll*, y el tipo de datos *TClientDataSet*, que se comunica con ella. Al aislar el código de manejo de bases de datos en memoria en esta biblioteca, han podido mejorar ostensiblemente las posibilidades de la misma al desarrollar la versión 4 de Delphi.

Los *conjuntos de datos clientes*, o *client datasets*, son objetos que pertenecen a una clase derivada de *TDataSet*: la clase *TClientDataSet*. Estos objetos almacenan sus datos en memoria RAM local, pero la característica que los hizo famosos en Delphi 3 es que pueden leer estos datos desde un servidor de datos remoto mediante automatización DCOM. En la edición anterior de este libro, los conjuntos de datos clientes se explicaban junto a las técnicas de comunicación con bases de datos remotas mediante Midas. Pero, como he explicado antes, hay tantas nuevas características que estos componentes se han ganado un capítulo independiente.

Aquí nos concentraremos en las características de *TClientDataSet* que nos permiten gestionar bases de datos en memoria, utilizando como origen de datos a ficheros "planos" del sistema operativo. ¿Qué haría usted si tuviera que implementar una aplicación de bases de datos, pero cuyo volumen de información esté en el orden de 1.000 a 10.000 registros? En vez de utilizar Paradox, dBase o Access, que requieren la presencia de un voluminoso y complicado motor de datos, puede elegir los conjuntos de datos en memoria. Veremos que en Delphi 4 estos conjuntos soportan características avanzadas como tablas anidadas, valores agregados mantenidos, índices dinámicos, etc. Para el siguiente capítulo dejaremos las propiedades, métodos y eventos que hacen posible la comunicación con servidores Midas.

Creación de conjuntos de datos

El componente *TClientDataSet* está situado en la página *Midas* de la Paleta de Componentes. Usted trae uno de ellos a un formulario y se pregunta: ¿cuál es el esquema

de los datos almacenados por el componente? Si el origen de nuestros datos es un servidor remoto, la definición de campos, índices y restricciones del conjunto de datos se leen desde el servidor, por lo cual no tenemos que preocuparnos en ese sentido.

Supongamos ahora que los datos deban extraerse de un fichero local. Este fichero debe haber sido creado por otro conjunto de datos *TClientDataSet*, por lo cual en algún momento tenemos que definir el esquema del conjunto y crearlo, para evitar un círculo vicioso. Para crear un conjunto de datos en memoria debemos definir los campos e índices que deseemos, y aplicar posteriormente el método *CreateDataSet*:

```
with ClientDataSet1.FieldDefs do
begin
   Clear;
   Add('Nombre', ftString, 30, True);
   Add('Apellidos', ftString, 30, True);
   Add('Direccion', ftString, 30, True);
   Add('Telefono', ftString, 10, True);
   Add('FAX', ftString, 10, True);
   Add('Mail', ftString, 80, True);
end;
ClientDataSet1.CreateDataSet;
```

Al igual que sucede con las tablas, en Delphi 4 existe la alternativa de llenar la propiedad *FieldDefs* en tiempo de diseño, con lo cual la propiedad *StoredDefs* se vuelve verdadera. Cuando deseemos crear el conjunto de datos en tiempo de ejecución, solamente tenemos que llamar a *CreateDataSet*. Si no existen definiciones en *FieldDefs*, pero sí se han definido campos persistentes, se utilizan estos últimos para crear el conjunto de datos.

También se puede crear el conjunto de datos junto a sus índices, utilizando la propiedad *IndexDefs* en tiempo de diseño o de ejecución:

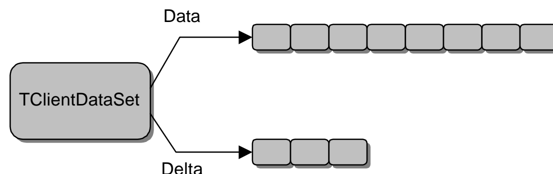
```
with ClientDataSet1.FieldDefs do
begin
   Clear;
   Add('Nombre', ftString, 30, True);
   Add('Apellidos', ftString, 30, True);
   Add('Direccion', ftString, 30, False);
   Add('Telefono', ftInteger, 0, True);
   Add('FAX', ftInteger, 0, False);
   Add('Mail', ftString, 80, False);
end;
with ClientDataSet1.IndexDefs do
begin
   Clear;
   Add('xNombre', 'Nombre', [ixCaseInsensitive]);
   Add('xApellidos', 'Apellidos', [ixCaseInsensitive]);
   Add('xTelefono', 'Telefono', []);
end;
ClientDataSet1.CreateDataSet;
```

Más adelante veremos todos los tipos de índices que admiten los conjuntos de datos clientes.

Por último, también podemos crear el conjunto de datos cliente en tiempo de diseño, pulsando el botón derecho del ratón sobre el componente y ejecutando el comando del menú local *Create DataSet*.

Cómo el TClientDataSet consiguió sus datos...

Un componente *TClientDataSet* siempre almacena sus datos en dos vectores situados en la memoria del ordenador donde se ejecuta la aplicación. 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*.



Como mencionaba en la introducción, hay varias formas de obtener estos datos:

- A partir de ficheros del sistema operativo.
- Copiando los datos almacenados en otro conjunto de datos.
- Mediante una conexión a una interfaz *IProvider* suministrada por un servidor de datos remoto.

Para obtener datos a partir de un fichero, se utiliza el método *LoadFromFile*. El fichero debe haber sido creado por un conjunto de datos (quizás éste mismo) mediante una llamada al método *SaveToFile*, que también está disponible en el menú local del componente. Los ficheros, cuya extensión por omisión es *cds*, no se guardan en formato texto, sino en un formato que es propio de estos componentes. Podemos leer un conjunto de datos cliente a partir de un fichero aunque el componente no contenga definiciones de campos, ya sea en *FieldDefs* o en *Fields*.

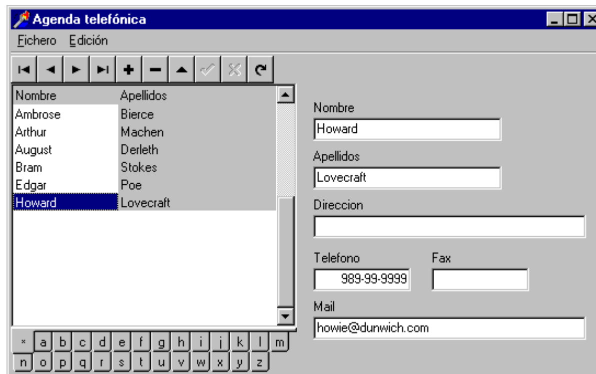
Sin embargo, la forma más común de trabajar con ficheros es asignar un nombre de fichero en la propiedad *FileName* del componente. Si el fichero existe en el momento

de la apertura del conjunto de datos, se lee su contenido. En el momento en que se cierra el conjunto de datos, se sobrescribe el contenido del fichero con los datos actuales.

Navegación, búsqueda y selección

Como todo buen conjunto de datos, los *TClientDataSet* permiten las ya conocidas operaciones de navegación y búsqueda: *First*, *Next*, *Last*, *Prior*, *Locate*, rangos, filtros, etc. Permiten también especificar un criterio de ordenación mediante las propiedades *IndexName* e *IndexFieldNames*. En esta última propiedad, al igual que sucede con las tablas SQL, podemos indicar cualquier combinación de columnas, pues el conjunto de datos cliente puede generar dinámicamente un índice de ser necesario.

La siguiente imagen muestra una sencilla aplicación para el mantenimiento de una libreta de teléfonos. La aplicación completa viene incluida en el CD-ROM del libro:



En todo momento, el conjunto de datos se encuentra ordenado por el campo *Nombre* o por el campo *Apellidos*. El orden se puede cambiar pulsando el ratón sobre el título de la columna en la rejilla:

```
procedure TwndMain.DBGrid1TitleClick(Column: TColumn);
begin
   ClientDataSet1.CancelRange;
   ClientDataSet1.IndexFieldNames := Column.FieldName;
   TabControl1.TabIndex := 0;
end;
```

Debajo de la rejilla he situado un componente *TTabControl*, con todas las letras del alfabeto. En respuesta al evento *OnChange* de este control, se ejecuta este método:

```
procedure TwndMain.TabControl1Change(Sender: TObject);
var
   Tab: string;
```

```
begin
   Tab := TabControl1.Tabs[TabControl1.TabIndex];
   if Tab = '*' then
      ClientDataSet1.CancelRange
   else
      ClientDataSet1.SetRange([Tab], [Tab + 'zzz']);
end;
```

Por supuesto, los índices que se han definido sobre este conjunto de datos no hacen distinción entre mayúsculas y minúsculas.

Hay una simpática peculiaridad de estos conjuntos de datos que concierne a la navegación: la propiedad *RecNo* no solamente nos indica en qué número de registro estamos situados, sino que además permite que nos movamos a un registro determinado asignando su número de posición en la misma:

```
procedure TwndMain.IrAExecute(Sender: TObject);
var
   S: string;
begin
   if InputQuery('Ir a posición', 'Número de registro:', S) then
      ClientDataSet1.RecNo := StrToInt(S);
end;
```

Filtros

Una agradable sorpresa: las expresiones de filtros de los conjuntos de datos clientes soportan montones de características no permitidas por los conjuntos de datos del BDE. En primer lugar, ya podemos saber si un campo es nulo o no utilizando la misma sintaxis que en SQL:

```
Direccion is null
```

Podemos utilizar 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)
```

Edición de datos

Como hemos explicado, los datos originales de un *TClientDataSet* se almacenan en su propiedad *Data*, mientras que los cambios realizados van a parar a la propiedad *Delta*. Cada vez que un registro se modifica, se guardan los cambios en esta última propiedad, pero sin descartar o mezclar con posibles cambios anteriores. Esto permite que el programador ofrezca al usuario comandos sofisticados para deshacer los cambios realizados, en comparación con las posibilidades que ofrece un conjunto de datos del BDE.

En primer lugar, tenemos el método *UndoLastChange*:

```
procedure TClientDataSet.UndoLastChange(FollowChanges: Boolean);
```

Este método deshace el último cambio realizado sobre el conjunto de datos. Cuando indicamos *True* en su parámetro, el cursor del conjunto de datos se desplaza a la posición donde ocurrió la acción. Con *UndoLastChange* podemos implementar fácilmente el típico comando *Deshacer* de los procesadores de texto. Claro, nos interesa también saber si hay cambios, para activar o desactivar el comando de menú, y en esto nos ayuda la siguiente propiedad:

```
property ChangeCount: Integer;
```

Tenemos también la propiedad *SavePoint*, de tipo entero. El valor de esta propiedad indica una posición dentro del vector de modificaciones, y sirve para establecer un punto de control dentro del mismo. Por ejemplo, supongamos que usted desea imitar una transacción sobre un conjunto de datos cliente para determinada operación larga. Esta pudiera ser una solución sencilla:

```
procedure TwndMain.OperacionLarga;
var
   SP: Integer;
begin
   SP := ClientDataSet1.SavePoint;
   try
      // Aquí se realizan las distintas modificaciones …
   except
      ClientDataSet1.SavePoint := SP;
```

```
      raise;
    end;
  end;
```

A diferencia de lo que sucede con una transacción verdadera, podemos anidar varios puntos de verificación.

Cuando se guardan los datos de un *TClientDataSet* en un fichero, se almacenan por separado los valores de las propiedades *Delta* y *Data*. De este modo, cuando se reanuda una sesión de edición que ha sido guardada, el usuario sigue teniendo a su disposición todos los comandos de control de cambios. Sin embargo, puede que nos interese mezclar definitivamente estas dos propiedades, para lo cual debemos utilizar el método siguiente:

```
procedure TClientDataSet.MergeChangeLog;
```

Como resultado, el espacio necesario para almacenar el conjunto de datos disminuye, pero las modificaciones realizadas hasta el momento se hacen irreversibles. Si le interesa que todos los cambios se guarden directamente en *Data*, sin utilizar el *log*, debe asignar *False* a la propiedad *LogChanges*.

Estos otros dos métodos también permiten deshacer cambios, pero de forma radical:

```
procedure TClientDataSet.CancelUpdates;
procedure TClientDataSet.RevertRecord;
```

Como puede ver, estos métodos son análogos a los que se utilizan en las actualizaciones en caché. Recuerde que cancelar todos los cambios significa, en este contexto, vaciar la propiedad *Delta*, y que si usted no ha mezclado los cambios mediante *MergeChangeLog* en ningún momento, se encontrará de repente con un conjunto de datos vacío entre sus manos.

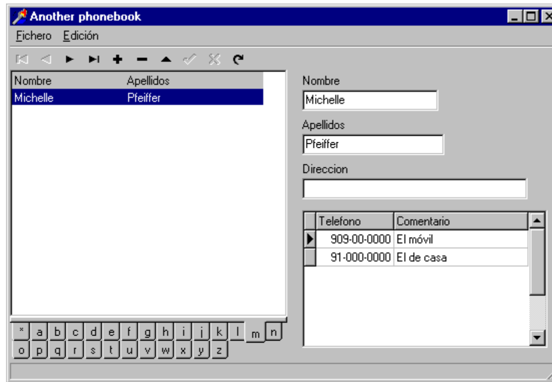
Conjuntos de datos anidados

Los conjuntos de datos clientes de la versión 4 de Delphi permiten definir campos de tipo *ftDataSet*, que contienen tablas anidadas. Este recurso puede servir como alternativa al uso de tablas conectadas en relaciones *master/detail*. Las principales ventajas son:

- Todos los datos se almacenan en el mismo fichero.
- Las tablas anidadas funcionan a mayor velocidad que las relaciones *master/detail*.
- Las técnicas de deshacer cambios funcionan en sincronía para los datos de cabecera y los datos anidados.

- Como las transacciones en bases de datos tradicionales pueden simularse mediante el registro de cambios, podemos realizar transacciones atómicas que incluyan a los datos de cabecera y los de detalles, simultáneamente.

La siguiente imagen muestra una variante de la aplicación de la libreta de teléfonos que utiliza una tabla anidada para representar todos los números de teléfonos que puede poseer cada persona.



Comencemos por definir el conjunto de datos. Traemos un componente de tipo *TClientDataSet* al formulario, lo nombramos *People*, vamos a su propiedad *FieldDefs* y definimos los siguientes campos:

Campo	Tipo	Tamaño	Atributos
Nombre	*ftString*	*30*	*ftRequired*
Apellidos	*ftString*	*30*	*ftRequired*
Direccion	*ftString*	*35*	
Telefonos	*ftDataSet*	*0*	

Este es el formato de los datos de cabecera. Para definir los detalles, seleccionamos la definición del campo *Telefonos*, y editamos su propiedad *ChildDefs*:

Campo	Tipo	Tamaño	Atributos
Telefono	*ftInteger*	*0*	*ftRequired*
Comentario	*ftString*	*20*	

También añadiremos un par de índices persistentes al conjunto de datos. Editamos la propiedad *IndexDefs* de este modo:

Indice	Campos	Atributos
Nombre	*Nombre*	*ixCaseInsensitive*
Apellidos	*Apellidos*	*ixCaseInsensitive*

El próximo paso es crear el conjunto de datos en memoria, para lo cual pulsamos el botón derecho del ratón sobre *People*, y ejecutamos el comando *Create DataSet*. Este comando deja la tabla abierta, asignando *True* a *Active*. Aproveche y cree ahora los campos del conjunto de datos mediante el Editor de Campos. Asigne también a la propiedad *IndexFieldNames* el campo *Nombre*, para que los datos se presenten inicialmente ordenados por el nombre de la persona.

He dejado intencionalmente vacía la propiedad *FileName* del conjunto de datos. Mi propósito es redefinir el método *Loaded* del formulario, para asignar en tiempo de carga el nombre del fichero de trabajo:

```
procedure TwndMain.Loaded;
begin
   inherited Loaded;
   People.FileName := ChangeFileExt(Application.ExeName, '.cds');
end;
```

También es conveniente interceptar el evento *OnClose* del formulario, para mezclar los cambios y los datos originales antes de que el conjunto de datos cliente los guarde en disco:

```
procedure TwndMain.FormClose(Sender: TObject; Action: TCloseAction);
begin
   People.MergeChangeLog;
end;
```

Los datos anidados se guardan de forma eficiente en la propiedad *Data*, es decir, sin repetir la cabecera. Pero cuando se encuentran en la propiedad *Delta*, es inevitable que se repita la cabecera. Por lo tanto, es conveniente mezclar periódicamente los cambios, con el fin de disminuir las necesidades de memoria.

> No intente llamar a *MergeChangeLog* en el evento *BeforeClose* del conjunto de datos, pues al llegar a este punto ya se han guardado los datos en disco, al menos en la versión actual del componente.

Si en este momento mostramos todos los campos de *People* en una rejilla de datos, podemos editar el campo *Telefonos* pulsando el botón que aparece en la celda de la rejilla. Al igual que sucede con las tablas anidadas de Oracle 8, esta acción provoca la aparición de una ventana emergente con otra rejilla. Esta última contiene dos columnas: una para los números de teléfono y otra para los comentarios asociados.

Sin embargo, es más apropiado traer un segundo conjunto de datos cliente al formulario. Esta vez lo llamaremos *Phones*, y para configurarlo bastará asignar a su propiedad *DataSetField* el campo *PeopleTelefonos*. Cree campos persistentes para este conjunto de datos, y asigne a *IndexFieldNames* el campo *Telefono*, para que estos aparezcan de forma ordenada.

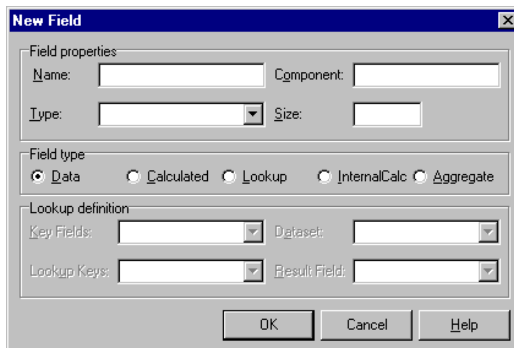
Es muy probable también que desee eliminar todos los teléfonos antes de eliminar a una persona[29], para lo que basta este método:

```
procedure TwndMain.PeopleBeforeDelete(DataSet: TDataSet);
begin
   Phones.First;
   while not Phones.EOF do
      Phones.Delete;
end;
```

Y ya puede definir formatos de visualización, comandos de edición y reglas de integridad al modo tradicional para completar este ejemplo.

Campos calculados internos

Por supuesto, podemos definir campos calculados y campos de búsqueda al utilizar conjuntos de datos clientes. Hay, sin embargo, un nuevo tipo de campo calculado que es típico de esta clase de componentes: los *campos calculados internos*. Para definir este tipo de campo, se utiliza el comando *New field*, del Editor de Campos:



Como podemos apreciar, el grupo de botones de radio que determina el tipo de campo tiene dos nuevas opciones: *InternalCalc* y *Aggregate*. La primera es la que nos interesa por el momento, pues la segunda se estudiará en la siguiente sección. El campo calculado interno se define marcando la opción *InternalCalc*, y especificando el

[29] Le apuesto una cena a que al leer la frase anterior pensó en alguien de su entorno. ¿No? Bueno, yo sí.

nombre y tipo de datos del campo. El valor del campo se calcula durante la respuesta al evento *OnCalcFields* del conjunto de datos.

¿En qué se diferencian los campos calculados internos de los ya conocidos? Simplemente, en que el valor de los campos calculados internos se almacena también junto al conjunto de datos. La principal consecuencia de esta política de almacenamiento es que pueden definirse índices basados en estos campos. Así que tenemos algo parecido a índices por expresiones para los conjuntos de datos clientes.

Una información importante: aunque ambos tipos de campos calculados reciben sus valores en respuesta al evento *OnCalcFields*, en realidad este evento se dispara dos veces, una para los campos calculados internos y otra para los normales. Para diferenciar entre ambas activaciones, debemos preguntar por el valor de la propiedad *State* del conjunto de datos. En un caso, valdrá *dsCalcFields*, mientras que en el otro valdrá *dsInternalCalc*. El código de respuesta al evento puede evitar entonces realizar cálculos innecesarios.

Indices, grupos y valores agregados

Existen tres formas de crear índices en un conjunto de datos cliente:

- 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.

Para entender cómo es posible indicar tales opciones, examinemos los parámetros del método *AddIndex*:

```
procedure TClientDataSet.AddIndex(
   const Name, Fields: string;
   Options: TIndexOptions;
```

```
        const DescFields: string = '';
        const CaseInsFields: string = '';
        const GroupingLevel: Integer = 0 );
```

Los nombres de los campos que formarán parte del índice se pasan en el parámetro *Fields*. Ahora bien, podemos pasar en *DescFields* y en *CaseInsFields* un subconjunto de estos campos, indicando cuáles deben ordenarse en forma descendente y cuáles deben ignorar mayúsculas y minúsculas. Por ejemplo, el siguiente índice ordena ascendentemente por el nombre, teniendo en cuenta las mayúsculas y minúsculas, de modo ascendente por el apellido, pero distinguiendo los tipos de letra, y de modo descendente por el salario:

```
ClientDataSet1.AddIndex('Indice1', 'Nombre;Apellidos;Salario',
   'Salario', 'Apellidos');
```

El tipo *TIndexDef* utiliza las propiedades *DescFields*, *CaseInsFields* y *GroupingLevel* para obtener el mismo efecto que los parámetros correspondientes de *AddIndex*.

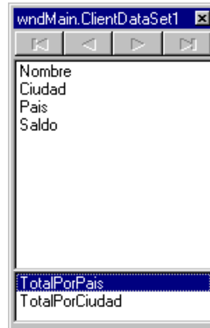
¿Y qué hay acerca del nivel de agrupamiento? Este valor indica al índice que dispare ciertos eventos internos cuando alcance el final de un grupo de valores repetidos. Tomemos como ejemplo el conjunto de datos que mostramos en la siguiente imagen:

Pais	Ciudad	Nombre	Saldo	Total por pais	Total por ciudad
British West Indies	Grand Cayman	Cayman Divers World Unlimited	$59.660,05	76511,8	76511,8
		Fisherman's Eye	$12.022,00		
		Safari Under the Sea	$4.829,75		
Canada	Kitchener	Marmot Divers Club	$12.223,25	97358,9	12223,25
	Vancouver	Davy Jones' Locker	$44.073,65		44073,65
	Winnipeg	On-Target SCUBA	$41.062,00		41062
Columbia	Bogota	Fantastique Aquatica	$90.143,40	90143,4	90143,4
Cyprus	Kato Paphos	Sight Diver	$261.575,80	261575,8	261575,8
Fiji	Suva	Divers-for-Hire	$21.534,00	37076	21534
	Taveuni	Princess Island SCUBA	$15.542,00		15542
Greece	Ayios Matthaios	Divers of Corfu, Inc.	$98.278,60	98278,6	98278,6
Republic So. Africa	Johannesburg	Central Underwater Supplies	$6.675,95	6675,95	6675,95
US	Catalina Island	Catamaran Dive Club	$52.703,55	1494901,8	52703,55
	Eugene	Frank's Divers Supply	$20.602,00		20602

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 activar el agrupamiento debemos definir *agregados* (*aggregates*) sobre el conjunto de datos. Hay dos formas de hacerlo: definir los agregados en la propiedad *Aggregates*, o definir campos agregados en el Editor de Campos. En nuestro ejemplo seguiremos la segunda vía. En el Editor de Campos ejecutamos el comando *New field*. Como

recordará de la sección anterior, ahora tenemos la opción *Aggregate* en el tipo de campo. Como nombre, utilizaremos *TotalPorPais*. Repetiremos la operación para crear un segundo campo agregado: *TotalPorCiudad*. En el Editor de Campos, estos dos campos se muestran en un panel situado en la parte inferior del mismo:



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 *Field-Name* (Delphi no muestra los campos agregados en la lista desplegable de la propiedad). Para completar el ejemplo, debemos asignar *False* en la propiedad *DefaultDrawing* de la rejilla, y dar respuesta su evento *OnDrawColumnCell*:

```
procedure TwndMain.DBGrid1DrawColumnCell(Sender: TObject;
   const Rect: TRect; DataCol: Integer; Column: TColumn;
   State: TGridDrawState);
begin
   if (Column.Field = ClientDataSet1Pais) or
      (Column.Field = ClientDataSet1TotalPorPais) then
      if not (gbFirst in ClientDataSet1.GetGroupState(1)) then Exit;
   if (Column.Field = ClientDataSet1Ciudad) or
      (Column.Field = ClientDataSet1TotalPorCiudad) then
      if not (gbFirst in ClientDataSet1.GetGroupState(2)) then Exit;
   DBGrid1.DefaultDrawColumnCell(Rect, DataCol, Column, State);
end;
```

Aquí hemos aprovechado la función *GetGroupState* del conjunto de datos. Esta función devuelve un conjunto con los valores *gbFirst*, *gbMiddle* y *gbLast*, para indicar si estamos al inicio, al final o en mitad de un grupo.

Capítulo

39

El Modelo de Objetos Componentes: la teoría

LA PRIMERA GENERACIÓN de herramientas RAD impulsó la programación en el modelo cliente/servidor al permitir utilizar de manera sencilla datos ubicados en servidores SQL remotos. Como ya conoce el lector, las bases de datos SQL permiten la especificación de reglas de empresa que son procesadas en el servidor, con las lógicas ventajas que resultan de distribuir el procesamiento entre al menos un par de ordenadores, y de centralizar las verificaciones de consistencia y seguridad. Este modelo de programación es el conocido como *modelo en dos capas* (*two-tiered model*). Si son aprovechados correctamente, los servidores SQL pueden disminuir el tráfico total de red en comparación con las aplicaciones en una capa para redes punto a punto, en la que los datos residen en ficheros al estilo Paradox y dBase.

Pero para aprovechar todas las ventajas del modelo, debemos ir más allá. ¿Por qué se afirma que el procesamiento en el modelo de dos capas es distribuido? Porque hay código en el servidor, en la forma de reglas declarativas, *triggers* y procedimientos almacenados, que son ejecutados en ese nodo. ¿Qué posibilidad tenemos de mover más código del cliente al servidor? Poco más podemos hacer, pues esta programación se realiza en el dialecto SQL de cada fabricante de bases de datos; el lenguaje SQL, además, no es adecuado para ciertas tareas: procesamiento gráfico, Internet, etc.

Existe una amplia gama de técnicas para desarrollar aplicaciones distribuidas en el entorno de Windows, como RPC, zócalos (*sockets)* y las tuberías con nombre (*named pipes*). Delphi, en particular, ofrece un par de componentes para la programación de *sockets*: *TClientSocket* y *TServerSocket*, para desarrollar aplicaciones que se comuniquen entre sí en redes que soporten los protocolos TCP/IP ó IPX/SPX. No obstante, la tendencia actual es utilizar interfaces de mayor nivel, que permitan la creación y uso de *objetos distribuidos*. En el mundo de Microsoft, estas técnicas están basadas en el *Modelo de Objetos Componentes*, o COM, según sus siglas en inglés, y en su extensión DCOM para la comunicación entre objetos remotos.

Pero antes de poder realizar siquiera una aplicación mínima basada en COM, tenemos que asimilar un gran volumen de información acerca de los fundamentos de este modelo. No es el propósito de este libro explicar en profundidad COM y OLE, por lo que nos limitaremos al impacto de estas técnicas sobre la programación en Delphi. En el presente capítulo expondremos la teoría y terminología del modelo, y en el siguiente mostraremos ejemplos prácticos.

> CORBA es un sistema de soporte para objetos distribuidos que es aproximadamente equivalente a COM, aunque puede ejecutarse sobre más sistemas operativos y plataformas. Delphi 4 también permite el uso y programación de objetos CORBA, pero por consideraciones de tiempo no voy a tratar ese tema en esta edición.

COM, DCOM, OLE...

Hace ya cierto tiempo, cuando nadie apostaba un duro por Windows como sistema operativo, a Microsoft se le planteó un problema: era necesario que en las presentaciones generadas con uno de sus productos, PowerPoint, se pudieran incluir gráficos diseñados con Microsoft Graph. La solución, presentada en 1991, fue OLE 1.0: un protocolo mediante el cual podían incluirse objetos creados y mantenidos por otras aplicaciones dentro de documentos compuestos. OLE 1 estaba implementado sobre la base del protocolo DDE, que a su vez utilizaba mensajes y memoria global para la comunicación entre aplicaciones.

Pero OLE 1 era lento y frágil, y peor documentado que de costumbre. La solución que se ensayó en la siguiente versión fue comenzar desde cero, definiendo un modelo de objetos que permitiera de la forma más directa y transparente la comunicación entre procesos diferentes. El trabajo teórico fue desarrollado inicialmente a cuatro manos, por Microsoft y DEC, que ni siquiera pudieron ponerse de acuerdo en el significado de las siglas del modelo resultante: COM; *Component Object Model* (*Modelo de Objetos Componentes*) para Gates y familia, y *Common Object Model*, para Digital. Al final, DEC abandonó el proyecto.

COM es un modelo binario, independiente de los lenguajes de programación que pueden soportarlo, y del sistema operativo o plataforma donde se ejecuta. El concepto principal de COM es la *interfaz*, un conjunto abstracto de prototipos de métodos, que se interpretan como un conjunto de servicios que pueden ofrecer ciertos objetos. Los objetos de este modelo pueden implementar una o más interfaces simultáneamente, suministrando código a las definiciones abstractas de métodos de éstas. El modelo define varias interfaces básicas que hacen frente a las tareas más comunes de introspección, manejo de memoria e instanciación.

Sobre esta base, la segunda versión de OLE puede entenderse como una "implementación" particular de COM[30]. En primer lugar, OLE proporciona código concreto para ayudar en la programación de objetos e interfaces. En segundo lugar, se definen nuevas interfaces de servicios: la automatización OLE, los controles ActiveX y, cómo no, la incrustación y vinculación de objetos, ampliada esta vez con técnicas como la activación *in-situ* (*in-place activation*).

Si el lector no se toma demasiado en serio la comparación, le propongo la siguiente: Windows permite utilizar bibliotecas de funciones desarrolladas en un lenguaje determinado desde cualquier otro lenguaje; el código de la implementación de estas funciones se comparte físicamente por todas las aplicaciones que concurrentemente hagan uso del mismo. Estoy hablando de las DLLs, por supuesto. A estas alturas, todos conocemos las ventajas que se obtienen al pasar de la programación tradicional, basada en bibliotecas de funciones, a la programación orientada a objetos, con bibliotecas de clases. Pues bien, COM ofrece, aproximadamente, el equivalente "orientado a objetos" de las Bibliotecas de Enlace Dinámico. Digo aproximadamente porque, por ejemplo, sólo parte de los objetos COM se implementan mediante DLLs, el cliente y el servidor no tienen que residir en el mismo ordenador, etc. La analogía queda en que ambas técnicas ponen a disposición del programador una serie de "servicios" o "recursos" bien determinados, las DLLs en forma de funciones y procedimientos, y el modelo COM en forma de clases y objetos. Además, la implementación de estos servicios es transparente para el programador; éste puede programar en Delphi, y los servicios haber sido implementados en FORTRAN (me parece que he exagerado un poco, ¿no?).

Con posterioridad a la presentación de COM, Microsoft introdujo la especificación *DCOM*, o *Distributed COM*, que explica cómo aplicaciones situadas en diferentes ordenadores pueden compartir objetos entre sí. Aunque desde el principio COM estaba abierto a esta posibilidad, la especificación DCOM afina detalles acerca de cómo realizarla. Recientemente, Microsoft ha incluido soporte DCOM en Windows NT 4.0 y en Windows 98, y ha puesto a disposición del público un parche para los usuarios de Windows 95 que quieran utilizar DCOM.

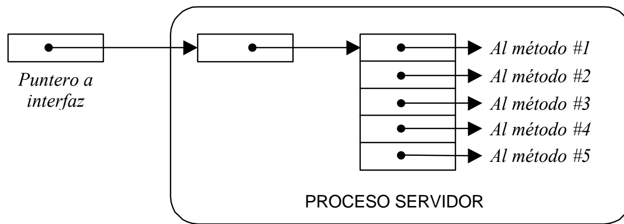
Interfaces

La base de todo el Modelo de Objetos Componentes es, como hemos mencionado, el concepto de *interfaz*. Sabemos que una de las técnicas fundamentales de la programación consiste en denegar al programador el acceso a la implementación de un recurso; de esta forma lo protegemos de los cambios que ocurrirán muy probable-

[30] Aunque al principio todos escribían "OLE 2", Microsoft ha insistido en eliminar el ordinal de la denominación del producto, para indicar que ésta sí es la versión "definitiva", y que no se esperan cambios revolucionarios en el modelo de programación. Vivir para ver.

mente en la misma. Consecuentemente con esta filosofía, en COM usted nunca trabajará con un objeto COM, ni con el puntero al objeto. Por el contrario, con lo que podrá contar es con *punteros a interfaces*.

Una *interfaz* define un conjunto de servicios ofrecidos por un objeto. Físicamente, le corresponde una tabla de punteros a métodos, similar a la Tabla de Métodos Virtuales que hemos estudiado en el capítulo sobre Herencia y Polimorfismo.



Aproximadamente, puede entenderse una interfaz como una clase en la que solamente se han declarado métodos virtuales abstractos, sin implementación alguna. Tenga siempre presente que COM es una especificación binaria, por lo cual la representación textual de la definición de una interfaz depende del lenguaje en que estemos trabajando. Delphi trata las interfaces como tipos de datos especiales, que se declaran mediante la palabra clave **interface**. Como convenio, cada nombre de interfaz comienza con la letra *I*, del mismo modo que los nombres de clases de Borland comienzan con *T*. Muestro a continuación la descripción de una interfaz en Delphi:

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

Si prestamos atención a la declaración veremos, a continuación de la palabra **interface**, algo parecido a un insulto en hexadecimal. En realidad, se trata de un entero de 128 bits que identifica a la interfaz ante el mundo. Podemos prescindir de este número, pero en tal caso solamente podríamos utilizar la interfaz dentro de la aplicación que la define.

A este tipo de números se les conoce como *GUIDs* (*Global Unique Identifiers*, o identificadores únicos globales), y los guiones y llaves son pura parafernalia mnemotécnica, por si algún chalado es capaz de recordarlos. Si usted o yo creamos una interfaz, debemos asignarle un número único de éstos. Podemos pedirle uno a Microsoft, de forma similar en que las direcciones de Internet se piden a la organización INTERNIC; realmente, Microsoft lo tiene más fácil, al tratarse de 128 bits en vez de 32.

Pero los GUID también pueden generarse localmente, por medio de una función definida en el API de Windows, *CoCreateGuid*. Esta función genera números con *unicidad estadística*; vamos, que la probabilidad de un conflicto es menor que la de que acertemos la lotería. El algoritmo empleado introduce la fecha y la hora dentro de un sombrero de copa, le añade el identificador de red del usuario, o información del BIOS, si no estamos en red, realiza un par de pases con la varita mágica y saca un conejo con un GUID en la boca.

> La mayoría de las veces, Delphi genera por nosotros los identificadores de interfaces y otros tipos de GUID utilizados por COM. Pero si en algún momento necesita uno de estos números, pulse la combinación CTRL+MAY+G en el Editor de Delphi.

La relación entre objetos e interfaces es la siguiente: un objeto COM *implementa* una o varias interfaces. Los objetos de la clase *TMarteens* pueden implementar las interfaces *IProgramador* e *IAlpinista* (es un decir). Una clase implementa una interfaz cuando suministra un cuerpo a todos los métodos de las interfaces soportadas; un poco más adelante veremos cómo se hace. El otro punto importante es que cuando creamos un objeto mediante COM, nunca trabajamos directamente con dicho objeto, sino con alguna de sus interfaces.

Necesito hacer otra aclaración: dentro de la aplicación que actúa como servidora, la implementación del objeto puede realizarse internamente mediante una clase del lenguaje de programación en que se desarrolló la aplicación. Tal es el caso de los objetos COM desarrollados con Delphi. Pero esto no es necesario. Recuerde que se puede programar un servidor COM en C puro y duro. Internamente, el servidor representará el objeto mediante estructuras, *arrays* y punteros, aunque externamente a los clientes se les ofrezca un "objeto".

La interfaz IUnknown

IUnknown, cuya definición presenté en la sección anterior, es la interfaz fundamental que todos los objetos COM deben implementar. Viene a ser algo así como la clase *TObject* con respecto a las demás clases de Delphi; toda interfaz debe contener como mínimo los tres prototipos de métodos definidos en *IUnknown*. Los objetivos de estos métodos son:

- *QueryInterface*: Aporta introspección a los objetos. Dada una interfaz cualquiera, nos dice si el objeto asociado soporta otra interfaz determinada; en caso afirmativo, nos devuelve el puntero a esa interfaz. Si tenemos un puntero a una interfaz de tipo *IProgramador* en nuestras manos, por ejemplo, podemos aprovechar *QueryInterface* para averiguar si el objeto asociado soporta también la interfaz *IAlpinista*.

- *AddRef*: Debe incrementar un contador de referencias asociado al objeto. En conjunción con el siguiente método, se utiliza para controlar el tiempo de vida de los objetos COM.
- *Release*: Debe decrementar el contador de referencias, y destruir el objeto si el contador se vuelve cero.

Existe una relación de herencia simple entre interfaces. Tomemos como ejemplo la siguiente definición extraída de Delphi:

```
type
   IDataBroker = interface(IDispatch)
      ['{6539BF65-6FE7-11D0-9E8C-00A02457621F}']
      function GetProviderNames: OleVariant; safecall;
   end;
```

Según la definición, la interfaz *IDataBroker* debe ofrecer todos los métodos de la interfaz *IDispatch*, sean los que sean, más el método *GetProviderNames*. Cuando una declaración de interfaz no menciona ninguna interfaz madre, se asume que ésta es *IUnknown*. Por lo tanto, todas las interfaces ofrecen obligatoriamente los tres métodos *AddRef*, *Release* y *QueryInterface*.

Cuando utilizamos punteros a interfaces en lenguajes tradicionales como C y C++, tenemos que ocuparnos explícitamente de llamar a los métodos *AddRef* y *Release* para que el objeto COM asociado sepa cuándo destruirse. Estos lenguajes no tienen un tipo especial para las interfaces, sino que éstas se consideran como punteros. En cambio, Delphi trata de forma especial a las interfaces, y se encarga automáticamente de llamar a los métodos *AddRef* y *Release* en las asignaciones, y cuando una variable de interfaz termina su tiempo de vida. Por ejemplo, nosotros tecleamos esta simple función:

```
procedure HacerAlgo(I: IUnknown);
var
   J: IUnknown;
begin
   J := I;
   // … más instrucciones …
end;
```

Delphi la traduce, obviando posibles optimizaciones, de este modo:

```
procedure HacerAlgo(I: IUnknown);
var
   J: IUnknown;
begin
   J := nil;
   try
      J := I;
      J.AddRef;
      // … más instrucciones …
```

```
    finally
      if J <> nil then J.Release;
    end;
  end;
```

También *QueryInterface* se disfraza en Delphi. El operador **as** se encarga de convertir punteros de un tipo de interfaz en otro. Para que esta conversión sea posible, sin embargo, las interfaces que participan en la misma deben poseer un identificador de interfaz.

Implementación de interfaces en Delphi

Ahora veremos cómo Delphi permite crear clases, en el sentido tradicional del término, que implementan los métodos ofrecidos por las interfaces. Las clases que resultan de esta operación pueden utilizarse internamente por la misma aplicación o ser exportadas para su utilización por otras aplicaciones, mediante mecanismos que involucran al Registro de Windows y que estudiaremos más adelante.

Recuerde que la interfaz solamente define *qué* métodos soporta, y *qué* deben hacer esos métodos, no *cómo* lo hacen. Esto en Delphi es responsabilidad de la clase que soporta la interfaz. Supongamos que declaramos una interfaz *IProgramador* de la siguiente forma:

```
type
  IProgramador = interface(IUnknown)
    // No hace falta aclarar que desciende de IUnknown
    procedure BeberCafe(Marca: string);
    procedure TragarAspirinas(Cantidad: Integer);
  end;
```

Podemos definir entonces una clase *TMarteens*, cuyos objetos soporten dicha interfaz:

```
type
  TMarteens = class(TInterfacedObject, IProgramador, IAlpinista)
    // Estos métodos sí necesitan un cuerpo
    procedure BeberCafe(Marca: string);
    procedure TragarAspirinas(Cantidad: Integer);
    procedure Trepar(Cumbre: string);
    procedure Aterrizar(Sano: Boolean);
  end;
```

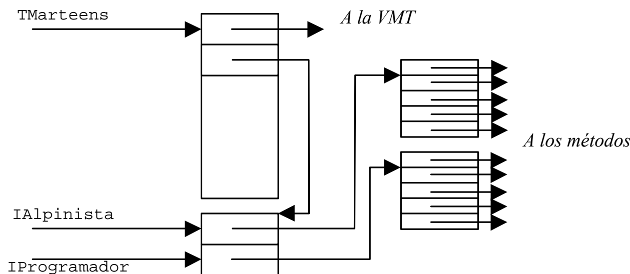
Aquí la clase proporciona el cuerpo de los métodos que necesita *IProgramador*. La asociación se produce por el nombre del método, aunque existen mecanismos sintácticos para cuando el método ofrecido por la clase no tiene el mismo nombre que el que necesita la interfaz. Eso sí: la cantidad y tipos de parámetros deben ser siempre idénticos. Observe además que esta misma clase *TMarteens* implementa la interfaz *IAlpinista*, que contiene los métodos *Trepar* y *Aterrizar*.

¿Por qué hemos heredado de la clase *TInterfacedObject*? La idea es que esta nueva clase implementa los métodos de *IUnknown* que están implícitos en las dos interfaces soportadas. Esto es, *TInterfacedObject* tiene métodos *QueryInterface*, *_AddRef* y *_Release*, que satisfacen la axiomática requerida por *IUnknown*. Es más, estas implementaciones se aplican tanto a los prototipos de *IUnknown* contenidos en *IProgramador* como a los de *IAlpinista*. Gracias a esto, el comportamiento de una llamada a *QueryInterface*, por ejemplo, será el mismo tanto si consideramos al objeto como programador o como alpinista.

Cuando digo que a un objeto de tipo *TMarteens* podemos considerarlo como programador o alpinista, me estoy refiriendo a que se pueden realizar asignaciones polimórficas entre interfaces, y entre objetos e interfaces. Por ejemplo:

```
var
   Ian: TMarteens;
   UnProgramador: IProgramador;
   UnAlpinista: IAlpinista;
begin
   Ian := TMarteens.Create;
   UnAlpinista := Ian;
   UnAlpinista.Trepar('K2');
   UnProgramador := Ian;
   UnProgramador.TragarAspirinas(1);
   UnAlpinista.Aterrizar(True);
   Ian.Free;
end;
```

El Modelo de Objetos Componentes no dice nada acerca de cómo el objeto debe representar físicamente sus interfaces, así que cada lenguaje tiene plena libertad en este sentido. Una posible representación de un objeto de la clase *TMarteens* sería la siguiente:



Esto quiere decir que la aparentemente simple asignación:

```
UnProgramador := Ian;
```

no se limita a copiar un puntero dentro de otro, sino que tiene que realizar una búsqueda o desplazamiento.

Cómo obtener un objeto COM

Las interfaces pueden utilizarse como una técnica de programación más, sin necesidad de vincularlas al resto de las técnicas de COM. La misma aplicación puede definir las interfaces, crear las clases que las implementan y manejar internamente sus propios objetos. Sin embargo, lo habitual es que el cliente que utiliza un objeto y el servidor que lo define e implementa sean módulos diferentes. Para que una aplicación pueda crear un objeto definido en otro módulo, sea ejecutable o DLL, necesitamos funciones y procedimientos definidos por COM.

La función más sencilla que permite crear un objeto COM es *CoCreateInstance*. Sin embargo, Delphi ofrece una función más conveniente, basada en la misma:

```
function CreateComObject(const ClassID: TGUID): IUnknown;
```

Al parecer el parámetro es uno de los identificadores globales únicos que ya conocemos, ¿de qué se trata ahora? Las clases que implementan objetos COM también se identifican a nivel del sistema mediante GUID. Estos *identificadores de clases* son asignados por el implementador de la clase, y deben estar registrado en alguna estructura de datos global para que cualquier aplicación pueda hacer uso de ellos. Esta estructura es, por supuesto, el Registro. Más adelante veremos cómo se almacena la información sobre clases en el Registro.

> A las clases COM también se les denomina en inglés *CoClasses*, o clases de componentes. Sin embargo, en este libro traduciremos el término cómo *clases COM*, para evitar la confusión con los componentes de Delphi.

Una clase COM puede implementar varias interfaces; ya hemos visto cómo Delphi lo hace. Pero *CreateComObject* devuelve un puntero a alguna de sus interfaces *IUnknown*. Esta política es válida como punto de partida, porque sabemos que utilizando *Query-Interface* o el operador **as** podemos obtener cualquier otra interfaz soportada por el mismo objeto.

Otra variante de creación de objetos COM la ofrece la siguiente función de Delphi:

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

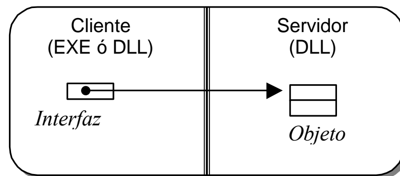
Esta vez COM busca la información de la clase indicada en el registro de otra máquina, crea el objeto basado en la clase en el espacio de direcciones de ese ordenador, y devuelve nuevamente un "puntero" al objeto creado. Más adelante veremos como DCOM hace posible esta magia.

Dentro del proceso, en la misma máquina, remoto...

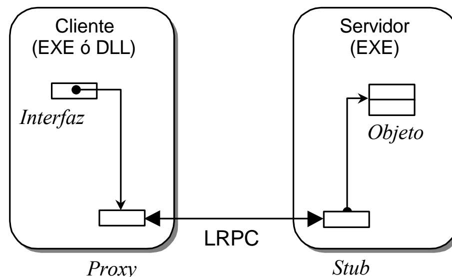
¿Dónde pueden residir los objetos COM? Actualmente existen tres modelos diferentes de servidor:

- El objeto reside en una DLL.
- El objeto reside en una aplicación local, dentro del mismo ordenador.
- El objeto 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 con servidores de este estilo. La ventaja de los servidores dentro del proceso es que comparten el mismo espacio de direcciones que la aplicación cliente. De esta forma, el puntero a la interfaz que utiliza que cliente apunta a la verdadera interfaz proporcionada por el objeto, y no es necesario realizar traducción alguna de los parámetros en las llamadas a métodos, resultando en tiempos de ejecución muy eficientes.



El siguiente paso son los servidores locales; los programas de Office pertenecen a esta categoría. La aplicación servidora puede ejecutarse frecuentemente por sí misma, además de efectuar su papel de servidor de objetos. En este modelo, cliente y servidor funcionan sobre espacios de direcciones diferentes y, gracias a las características del modo protegido del procesador, las aplicaciones no tienen acceso directo a las direcciones ajenas. Por lo tanto, OLE implementa un protocolo de comunicación entre aplicaciones llamado *Lightweight Remote Procedure Calls*, ó *LRPC*.



En este caso, el objeto del servidor no puede pasar directamente un puntero a interfaz a sus clientes. Entonces COM crea, de forma transparente para ambos procesos, *objetos delegados* (*proxies*) en el cliente y en el servidor, que se encargan de la comunicación real entre ambos. El cliente piensa que está trabajando directamente con la interfaz del servidor, pero realmente está actuando sobre una especie de mando de distancia. Cuando se produce una llamada a un método, los parámetros deben empaquetarse en el lado cliente, enviarse vía LRPC, para que finalmente el servidor los desempaquete. Lo mismo sucede con los valores retornados por el servidor. A este proceso se le denomina *marshaling*, y más adelante veremos que ciertas interfaces ofrecen soporte predefinido para el mismo.

Si el servidor y el cliente se ejecutan en distintos puestos, el protocolo *LRPC* se sustituye por el protocolo de red *RPC*, ó *Remote Procedure Calls*. RPC permite ejecutar procedimientos remotos pasando por alto incluso las diferencias en la representación de tipos de datos entre distintos tipos de procesadores. Por ejemplo, el orden entre el byte más significativo y el menos significativo es diferente en los procesadores de Intel y de Motorola. Por supuesto, el *marshaling* debe ocuparse de estos detalles.

La implementación actual de DCOM solamente admite aplicaciones como contenedores remotos de objetos COM, no DLLs.

El huevo, la gallina y las fábricas de clases

Una de las interfaces importantes definidas por COM es *IClassFactory*, que proporciona los servicios necesarios para la creación de objetos. De esta manera COM resuelve el eterno dilema de quién fue primero, si el huevo, la gallina o la tortilla de patatas: si usted quiere crear un objeto, la aplicación o DLL servidora debe ser capaz de ofrecer primero una interfaz *IClassFactory* para que sea ésta quien se encargue de la creación. Los clases que soportan esta interfaz reciben el nombre de *fábricas de clases*. Para que la petición de una interfaz *IClassFactory* al servidor sea exitosa, debemos ser capaces de ofrecer a COM un objeto que soporte dicha interfaz.

Si se trata de un servidor ejecutable, el objeto se registra como fábrica de clases mediante la función del API *CoRegisterClassObject*. Delphi realiza estas acciones automáticamente, creando las fábricas de clases en las secciones de inicialización de las unidades que definen objetos COM. Si el servidor es una DLL, COM pide el objeto a la fábrica de clases mediante una función que la DLL debe exportar:

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

El mismo mecanismo que utiliza Delphi en el caso de servidores ejecutables sirve para los servidores DLL, y es transparente para el programador.

Como todas las interfaces, *IClassFactory* contiene los métodos de *IUnknown*, además de definir sus métodos específicos:

```
function CreateInstance(const UnkOuter: IUnknown;
   const IID: TGUID; out Obj): HResult; stdcall;
function LockServer(fLock: BOOL): HResult; stdcall;
```

CreateInstance hace el papel de constructor de objetos: se le suministra el identificador único de clase y, si puede crear objetos de esa clase, devuelve un puntero a la interfaz *IUnknown* del objeto recién creado. *LockServer*, por su parte, mantiene a la aplicación servidora en memoria, para acelerar la creación posterior de objetos por la misma. Estos métodos son implementados dentro de la clase *TComObject*, que de forma similar a *TInterfacedObject* con respecto a *IUnknown*, puede servir de base a clases que actúen como fábricas de clases.

Un factor a tener en cuenta es el modo de instanciación de cada clase. Las dos formas principales son la de *instancias múltiples* y la de *instancia simple*. En la primera, cada petición de creación de un objeto COM crea efectivamente un objeto diferente, mientras que en la segunda, se devuelve el puntero a la interfaz del objeto ya creado.

Existe una interfaz adicional, *IClassFactory2*, que es utilizada por los controles ActiveX para manejar los ficheros de licencia.

OLE y el registro de Windows

Para que las aplicaciones clientes puedan sacar provecho de los servidores, estos últimos deben anunciarse como tales en algún lugar. El tablón de anuncios de Windows es el Registro de Configuraciones. Los datos acerca de servidores COM se guardan bajo la clave *HKEY_CLASSES_ROOT*. Estas son, por ejemplo, las entradas del registro asociadas con un servidor COM programado con Delphi; el formato es el ficheros de exportación del Editor del Registro:

```
[HKEY_CLASSES_ROOT\SemServer.Semaforo]
@="SemaforoObject"
[HKEY_CLASSES_ROOT\SemServer.Semaforo\Clsid]
@="{9346D962-195A-11D1-9412-00A024562074}"
[HKEY_CLASSES_ROOT\CLSID\{9346D962-195A-11D1-9412-00A024562074}]
@="SemaforoObject"
[HKEY_CLASSES_ROOT\CLSID\{9346D962-195A-11D1-9412-
00A024562074}\LocalServer32]
@="C:\\MARTEENS\\PROGS\\SEMSERVER\\SEMSERVER.EXE"
[HKEY_CLASSES_ROOT\CLSID\{9346D962-195A-11D1-9412-
00A024562074}\ProgID]
@="SemServer.Semaforo"
[HKEY_CLASSES_ROOT\CLSID\{9346D962-195A-11D1-9412-
00A024562074}\Version]
@="1.0"
```

```
[HKEY_CLASSES_ROOT\CLSID\{9346D962-195A-11D1-9412-
00A024562074}\TypeLib]
@="{9346D960-195A-11D1-9412-00A024562074}"
```

Para explicar el significado de esta información, lo mejor es mostrar paso a paso lo que ocurre cuando una aplicación pide la creación de un objeto COM. Es más fácil comenzar con la creación mediante *CreateComObject*:

```
var
   RefInterfaz: IUnknown;
begin
   RefInterfaz := CreateComObject(CLSID_Semaforo);
   // …
end;
```

- El identificador de clase se busca en la clave *CLSID* de la raíz de las clases del registro: *HKEY_CLASSES_ROOT*. Dentro de la misma, se nos dice que el servidor es una aplicación local (*LocalServer32*) y obtenemos su nombre y su ruta, para ejecutarla.
- A la aplicación se le pide una interfaz *IClassFactory*. La forma en que COM realiza esta operación depende, como hemos visto, de si se trata de un servidor DLL o de un ejecutable. Con la interfaz en la mano, COM crea el objeto, y devuelve un puntero a su interfaz *IUnknown* principal a la aplicación cliente.

Sin embargo, es bastante incómodo tener que trabajar con los identificadores globales de clase, por lo que la mayoría de los servidores definen un nombre "legible" para las clases que implementan. En la jerga de COM, a estos nombres se les conoce como *identificadores de programas*. En concreto, la función *CreateOleObject*, que estudiaremos más adelante, utiliza identificadores de programas para señalar la clase:

```
var
   RefInterfaz: IDispatch;
begin
   RefInterfaz := CreateOleObject('SemServer.Semaforo');
   // …
end;
```

Cuando esto sucede, se efectúa este paso en primer lugar:

- Windows busca el nombre de clase directamente bajo la clave raíz de las clases, y al encontrarla se queda con el identificador de la clase, en la clave subordinada *CLSID*; en nuestro caso, el identificador es {9346D962-195A-11D1-9412-00A024562074}. Los dos pasos antes mencionados se ejecutan a continuación.

La función *ProgIdToClassId*, de Delphi, convierte un identificador de programa en el GUID correspondiente a su clase.

¿Cómo se registra un servidor?

Existen técnicas estándar para registrar un servidor COM, y dependen de si éste es una DLL o un ejecutable. En el primer caso, la DLL debe exportar estas funciones:

```
function DllRegisterServer: HResult; stdcall;
funcion DllUnregisterServer: HResult; stdcall;
```

Es responsabilidad del servidor implementar estas dos funciones. Aquí Delphi nos echa una mano, pues ofrece la unidad *ComServ* ofrece estas funciones, que deciden qué clases registrar en dependencia de las fábricas de clases que el servidor inicializa.

Si se trata de un ejecutable, el convenio para registrarlo es invocarlo con el parámetro */regserver*, o sin parámetros. En el primer caso, solamente se graban las entradas del registro y la ejecución del programa termina inmediatamente. Para borrar las entradas del registro, el convenio consiste en utilizar el parámetro */unregserver*. Nuevamente, la unidad *ComServ* de Delphi asume de forma automática estas responsabilidades.

Inprise ofrece el programa *tregsvr.exe*, cuyo código fuente se encuentra entre las demostraciones de Delphi, para registrar indistintamente servidores ejecutables y bibliotecas dinámicas.

Automatización OLE

Quizás la interfaz más popular de OLE sea *IDispatch*, que define los servicios de *automatización OLE* (*OLE Automation*). Esta es una forma de ejecutar métodos de un objeto COM, que puede residir en una DLL o en otra aplicación. La llamada a los métodos remotos puede efectuarse por medio del nombre de los mismos, en el estilo de un lenguaje de macros. De hecho, la automatización OLE sustituye al viejo mecanismo de ejecución de macros de DDE. La automatización OLE se ha hecho popular gracias sobre todo a que Word, y los restantes productos de Office, pueden actuar como servidores de automatización. Pero muchos otros productos comienzan a aprovechar esta técnica, ya sean como servidores o controladores. Por ejemplo, MS SQL Server puede controlar objetos de automatización desde *scripts* programados en Transact-SQL. La automatización también es importante para Delphi, pues es una de las formas en que se implementan las técnicas de acceso a bases de datos remotas con Midas. Delphi define una interfaz *IDataBroker*, derivada de *IDispatch*, y por medio de la misma comunica las aplicaciones clientes con los servidores de aplicaciones.

La declaración de *IDispatch* es la siguiente:

```
type
  IDispatch = interface(IUnknown)
    ['{00020400-0000-0000-C000-000000000046}']
    function GetTypeInfoCount(out Count: Integer): Integer;
      stdcall;
    function GetTypeInfo(Index, LocaleID: Integer;
      out TypeInfo): Integer; stdcall;
    function GetIDsOfNames(const IID: TGUID; Names: Pointer;
      NameCount, LocaleID: Integer; DispIDs: Pointer): Integer;
      stdcall;
    function Invoke(DispID: Integer; const IID: TGUID;
      LocaleID: Integer; Flags: Word; var Params;
      VarResult, ExcepInfo, ArgErr: Pointer): Integer; stdcall;
  end;
```

La clase de Delphi *TAutoObject* ofrece una implementación de los prototipos de *IDispatch*, y sirve como clase base para implementar servidores de automatización. Un objeto que implemente esta interfaz define usualmente métodos adicionales, que corresponden a su funcionalidad específica, y que se ponen a disposición de las aplicaciones clientes. Ahora bien, los métodos de automatización, en principio, se ejecutan especificando el nombre como una cadena de caracteres, en el mismo estilo de una macro. La función *GetIDsOfNames* se encarga de traducir los nombres de métodos y de parámetros en códigos numéricos; estos códigos son finalmente pasados a *Invoke*, que se encarga de ejecutar el método deseado.

¿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? Mediante una tabla, denominada *dispatch interface*. En Delphi se definen mediante la declaración de tipos **dispinterface**. Por ejemplo:

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

Esta no es una interfaz, en el sentido normal de la palabra, sino que es una tabla para el consumo interno del objeto que implemente una interfaz de automatización con los dos métodos anteriores.

> Delphi 2 permitía la programación de clientes y servidores de automatización. Para programar servidores, se utilizaba una sección **automated** dentro de los objetos COM que se creaban. Esta forma de programación sigue existiendo en Delphi, pero existen actualmente técnicas más potentes, que hacen obsoleta a la anterior.

Controladores de automatización en Delphi

A los clientes de objetos de automatización se les llama *controladores de automatización*. Un controlador de automatización escrito en Delphi puede obtener un puntero a la interfaz *IDispatch* de un objeto de automatización mediante una llamada a la función *CreateOleObject*, de la unidad *ComObj*, que he mencionado poco antes:

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

Esta función nos evita realizar una búsqueda del identificador de programa dentro del Registro y obtener a continuación el puntero a dicha interfaz.

¿Qué hacemos cuando tenemos la interfaz *IDispatch* en la mano? Si estuviéramos trabajando en C/C++, tendríamos que realizar llamadas a *GetIDOfNames* y a *Invoke* de la forma más difícil, empaquetando los parámetros. En Delphi, por el contrario, se puede utilizar una extensión sintáctica que permite ejecutar los métodos de automatización de forma similar a como ejecutaríamos los métodos de una clase normal. La técnica consiste en asignar el valor de retorno de *CreateOleObject* a una variable de tipo *Variant*, y aplicar los métodos directamente a esta variable, como si fuera una variable de clase:

```
var
   WordIntf: Variant;
begin
   WordIntf := CreateOleObject('Word.Basic');
   WordIntf.AppShow;
   WordIntf.FileNewDefault;
   WordIntf.Insert('Mira lo que hago con Word'#13);
   WordIntf.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*, exportada por Microsoft Word. Los métodos que se llaman a continuación son métodos exportados por esa clase. El primero hace visible a Word, el segundo crea un fichero con propiedades por omisión, el siguiente inserta una línea y el último guarda el fichero en el directorio de la aplicación. No hay que destruir explícitamente al objeto creado, pues el compilador se encarga de hacerlo en el código de finalización de la rutina, cuando desaparece la variable local

WordIntf. Si quisiéramos destruir el objeto explícitamente, tendríamos que utilizar una asignación como la siguiente:

```
WordIntf := UnAssigned;
```

También puede controlarse Word mediante la clase *Word.Application*, a partir de Office 97. El siguiente ejemplo es equivalente al anterior:

```
procedure TForm1.Button1Click(Sender: TObject);
var
   WApp, WDoc: Variant;
begin
   WApp := CreateOleObject('Word.Application');
   WApp.Visible := True;
   WDoc := WApp.Documents.Add;
   WDoc.Activate;
   WDoc.Content.InsertAfter('Mira lo que hago con Word'#13 +
      '(Otra línea)');
   WDoc.SaveAs(
      FileName := ChangeFileExt(Application.ExeName, '.doc'));
end;
```

En la última instrucción estoy mostrando un ejemplo de parámetro con nombre. Resulta que el método *SaveAs*, del objeto *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'));
```

Pero optamos por especificar directamente el nombre del parámetro a utilizar. Recuerde que esta rocambolesca sintaxis se traduce, en definitiva, a llamadas a *GetIDOfNames* y en operaciones sobre los parámetros de *Invoke*.

Interfaces duales

La gran ventaja de la automatización OLE es que permite un acoplamiento débil entre cliente y servidor. El cliente no tiene por qué conocer todos los detalles de la interfaz del servidor. Piense un momento en un servidor de automatización como Word, que ofrece cientos de funciones, con montones de parámetros y valores por omisión. A usted, sin embargo, solamente le interesa saber cómo abrir un fichero e imprimirlo. ¿Para qué tener que aprender todos los demás métodos? Esto hace que sea fácil programar clientes para este tipo de servidores.

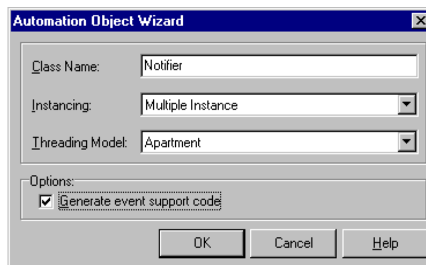
Pero también hay ventajas a la inversa: un cliente puede trabajar con más servidores. Si yo necesito ejecutar un método *Imprimir* y sé que determinado servidor lo imple-

menta, me da lo mismo la existencia de otros métodos dentro de ese servidor, siempre que pueda utilizar su interfaz *IDispatch*.

No obstante, y a pesar de las mejoras en tiempo de acceso que ofrecen los *DispId*, las llamadas a métodos por medio de *Invoke* son inevitablemente lentas. Para paliar el problema, los servidores de automatización generados en Delphi ofrecen *interfaces duales*. Esto quiere decir que podemos llamar a los métodos de automatización como si fueran macros, por medio de la función *Invoke*, pero que el mismo objeto implementa una interfaz personalizada que exporta directamente dichos métodos. En el siguiente capítulo veremos cómo aprovechar las interfaces duales generadas por Delphi.

Eventos y controles ActiveX

El tipo de comunicación más frecuente en la automatización OLE es unidireccional: el cliente da órdenes al objeto servidor. El paso siguiente para facilitar la programación es que el servidor pueda enviar eventos al cliente. El mecanismo ofrecido por COM para esto son los *puntos de conexión*, definidos mediante las interfaces *IConnectionPoint* e *IConnectionPointContainer*. Para programar servidores con emisión de eventos en Delphi 3 teníamos que incluir e implementar manualmente estas interfaces, lo cual significaba bastante trabajo. En Delphi 4, la mayor parte del trabajo la realizan los asistentes para la creación de servidores COM. La siguiente imagen corresponde al asistente para la creación de objetos de automatización; preste atención a la casilla de la parte inferior del cuadro de diálogo:

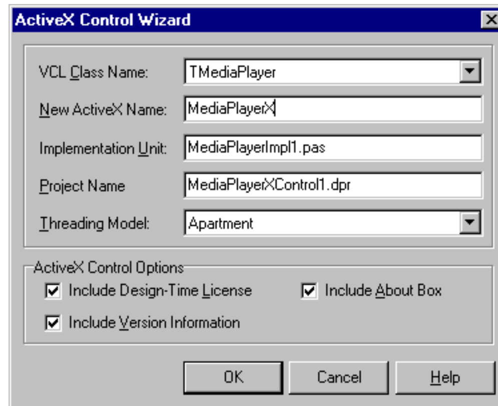


De todos modos, es bastante laborioso crear un cliente que intercepte los eventos de un servidor de automatización, pues este cliente tiene que crear a su vez un pequeño servidor de automatización que implemente las respuestas a los eventos, y pasar su interfaz al verdadero servidor. En el directorio *demos\ActiveX\Oleauto\Word8* de Delphi hay un ejemplo de cómo atender a los eventos que lanza Word 8.

Otro tipo de servidores COM que lanzan eventos son los *controles ActiveX*. Estos combinan una interfaz *IDispatch*, interfaces para eventos y otras interfaces responsables de la persistencia de objetos y de su inclusión dentro de otros objetos visuales.

Una característica a destacar de los controles ActiveX es que deben implementarse dentro de DLLs, es decir, son servidores dentro del proceso.

Afortunadamente para nosotros, para programar un control ActiveX en Delphi solamente necesitamos tener un control VCL equivalente a mano; la programación de este último es relativamente más sencilla. Solamente necesitamos utilizar después un programa experto del Depósito de Objetos para convertir el control VCL en un flamante control ActiveX.

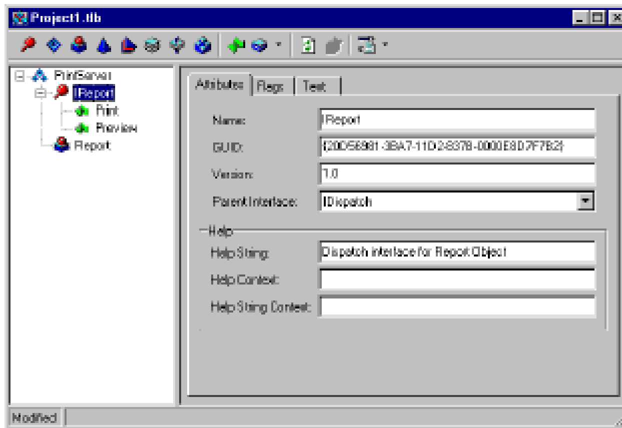


Delphi no permite convertir los controles de bases de datos propios. El motivo es que los controles ActiveX utilizan un mecanismo diferente al de Delphi para comunicarse con bases de datos. Ciertos métodos y propiedades tampoco pueden convertirse de forma automática, esta vez, porque el *marshaling* predefinido no puede trabajar con todos los tipos de datos de Delphi.

Bibliotecas de tipos

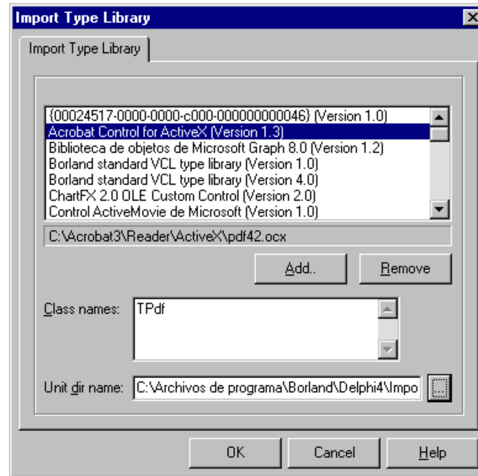
Con frecuencia es conveniente que los clientes potenciales conozcan dinámicamente los servicios ofrecidos por un servidor COM. El caso más típico es el de un control ActiveX; supongamos que incorporamos un nuevo control ActiveX, desarrollado en un desconocido lenguaje, en el entorno de Delphi. El entorno tiene que conocer qué propiedades, métodos y eventos soporta el nuevo control, para poder modificarlos desde el Inspector de Objetos. Otro ejemplo evidente es la optimización de llamadas a servidores de automatización OLE, aprovechando las interfaces duales. La forma en que COM almacena esta información es mediante *bibliotecas de tipos* (*type libraries*). Una biblioteca de tipos es un recurso de Windows, cuya clase se define como *typelib*. Este recurso acompaña a la DLL o aplicación que implementa el servidor COM.

Cuando utilizamos el asistente de Delphi para crear un objeto de automatización, generamos automáticamente una biblioteca de tipos para el proyecto, aunque ésta no es un requisito obligatorio para la automatización OLE. Los controles ActiveX, por el contrario, si exigen la presencia de esta estructura. En otros entornos de programación, para definir bibliotecas de tipos se utilizan los lenguajes de *script* ODL e IDL, pero Delphi utiliza sus propias herramientas visuales para facilitar la tarea. Las bibliotecas de tipos de los proyectos de Delphi se almacenan en ficheros de extensión *tlb* (*type libraries*) y son, por supuesto, globales al proyecto completo. Si un proyecto incluye una de estas bibliotecas, podemos leerla y modificarla mediante el comando de menú *View|Type library*. El editor de bibliotecas de tipos se muestra a continuación:



Paralelamente a la edición visual de la biblioteca de tipos, Delphi mantiene actualizada una unidad que contiene las declaraciones equivalentes en Pascal. Si declaramos en la biblioteca, por ejemplo, una interfaz dada, en la unidad se crea un tipo **interface** correspondiente. De esta forma, el proyecto puede aprovechar la información contenida en la biblioteca. El texto de esta unidad no debe ser modificado manualmente por el programador, pues en definitiva la información prioritaria es la que contiene la biblioteca de tipos. Además, cuando se realiza cualquier modificación en esta última, se sobrescribe el contenido de la unidad asociada. La unidad puede utilizarse también en aplicaciones clientes escritas en Pascal, para acceder a datos tales como los identificadores de clases y de interfaces, los tipos de interfaces, etc. Este paso es esencial para aprovechar las interfaces duales.

Cuando una aplicación cliente trabaja con un servidor creado con un lenguaje que no es Delphi, también podemos aprovechar su biblioteca de tipos, si es que ésta existe. Pero primero hay que crear la unidad Pascal equivalente a las declaraciones de la biblioteca. Para ello hay que ejecutar el comando *Project|Import type library*, estando activo el proyecto cliente:



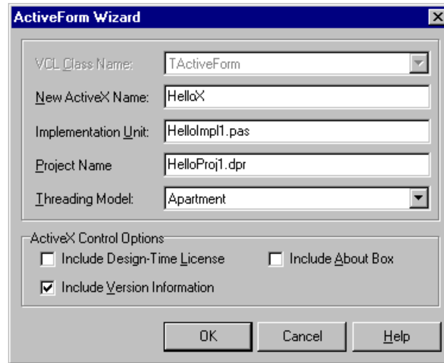
En la lista de bibliotecas aparecen todas aquellas que están registradas. Con los botones *Add* y *Remove* podemos registrar cualquier otra biblioteca, o eliminar alguna del Registro. En el caso de que la biblioteca contenga uno o más controles ActiveX, la unidad generada contendrá declaraciones de componentes para los mismos, y un método *Register* para poder incluir estos componentes en la Paleta de Componentes, si así lo desea. En la imagen de ejemplo, se ha seleccionado la biblioteca de tipos para el control ActiveX de Adobe Acrobat. Como vemos, esta biblioteca contiene una declaración para el componente *TPdf*.

ActiveForms: formularios en la Web

Supongamos que queremos desarrollar una aplicación para Internet, cuyo lado cliente debe ejecutarse en el contexto de un explorador. Ya hemos visto una forma de hacerlo, en el capítulo 37, utilizando formularios HTML y extensiones del servidor. Pero los formularios HTML tienen una gran limitación: los controles de edición son poco fantasiosos y pertenecen a unos pocos tipos básicos que no se pueden extender. Aunque las más recientes extensiones dinámicas a HTML, como la inclusión de lenguajes *script* interpretados en el cliente (JavaScript, VBScript), permiten el manejo de eventos disparados por los controles de edición HTML, seguimos enfrentándonos a la imposibilidad de crear componentes de edición a la medida. Una de las soluciones a este problema (en la galaxia Wintel) es incluir controles ActiveX dentro de páginas Web. No es una técnica de mi agrado, sobre todo por los problemas de portabilidad y seguridad que plantea, pero intentaré explicar a grandes rasgos una peculiar aplicación de la misma que Delphi hace posible.

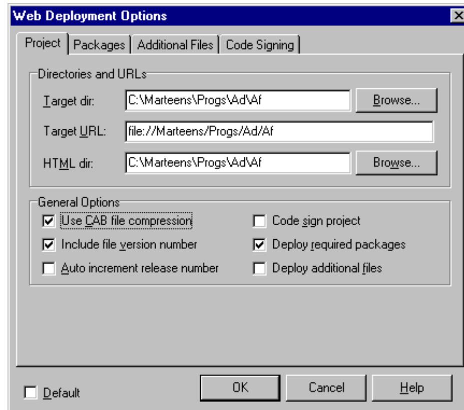
En vez de incluir controles ActiveX por separado dentro de una página Web, podemos crear con Delphi un *formulario activo*, o *ActiveForm*, que es sencillamente un

control ActiveX que se comporta como un formulario "normal" de Delphi: 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:



El formulario activo debe crearse dentro del contexto de una biblioteca dinámica ActiveX. De no ser éste el caso, Delphi cierra el proyecto activo y crea una nueva.

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*:

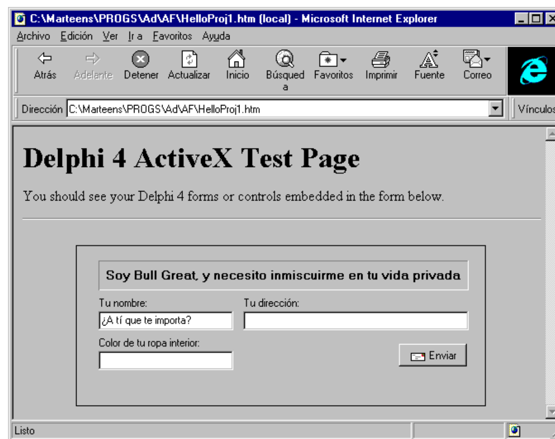


La idea es que Delphi puede generar automáticamente una página HTML de prueba para nuestra ActiveForm, y mediante el diálogo anterior podemos especificar las opciones de la página y la forma en la que vamos a distribuir el control. En la primera página, por ejemplo, debemos indicar dos directorios y una URL:

Opción	Significado
Target dir	Directorio donde se va a ubicar el control ActiveX compilado y listo para su distribución
Target URL	URL, vista desde un explorador Web, donde residirá el formulario activo
HTML dir	Directorio donde se va a crear la página HTML

En el resto del diálogo podemos indicar si queremos comprimir el formulario (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 siguiente imagen 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 usuarios más amigable, pero a costa de perder portabilidad.

La otra oportunidad consiste en situar en el formulario activo componentes de acceso a bases de datos. ¿Quiero decir conjuntos de datos del BDE? ¡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 desarrollaremos con Midas en el capítulo 41, para lo cual debemos colocar un servidor Midas en el mismo servidor Web. El cliente se conectaría al servidor a través de su dirección IP; en el capítulo sobre Midas veremos cómo se establece este tipo de conexión. Esta técnica asume, sin embargo, que el servidor HTTP puede ejecutar una aplicación Delphi (al

igual que lo asume el uso de extensiones ISAPI/NSAPI), y que el servidor Midas se ubica directamente en el ordenador que publica su IP a la Internet. Y no creo que esto último haga muy feliz al Administrador de su dominio, por motivos de seguridad.

Capítulo

40

El Modelo de Objetos Componentes: ejemplos

AHORA ESTAMOS EN CONDICIONES DE mostrar unos cuantos ejemplos de programación con el Modelo de Objetos Componentes. Incluiremos usos prácticos tanto de clientes como de servidores. Aunque no hay muchas diferencias en la programación COM entre las versiones 3 y 4 de Delphi, los ejemplos estarán basados principalmente en Delphi 4, como es de suponer.

Creación de objetos y manejo de interfaces

El primer ejemplo es muy sencillo: crearemos accesos directos mediante la interfaz del *shell* de Windows. En este caso, actuaremos como clientes de una clase definida por Microsoft. Crearemos un objeto de esa clase, que representará un nuevo acceso directo, cambiaremos sus propiedades de acceso directo y finalmente lo guardaremos en disco.

¿Cómo creamos el famoso objeto? ¿Debemos utilizar el nombre de la clase? No, utilizaremos el identificador de clase. Este identificador está definido en la unidad *ShlObj* de Delphi. Aquí muestro su definición, por si le entra la curiosidad:

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

Dentro de la misma unidad, está definido el identificador de la interfaz *IShellLink*, que es soportada por los objetos de la clase *CLSID_ShellLink*:

```
const
   SID_IShellLinkA = '{000214EE-0000-0000-C000-000000000046}';
```

Y, por supuesto, se define el tipo de interfaz *IShellLink*, cuya definición omito debido a su longitud. La clase *CLSID_ShellLink* también implementa la interfaz *IPersistFile*, que está definida en la unidad *ActiveX*. Por lo tanto, debemos añadir a la cláusula

uses de nuestro formulario las unidades *ShlObj* y *ActiveX*. Añada también la unidad *Registry*, que la necesitaremos para leer claves del registro.

El formulario principal de este proyecto tendrá un botón que, al ser pulsado, creará un acceso directo a la propia aplicación, dentro del menú de programas de Windows (no se olvide quitarlo cuando termine el proyecto). Este es el código que necesitamos:

```
procedure TForm1.Button1Click(Sender: TObject);
var
   U: IUnknown;
   SL: IShellLink;
   PF: IPersistFile;
   WS: WideString;
begin
   // Buscar la carpeta del menú de inicio
   with TRegistry.Create do
   try
      RootKey := HKEY_CURRENT_USER;
      OpenKey('SOFTWARE\Microsoft\Windows\CurrentVersion', False);
      OpenKey('Explorer\Shell Folders', False);
      WS := ReadString('Programs');
      WS := ChangeFileExt(WS + '\' +
         ExtractFileName(Application.ExeName), '.lnk');
   finally
      Free;
   end;
   // Crear el objeto
   U := CreateComObject(CLSID_ShellLink);
   // Definir las propiedades del acceso directo
   SL := U as IShellLink;
   SL.SetPath(PChar(Application.ExeName));
   SL.SetWorkingDirectory(
      PChar(ExtractFilePath(Application.ExeName)));
   SL.SetDescription(PChar(Application.Title));
   // Guardar el acceso directo
   PF := U as IPersistFile;
   PF.Save(PWideChar(WS), False);
end;
```

La primera parte del procedimiento se ocupa de buscar el directorio donde está la carpeta *Menú Inicio* de nuestra configuración de Windows. La buscamos en el registro bajo la raíz *HEY_CURRENT_USER*, y una vez que la tenemos, le añadimos el nombre de la aplicación cambiando su extensión a *lnk*. En la variable *WS* queda el nombre del fichero donde finalmente se almacenará el acceso directo.

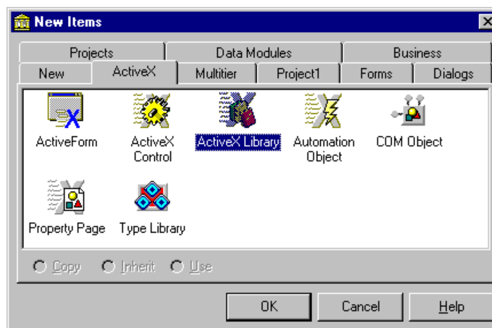
Interceptando operaciones en directorios

Cuando estudiamos algún tipo de sistema cliente/servidor (en el sentido más amplio, que va más allá de las bases de datos) y pensamos en algún ejemplo de programación, nuestras neuronas adoptan casi inmediatamente el papel de clientes. ¿Los servidores?,

bueno, alguien los debe programar... Sin embargo, en la programación COM es bastante frecuente que tengamos que desempeñar el papel de servidores.

Por ejemplo, si queremos definir extensiones al Explorador, Windows nos pide que le suministremos clases COM que implementen determinadas interfaces predefinidas. Las extensiones más comunes son los menús de contexto. ¿Ha instalado alguna vez el programa WinZip en su ordenador? ¿Se ha fijado en que el menú que aparece cuando se pulsa el botón derecho del ratón sobre un fichero cualquiera tiene una nueva opción, para añadir a un archivo comprimido? Esto sucede porque WinZip instala y registra DLLs que actúan como servidores dentro del proceso, y que son utilizadas automáticamente por el Explorador de Windows. Otro ejemplo de extensión se conoce como *copy hooks* (¿ganchos de copia?), y son módulos que se activan cuando el usuario va a realizar algún tipo de operación sobre un directorio.

En esta sección vamos a implementar un *copy hook*, porque aunque es la extensión más sencilla al Explorador, nos permite analizar cómo Delphi nos simplifica la programación de servidores COM. Como necesitamos un servidor DLL, vamos al Depósito de Objetos, a la página *ActiveX*, y seleccionamos el botón *ActiveX Library*:



Este asistente genera una DLL, que guardamos con el nombre *VerifyPas*. El código generado es el siguiente:

```
library VerifyPas;

uses
   ComServ;

exports
   DllGetClassObject,
   DllCanUnloadNow,
   DllRegisterServer,
   DllUnregisterServer;

{$R *.RES}

begin
end.
```

Como podemos apreciar, la DLL exporta cuatro funciones. La primera es la que COM utiliza para crear instancias de la clase exportada. La segunda sirve para que COM sepa cuándo puede descargar la DLL de memoria. Las dos últimas sirven, respectivamente, para crear y eliminar las entradas de la clase en el Registro de Windows. Todas ellas residen en la unidad *ComServ*, y su implementación se basa en la creación por Delphi de objetos *TComObjectFactory*, en donde residen las fábricas de clases del servidor.

Ahora creamos una nueva unidad vacía (en el Depósito de Objetos, *New|Unit*). En ella definiremos una clase COM que implementará la interfaz *ICopyHook*, definida en la unidad *ShlObj*. Primero necesitaremos un identificador para la clase, y lo declararemos como una constante:

```
const
   Class_VerPas: TGUID = '{F65A1CC0-3F50-11D2-837B-0000E8D7F7B2}';
```

¿De dónde he sacado ese número? Simplemente he pulsado CTRL+MAY+G en el Editor de Delphi, y he retocado un poco el resultado.

Es el momento de declarar la clase que implementa la interfaz *ICopyHook*. La clase basará su implementación en *TComObject*, que proporciona el cuerpo de los métodos básicos de *IUnknown*, y debe introducir un nuevo método con la cabecera que muestro a continuación:

```
type
   TVerifyPas = class(TComObject, ICopyHook)
   public
      function CopyCallback(Wnd: HWND; wFunc, wFlags: UINT;
         pszSrcFile: PAnsiChar; dwSrcAttribs: DWORD;
         pszDestFile: PAnsiChar; dwDestAttribs: DWORD): UINT;
         stdcall;
   end;
```

Se supone que *CopyCallback* es llamado por Windows cuando vamos a realizar alguna operación sobre un directorio. *wFunc* es la operación que se va a ejecutar, y *pszSrcFile* es el nombre del directorio origen de la operación. Esta función puede devolver uno de los tres valores siguientes:

Valor de retorno	Significado
IdYes	La operación es aprobada
IdNo	Cancela solamente esta operación
IdCancel	Cancela esta operación y las siguientes

Nosotros actuaremos solamente cuando se vaya a eliminar un directorio que contenga ficheros con extensión *pas* (¡partidismo!):

```
resourcestring
   SConfirmMsg = 'Este directorio contiene ficheros de Delphi'#13
      + '¿Desea eliminarlos?';

function TVerifyPas.CopyCallback(Wnd: HWND; wFunc, wFlags: UINT;
   pszSrcFile: PAnsiChar; dwSrcAttribs: DWORD;
   pszDestFile: PAnsiChar; dwDestAttribs: DWORD): UINT;
var
   SR: TSearchRec;
begin
   Result := IdYes;
   if wFunc = FO_DELETE then
   begin
      if FindFirst(StrPas(pszSrcFile) + '\*.pas',
         faAnyFile and not faDirectory, SR) = 0 then
         Result := MessageBox(Wnd, SConfirmMsg, 'Warning',
            MB_YESNOCANCEL);
      FindClose(SR);
   end;
end;
```

Ya tenemos la clase COM. ¿Cómo exportamos esta clase, y cómo asociamos el identificador de clase que hemos generado a la misma? Aquí debe ayudarnos Delphi, pues todo esto se logra creando, en la cláusula de inicialización de la unidad, un objeto de la clase *TComObjectFactory*, que suministrará al objeto global *ComServer* la información necesaria para el registro, y sobre la fábrica de clases. El objeto global *ConServer* está definido dentro de la unidad *ComServ*.

Lamentablemente, en este ejemplo no podemos utilizar directamente la clase *TComObjectFactory*. El objeto *ComServ* llama a la función *UpdateRegistry* de esta clase cuando necesita actualizar el Registro. Pero la implementación de esta función solamente crea las entradas correspondientes a un servidor normal, común y corriente. Para las extensiones del Explorador, hace falta añadir claves adicionales, pues el Explorador no va a revisar todas las clases registradas en el sistema, crear instancias de ellas y determinar cuál implementa una extensión y cuál no. En el caso de los *copy hooks*, debemos añadir valores en las siguiente claves:

```
HKEY_CLASSES_ROOT\Directory\shellex\CopyHookHandlers
HKEY_LOCAL_MACHINE\ SOFTWARE\Microsoft\Windows\CurrentVersion\
   Shell Extensions\Aproved
```

Lo que tenemos que hacer es derivar una clase a partir de *TComObjectFactory*, y sustituir su método *UpdateRegistry*:

```
type
   TVerifyPasFactory = class(TComObjectFactory)
   public
      procedure UpdateRegistry(Register: Boolean); override;
   end;

procedure TVerifyPasFactory.UpdateRegistry(Register: Boolean);
var
   ClassID: string;
```

```
begin
   if Register then
   begin
      inherited UpdateRegistry(Register);
      ClassID := GUIDToString(Class_VerPas);
      CreateRegKey('Directory\shellex\CopyHookHandlers\VerPas',
        '', ClassID);
      with TRegistry.Create do
      try
         RootKey := HKEY_LOCAL_MACHINE;
         OpenKey('SOFTWARE\Microsoft\Windows\CurrentVersion', True);
         OpenKey('Shell Extensions\Approved', True);
         WriteString(ClassID, 'Verify Delphi folders');
      finally
         Free;
      end;
   end
   else
   begin
      DeleteRegKey('Directory\shellex\CopyHookHandlers\VerPas');
      inherited UpdateRegistry(Register);
   end;
end;
```

Finalmente, creamos una cláusula de inicialización en la unidad, y creamos un objeto de la nueva clase:

```
initialization
   TVerifyPasFactory.Create(ComServer, TVerifyPas, Class_VerPas,
      '', 'Verify Delphi Folders', ciMultiInstance, tmApartment);
end.
```

Compile el proyecto y registre la DLL resultante. Puede utilizar el comando de menú *Run|Register ActiveX Server*, o el programa *tregsvr.exe* que viene en el directorio *bin* de Delphi. En cualquier caso, debe reiniciar el ordenador para que la extensión añadida surta efecto.

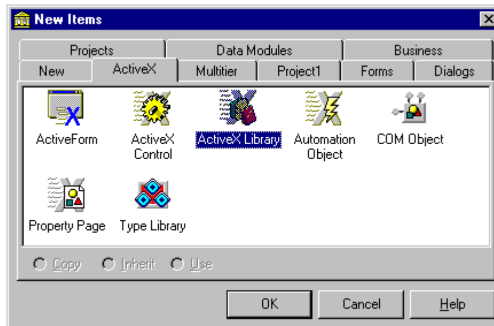
> Este ejemplo está basado en un programa de demostración de Delphi. En el directorio *ActiveX|ShellExt* de los ejemplos de Delphi podrá encontrar una interesante demostración acerca de cómo crear manejadores de menús de contexto.

Informes automatizados

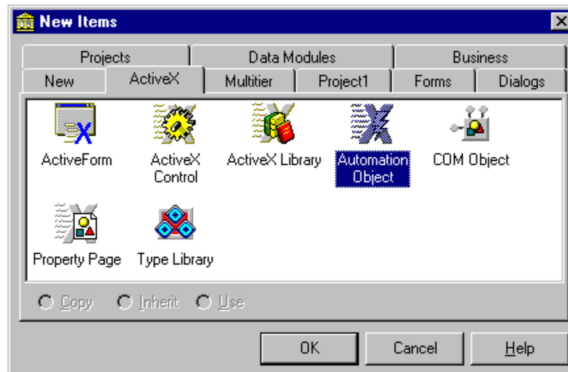
Ahora aprenderemos a crear servidores de automatización, es decir, servidores que ofrecen objetos que implementan la interfaz *IDispatch*, y que permiten llamar a sus objetos como si fueran macros. Utilizaremos la siguiente excusa para su creación: tenemos una aplicación de bases de datos y hemos definido una serie de informes con QuickReport sobre las tablas con las que trabaja el programa. Pero prevemos que el usuario querrá más adelante nuevos tipos de informes. ¿Qué hacemos cuando

el usuario tenga una idea genial de ese tipo? ¿Abrimos la aplicación para enlazar el nuevo informe? Claro que no. Una solución es definir los informes dentro de DLLs, y permitir la carga dinámica de las mismas. La otra, que exploraremos aquí, es programar los nuevos informes como servidores de automatización, y que la aplicación pueda imprimirlos llamando a métodos exportados tales como *Print*, *Preview* y *SetupPrint*.

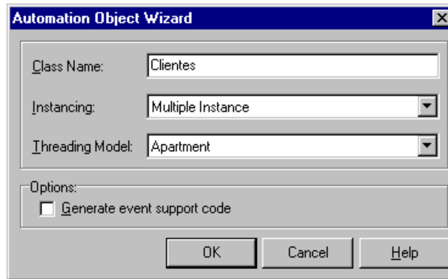
Comencemos por la creación de un servidor de este tipo. La primera decisión consiste en elegir entre un servidor dentro del proceso (una DLL) o fuera del proceso (un EXE). Por supuesto, una DLL es más eficiente, por lo que vamos al Depósito de Objetos, seleccionamos la página *ActiveX* y pulsamos el icono *ActiveX Library*.



Esta operación genera un esqueleto de DLL, que guardamos con el nombre *Informes*. Ya hemos visto, en la sección anterior, el formato de este fichero y las funciones que se exportan. También necesitaremos un informe. Para los propósitos de este ejercicio, da lo mismo qué informe utilicemos y cómo lo generemos; le sugiero que utilice el *QuickReport Wizard*, del Depósito de Objetos. Supondremos que se trata de un listado de clientes, que se implementa en el formulario *Form1*, y que este formulario contiene un informe llamado *QuickRep1*.



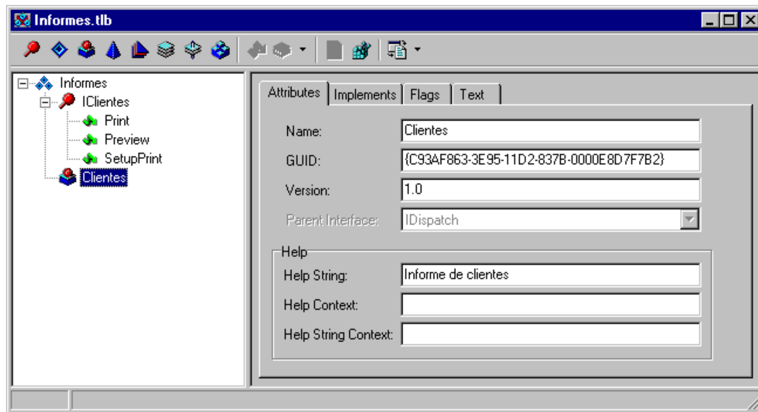
Para que esta DLL se convierta realmente en un servidor de automatización, debemos ejecutar el Depósito de Objetos, seleccionar y aceptar el icono *Automation Object*, de la página *ActiveX*. El diálogo que aparece a continuación nos ayuda a configurar las principales características de la clase que vamos a implementar. La primera de ellas es el nombre de la clase, donde tecleamos *Clientes*. Esto implica que el identificador de programa de la clase, para instanciar objetos con *CreateOleObject*, será *Informes.Clientes*, esto es, la concatenación del nombre del servidor con el nombre de la clase.



A continuación, se nos pide el modo de creación de instancias. Cuando se trata de un servidor dentro del proceso, como el nuestro, el modo de creación de instancias se ignora. También debemos especificar qué modelo de concurrencia vamos a utilizar. Como estamos dentro de una DLL, y no haremos uso de memoria global a sus diferentes instancias (léase ficheros asociados a memoria), estaremos cumpliendo con las condiciones del modelo *Apartment*. Por último, esta clase no enviará eventos a sus clientes, por lo que la casilla *Generate event support code* debe quedar sin marcar.

Cuando aceptamos el diálogo anterior, Delphi genera una biblioteca de tipos para la aplicación. Esta biblioteca es una entidad global al proyecto. Como nuestro proyecto se llama *Informes*, la biblioteca se guarda en el fichero *Informes.tlb*. Para la verificación de tipos en tiempo de compilación, Delphi genera una unidad, *Informes_TLB*, que contiene las declaraciones de la biblioteca de tipos expresadas con la sintaxis de ObjectPascal. Este fichero no debe editarse directamente, sino a través del editor visual de las bibliotecas de tipos, que puede activarse explícitamente mediante el comando de menú *View|Type Library*, y que muestro en la siguiente página.

Seleccionamos en el árbol de la izquierda del editor de la biblioteca de tipos el nodo correspondiente a la interfaz *IClientes*. Mediante el menú de contexto, o mediante los iconos de la barra de herramientas, añadimos tres métodos, a los cuales nombraremos *Print*, *Preview* y *SetupPrint*. Ninguno de los tres métodos tiene parámetros, ni valores de retorno; en caso contrario, deberíamos especificarlos en la segunda página de las propiedades de estos nodos. El resultado final de esta operación, expresado en ObjectPascal , corresponde al listado que incluyo a continuación:




```
unit Informes_TLB;

interface

uses Windows, ActiveX, Classes, Graphics, OleCtrls, StdVCL;

const
   LIBID_Informes: TGUID = '{C93AF860-3E95-11D2-837B-0000E8D7F7B2}';
   IID_IClientes: TGUID = '{C93AF861-3E95-11D2-837B-0000E8D7F7B2}';
   CLASS_Clientes: TGUID = '{C93AF863-3E95-11D2-837B-0000E8D7F7B2}';

type
   IClientes = interface;
   IClientesDisp = dispinterface;
   Clientes = IClientes;

   IClientes = interface(IDispatch)
      ['{C93AF861-3E95-11D2-837B-0000E8D7F7B2}']
      procedure Print; safecall;
      procedure Preview; safecall;
      procedure SetupPrint; safecall;
   end;

   IClientesDisp = dispinterface
      ['{C93AF861-3E95-11D2-837B-0000E8D7F7B2}']
      procedure Print; dispid 1;
      procedure Preview; dispid 2;
      procedure SetupPrint; dispid 3;
   end;

   CoClientes = class
      class function Create: IClientes;
      class function CreateRemote(const MachineName: string):
         IClientes;
   end;

implementation

uses ComObj;
```

```
class function CoClientes.Create: IClientes;
begin
   Result := CreateComObject(CLASS_Clientes) as IClientes;
end;

class function CoClientes.CreateRemote(const MachineName: string):
   IClientes;
begin
   Result := CreateRemoteComObject(MachineName, CLASS_Clientes)
      as IClientes;
end;

end.
```

Como podemos observar, Delphi ha definido la interfaz *IClientes* derivándola a partir de *IDispatch*. Así que cuando tengamos un puntero a esta interfaz en la mano, podremos ejecutar las rutinas *Print*, *Preview* y *SetupPrint* como macros, utilizando variables *Variant* de Delphi. La implementación del método *Invoke* de *IClientes* se basa en la estructura generada por la interfaz *IClientesDisp*, que es una **dispinterface**. Ahora bien, *IClientes* también ofrece un punto de entrada directo a las tres rutinas mencionadas. Si el cliente dispone en tiempo de compilación de la declaración de la interfaz *IClientes*, puede llamar directamente a cualquiera de las tres rutinas sin necesidad de pasar por el lento mecanismo de *Invoke*. Dos por el precio de uno, ¿ le parece bien? La clase *CoClientes*, por su parte, es una facilidad adicional para que las aplicaciones clientes que incluyan en su código esta unidad puedan crear instancias de la clase COM. Más adelante veremos un ejemplo de su uso.

Solo nos queda proporcionar una implementación a esta interfaz. Y aquí Delphi ha vuelto a echarnos una mano, pues ha generado una unidad, que hemos guardado con el nombre de *Clientes*, con el siguiente código:

```
unit Clientes;

interface

uses
   ComObj, ActiveX, Informes_TLB;

type
   TClientes = class(TAutoObject, IClientes)
   protected
      procedure Preview; safecall;
      procedure Print; safecall;
      procedure SetupPrint; safecall;
   end;

implementation

uses ComServ;

procedure TClientes.Preview;
begin
  // Nuestro código va aquí
```

```
end;
procedure TClientes.Print;
begin
   // Nuestro código va aquí
end;

procedure TClientes.SetupPrint;
begin
   // Nuestro código va aquí
end;

initialization
  TAutoObjectFactory.Create(ComServer, TClientes, Class_Clientes,
    ciMultiInstance, tmApartment);
end.
```

La instrucción que Delphi ha incluido en la inicialización crea la fábrica de clases, que es el punto de entrada que utiliza COM para generar las instancias de *Informes.Clientes*. Recuerde que también esta fábrica de clases es utilizada por Delphi para generar el código de registro que ejecutan las funciones exportadas *DllRegisterServer* y *DllUnregisterServer*.

La implementación de los tres métodos es elemental. Por brevedad, solamente mostraré la de *SetupPrint*:

```
procedure TClientes.SetupPrint;
begin
   with TForm1.Create(nil) do
   try
      QuickRep1.PrinterSetup;
      if QuickRep1.Tag = 0 then
         QuickRep1.Print;
   finally
      Free;
   end;
end;
```

Controladores de automatización

La creación de un controlador de automatización que utilice al servidor definido anteriormente es sumamente sencilla. Primero estudiaremos cómo utilizar el servidor que hemos desarrollado en la forma más sencilla posible, para mostrar después la técnica más segura y eficiente, que aprovecha la interfaz dual.

Creamos una nueva aplicación, con un solo formulario por el momento. Colocamos un cuadro de edición sobre el mismo, para que el usuario pueda teclear el nombre de la clase de automatización, y un botón para mostrar la vista preliminar del informe. Recuerde que el objetivo del ejemplo anterior era la creación de informes genéricos, que podían ejecutarse desde una aplicación con sólo conocer el nombre de la clase. Por supuesto, en una aplicación real emplearíamos un mecanismo más sofisticado

para ejecutar los informes genéricos, que quizás contara con la creación de claves especiales en el Registro durante la instalación de los informes. Pero aquí nos basta con una sencilla demostración de la técnica.

La respuesta al evento *OnClick* del botón que hemos situado en el formulario es la siguiente:

```
procedure TForm1.Button1Click(Sender: TObject);
var
   Rpt: Variant;
begin
   Rpt := CreateOleObject(Edit1.Text);
   Rpt.Preview;
end;
```

Como explicamos en el capítulo anterior, la aplicación de métodos a variantes se traduce a la resolución en tiempo de ejecución del cuerpo del método por medio de su nombre, a través de las funciones *GetIDOfNames* e *Invoke*, de la interfaz *IDispatch* obtenida. La técnica, conocida como *enlace tardío*, tiene un par de inconvenientes: es lenta e insegura. La lentitud es evidente, aunque tenemos que esperar a conocer alternativas para poder juzgar al respecto. En cuanto a la inseguridad, es fácil demostrar que un fallo en el nombre del método pasa inadvertido en tiempo de compilación y se producirá en tiempo de ejecución, probablemente en el momento menos oportuno, haciendo honor a la ley de Murphy. Sustituyamos la llamada al método *Preview* por la siguiente:

```
Rpt.SeñalarError;
```

Si compilamos este código, veremos que Delphi lo acepta sin chistar, a pesar de la *ñ* que hemos intercalado, y de que no hemos definido ningún método *SeñalarError* en el servidor de automatización. Aparentemente hemos conseguido un programa sin problemas, pero durante su funcionamiento, cada intento de ejecutar la instrucción anterior termina en catástrofe.

La alternativa a las llamadas mediante enlace tardío solamente está disponible a partir de Delphi 3. Aunque el puntero de interfaz que devuelve la función *CreateOleObject* es de tipo *IDispatch*, en realidad la clase correspondiente soporta la interfaz *IClientes*, que tiene una entrada en su tabla de punteros para el método *Preview*. Por supuesto, la

aplicación cliente debe disponer de la declaración de *IClientes*, que en este caso se encuentra en la unidad *Informes_TLB*, generada por Delphi. Añadimos esta unidad al proyecto, y la incluimos en la cláusula **uses** de la unidad del formulario principal. Ya podemos cambiar la respuesta del botón:

```
procedure TForm1.Button1Click(Sender: TObject);
var
   Rpt: IClientes;     // NUEVO
begin
   Rpt := CreateOleObject(Edit1.Text) as IClientes;
   Rpt.Preview;
end;
```

El operador **as** realmente está ejecutando el método *QueryInterface*, para ver si la interfaz *IDispatch* devuelta por *CreateOleObject* soporta una interfaz *IClientes* con el identificador de clase mencionado en la declaración del tipo. Es esencial, por lo tanto, que la interfaz que aparece a la derecha del operador **as** haya sido declarada con un GUID asociado. Por conveniencia, también podemos utilizar la clase *CoClientes* definida en *Informes_TLB*:

```
procedure TForm1.Button1Click(Sender: TObject);
var
   Rpt: IClientes;     // NUEVO
begin
   Rpt := CoClientes.Create;
   Rpt.Preview;
end;
```

Ahora Delphi puede verificar en tiempo de compilación que solamente utilicemos métodos aceptados por la interfaz. Además, la llamada al método se produce por medio de la tabla de punteros de la interfaz, en forma directa. Como el servidor que estamos utilizando es un servidor DLL, tenemos la ventaja adicional de que estamos trabajando con el objeto real, sin necesidad de *marshaling* intermedio alguno.

Declarando una interfaz común

Pero, un momento ... ¿no habíamos quedado en que íbamos a crear un mecanismo de impresión genérico? Entonces, ¿qué papel juega la referencia a la interfaz *IClientes* en el código anterior? Claro está, un informe de ventas no soportará dicha interfaz. La solución es, sin embargo, muy sencilla. Creamos una nueva unidad, a la cual nombraremos *GenRpt*, por ejemplo. En su interfaz definimos el siguiente tipo:

```
type
   IGenRpt = interface
      ['{444D1880-3F81-11D2-837B-0000E8D7F7B2}']
      procedure Print; safecall;
      procedure Preview; safecall;
      procedure SetupPrint; safecall;
```

```
   end;
```

El identificador de la interfaz ha sido generado con la combinación CTRL+MAY+G, que ya hemos empleado otras veces. Guardamos esta unidad, reabrimos el proyecto del servidor de automatización y vamos a la unidad *Clientes*, que es donde se implementa la clase del informe. Incluimos la unidad *GenRpt* en la cláusula **uses** de la interfaz y modificamos la declaración de la clase *TClientes*, para que también implemente la nueva interfaz *IGenRpt*:

```
type
   TClientes = class(TAutoObject, IClientes, IGenRpt)
   protected
      procedure Preview; safecall;
      procedure Print; safecall;
      procedure SetupPrint; safecall;
   end;
```

No hay que modificar la implementación de la clase, pues los métodos de *IGenRpt* corresponden a métodos existentes. Ahora vamos al controlador de automatización y volvemos a cambiar el código de respuesta del botón:

```
procedure TForm1.Button1Click(Sender: TObject);
var
   Rpt: IGenRpt;       // NUEVO
begin
   Rpt := CreateOleObject(Edit1.Text) as IGenRpt;
   Rpt.Preview;
end;
```

Esta vez la modificación es definitiva. Siempre que definamos una nueva clase de informe, nos aseguramos que implemente la interfaz *IGenRpt*, además de las interfaces que genere Delphi. De este modo, el controlador siempre podrá utilizar en forma directa los métodos de *IGenRpt*.

Un servidor de bloqueos

En determinadas aplicaciones, es inaceptable que un usuario se enfrasque en una larga operación de modificación de un registro ... para descubrir en el momento de la grabación que no puede guardar sus cambios, porque otro usuario ha modificado mientras tanto el mismo registro. Esto es lo que sucede habitualmente en las aplicaciones desarrolladas para bases de datos cliente/servidor.

A lo largo de este libro hemos visto técnicas que ayudan a disminuir la cantidad de conflictos que se pueden producir. Ahora ofreceremos una alternativa radical: implementaremos una aplicación que actúe como servidor remoto de bloqueos. La aplicación se ejecutará en un punto central de la red, que no tiene por qué coincidir con el del servidor SQL. Cada vez que un usuario intente acceder a un registro para modificarlo, debe pedir permiso al servidor, para ver si algún otro usuario está utili-

zándolo. El servidor mantendrá una lista de los registros "bloqueados". Y cuando una aplicación a la que se le haya concedido el derecho a editar un registro termine su edición, ya sea cancelando o grabando, la aplicación debe retirar el bloqueo mediante otra llamada al servidor.

Primero desarrollaremos el servidor, para después ponerlo a punto con una aplicación de prueba. De acuerdo a lo explicado anteriormente, son solamente dos los procedimientos que tendremos que implementar para el servidor de bloqueos:

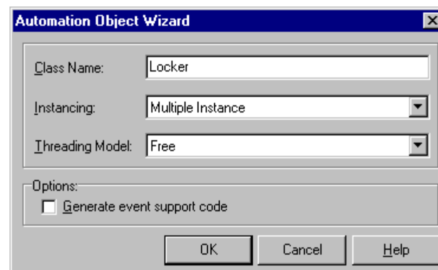
```
procedure Lock(TableName: WideString; Key: Integer);
procedure Unlock(TableName: WideString; Key: Integer);
```

Si esto no fuera Delphi, tendríamos que declarar funciones en lugar de procedimientos, para que devolviesen un código de error en el caso en que alguna operación no pudiera completarse. Sin embargo, nuestro objeto de automatización utilizará excepciones para señalar el fallo de un bloqueo.

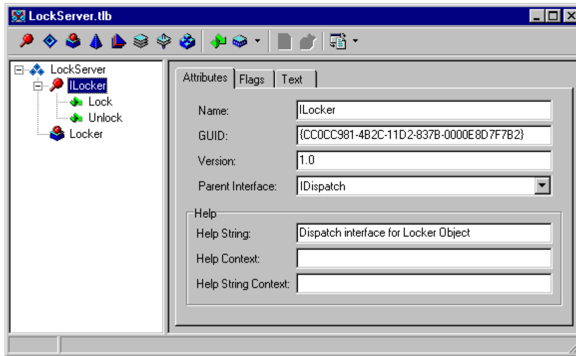
He decidido, para simplificar el código del servidor, que las claves primarias sean siempre un número entero. Según hemos visto en los ejemplos de este libro, existen razones convincentes para introducir claves artificiales incluso en los casos en que semánticamente la clave primaria natural sería una clave compuesta. El lector puede, no obstante, aumentar las posibilidades del servidor añadiendo métodos que bloqueen registros de tablas con otros tipos de claves primarias.

De modo que iniciamos una nueva aplicación (*File|New application*). A la ventana principal le damos el nombre de *wndMain* (¡qué original!). Más adelante, podrá modificar esta ventana para que aparezca como un icono en la bandeja de iconos de Windows. En el capítulo 8 hemos estudiado cómo.

Guardamos ahora la aplicación. Si lo desea, llame *Main* a la unidad de la ventana principal. Ahora bien, le ruego que llame *LockServer* al fichero de proyecto, y que no mueva durante el desarrollo el directorio donde estará situado. A continuación, ejecute el diálogo del Depósito de Objetos (*File|New*), vaya a la segunda página (*ActiveX*), y pulse el icono *Automation Object*:



Teclee *Locker* como nombre de clase, y deje el modo de generación de instancias como *Multiple instance*; esto quiere decir que una sola instancia de la aplicación podrá ejecutar concurrentemente el código de múltiples instancias de objetos de automatización. En el modelo de concurrencia especificaremos *Free*. Cuando cierre el diálogo, guarde el proyecto y nombre a la nueva unidad generada como *LockObj*.



La imagen anterior corresponde al Editor de la Biblioteca de Tipos del proyecto. Utilizando el botón *Method* debemos crear el par de métodos *Lock* y *Unlock* bajo el nodo correspondiente a la interfaz *ILocker*, con los prototipos establecidos al principio de esta sección. Después debemos pulsar el botón *Refresh*, para que los cambios se reflejen en la unidad de declaración de las interfaces, *LockServer_TLB*, y en la unidad *LockObj*, en la cual implementaremos los procedimientos *Lock* y *Unlock*.

La implementación de la lista de bloqueos

La estructura de datos a elegir para implementar la lista de bloqueos dependerá en gran medida de la estimación que hagamos del uso del servidor. Mis cálculos consisten en que un número relativamente pequeño de tablas contendrán un número elevado de bloqueos. Por lo tanto, mi sugerencia es utilizar una lista vectorial de nombres de tablas, ordenada alfabéticamente, y asociar a cada una de las entradas en esta lista, un puntero a un árbol binario ordenado.

Incluyo, a continuación, una unidad que implementa las operaciones de borrado y de búsqueda e inserción en un árbol binario.

```
unit Trees;

interface

type
  PTree = ^TTree;
  TTree = record
```

```
    Key: Integer;
    Left: PTree;
    Right: PTree;
  end;

function TreeInsert(AKey: Integer; var ATree: PTree): Boolean;
function TreeDelete(AKey: Integer; var ATree: PTree): Boolean;

implementation

type
  PPTree = ^PTree;

function TreeInsert(AKey: Integer; var ATree: PTree): Boolean;
var
   T: PPTree;
begin
   T := @ATree;
   while T^ <> nil do
      if AKey = T^.Key then
      begin
         Result := False;
         Exit;
      end
      else if AKey < T^.Key then
         T := @T^.Left
      else
         T := @T^.Right;
   T^ := New(PTree);
   T^.Key := AKey;
   T^.Left := nil;
   T^.Right := nil;
   Result := True;
end;

function TreeDelete(AKey: Integer; var ATree: PTree): Boolean;
var
   Del, Mayor: PTree;
   T: PPTree;
begin
   T := @ATree;
   while T^ <> nil do
   begin
      if AKey = T^.Key then
      begin
         Del := T^;
         if T^.Left = nil then
         begin
            T^ := Del.Right;
            Dispose(Del);
         end
         else if T^.Right = nil then
         begin
            T^ := Del.Left;
            Dispose(Del);
         end
         else
         begin
            // Buscar el mayor del subárbol izquierdo
            Mayor := Del.Left;
```

```
            while Mayor.Right <> nil do
               Mayor := Mayor.Right;
            Del.Key := Mayor.Key;
            TreeDelete(Mayor.Key, Del.Left);
         end;
         Result := True;
         Exit;
      end;
      if AKey < T^.Key then
         T := @T^.Left
      else
         T := @T^.Right;
   end;
   Result := False;
end;

end. { Trees }
```

La explicación de esta unidad va más allá de los propósitos del libro. El procedimiento *TreeInsert* trata de insertar un nuevo nodo con la clave indicada, y devuelve *False* cuando ya existe un nodo con esa clave. *TreeDelete*, por su parte, busca el nodo con la clave especificada para borrarlo, y devuelve *True* sólo cuando lo ha podido encontrar. Este procedimiento es ligeramente más complicado; el caso más barroco sucede cuando se intenta borrar un nodo que tiene ambos hijos asignados. Entonces hay que buscar el mayor de los nodos menores que el que se va a borrar (es decir, el nodo anterior en la secuencia de ordenación). Se mueve el valor de este nodo al nodo que se iba a borrar y se elimina físicamente el antiguo nodo, del que se ha extraído el valor anterior.

Control de concurrencia

La implementación directa de los métodos *Lock* y *Unlock* tiene lugar en la unidad *LockObj*, que es la que contiene la declaración de la clase *TLocker*. El primer paso es declarar cadenas para los mensajes de error de las excepciones, mediante definiciones de cadenas de recursos:

```
resourcestring
   SAlreadyLocked = 'Register locked by another user';
   SNotLocked = 'Register was not in use';
```

Esta declaración puede efectuarse lo mismo dentro de la interfaz como de la implementación de la unidad.

El próximo paso es definir variables globales, en la implementación de la unidad, para almacenar la lista de tablas que contienen bloqueos, el número de bloqueos existente en cualquier instante y un objeto de clase *TCriticalSection* que evitará colisiones en el uso de los objetos anteriores. Todos los objetos de automatización trabaja-

rán con estas variables. Si las definiéramos dentro de la clase *TLocker*, cada objeto de automatización tendría, por ejemplo, una lista de tablas diferente, lo cual es obviamente incorrecto:

```
var
   FTables: TStringList;
   FLocks: Integer;
   CritSect: TCriticalSection;
```

La clase *TCriticalSection* está definida dentro de la unidad *SyncObjs* de Delphi 4, que debemos incluir en la cláusula **uses** de *LockObj*. Las *secciones críticas* son uno de los mecanismos que ofrece Windows para sincronizar procesos paralelos. Operan dentro de una misma aplicación, a diferencia de otros recursos de sincronización como los semáforos y *mutexes*, pero son más eficientes.

La inicialización de las variables globales se produce durante la carga la unidad. Observe como se ha respetado la inicialización de la fábrica de clases:

```
initialization
   FTables := TStringList.Create;
   FTables.Sorted := True;
   FTables.Duplicates := dupIgnore;
   FLocks := 0;
   CritSect := TCriticalSection.Create;
   TAutoObjectFactory.Create(ComServer, TLocker, Class_Locker,
      ciMultiInstance, tmFree);
finalization
   FTables.Free;
   CritSect.Free;
end.
```

Ya estamos listos para implementar los métodos de automatización. La inserción de un registro en la lista de bloqueos se realiza del siguiente modo:

```
procedure TLocker.Lock(const TableName: WideString; Key: Integer);
var
   I: Integer;
   T: PTree;
begin
   CritSect.Enter;
   try
      I := FTables.Add(TableName);
      T := PTree(FTables.Objects[I]);
      if not TreeInsert(Key, T) then
         raise Exception.Create(SAlreadyLocked);
      FTables.Objects[I] := TObject(T);
      Inc(FLocks);
   finally
      CritSect.Leave;
   end;
end;
```

Este es el método que libera un bloqueo:

```
procedure TLocker.Unlock(const TableName: WideString;
   Key: Integer);
var
   I: Integer;
   T: PTree;
begin
   CritSect.Enter;
   try
      I := FTables.IndexOf(TableName);
      if I <> -1 then
      begin
         T := PTree(FTables.Objects[I]);
         if TreeDelete(Key, T) then
         begin
            FTables.Objects[I] := TObject(T);
            Dec(FLocks);
            Exit;
         end;
      end;
      raise Exception.Create(SNotLocked);
   finally
      CritSect.Leave;
   end;
end;
```

Observe el uso que se le ha dado a los métodos *Enter* y *Leave* de la sección crítica. Cada cliente que se conecte al servidor de bloqueos lanzará un hilo independiente. Si *Enter* encuentra que otro hilo ha ejecutado *Enter* y todavía no ha salido con *Leave*, pone en espera al hilo que lo ha ejecutado, hasta que quede liberada la sección. De este modo, garantizamos que un solo hilo a la vez manipule las estructuras de datos globales del servidor.

Por último, nos queda configurar el código de inicio del proyecto, para que pueda ejecutarse en este modelo de múltiples hilos que hemos presentado. Abrimos el fichero de proyecto (*Project|View source*) y añadimos las siguientes instrucciones antes de la inicialización de la aplicación:

```
begin
   CoInitFlags := COINIT_MULTITHREADED;      // ← Nuevo
   IsMultiThread := True;                    // ← Nuevo
   Application.Initialize;
   Application.CreateForm(TwndMain, wndMain);
   Application.Run;
end.
```

La variable *CoInitFlags* está definida en la unidad *ComObj*, y la constante que se le asigna reside en *ActiveX*. Esta variable es utilizada por la función *CoInitializeEx*, que se encarga de inicializar el sistema COM y que es llamada desde *Application.Initialize*. Por su parte, *IsMultiThread* indica al administrador de memoria de Delphi que estamos ejecutando una aplicación multihilos, y que debe utilizar mecanismos de sincronización para evitar conflictos durante la gestión de la memoria.

> Debido a un *bug* de Delphi 4, la aplicación anterior falla al terminar, lanzando un error de tiempo de ejecución. Mientras Inprise corrige este *bug* (o diseña otro mecanismo que no falle para habilitar la concurrencia de sus servidores) podemos seguir utilizando el servidor, con la condición de que lo ejecutemos antes de que cualquier cliente se conecte a él. De este modo, solamente se cierra el servidor cuando vamos a cerrar el sistema operativo del servidor, es decir, casi nunca.

Poniendo a prueba el servidor

Ahora que hemos completado la aplicación servidora, estamos en condiciones de ponerla a prueba. Comenzaremos las pruebas en modo local, con el servidor y los clientes situados en el mismo ordenador; de este modo evitaremos los problemas iniciales que pueden surgir durante la comunicación remota entre aplicaciones. El primer paso es compilar y ejecutar el servidor, al menos una vez, para que éste grabe las entradas necesarias en el registro de Windows.

A continuación creamos una nueva aplicación, en la cual colocamos una tabla enlazada a alguna base de datos SQL; el alias *IBLOCAL* de los ejemplos de Delphi puede servir. Para concretar, supongamos que la tabla elegida es *employee*, cuya clave primaria es la columna *Emp_No*. La interfaz de usuario, en sí, será muy sencilla: una rejilla de datos y una barra de navegación bastan. Vamos a la declaración de la clase *TForm1*, a la sección **private**, y añadimos la siguiente declaración:

```
type
   TForm1=class(TForm)
      // …
   private
      Locker: Variant;
   end;
```

Esta variable se inicializa durante el evento *OnCreate* del formulario; no es necesario destruir el enlace explícitamente, pues de eso se encarga el compilador:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
   Locker := CreateOleObject('LockServer.Locker');
   // Recuerde incluir la unidad ComObj
end;
```

El resto del código se concentrará en los eventos de transición de estado del componente de acceso a la tabla. Estos serán los eventos necesarios:

Acción	Eventos
Bloquear	*BeforeEdit*
Desbloquear	*OnEditError* *AfterPost* *AfterCancel*

En este caso simple, en que no necesitamos más instrucciones dentro de los manejadores de estos eventos, podemos crear tres métodos para que sirvan de receptores:

```
procedure TForm1.Table1BeforeEdit(DataSet: TDataSet);
begin
   Locker.Lock('employee', Table1['Emp_No']);
end;

procedure TForm1.Table1AfterPostCancel(DataSet: TDataSet);
begin
   // Compartido por AfterPost y AfterCancel
   Locker.Unlock('employee', Table1['Emp_No']);
end;

procedure TForm1.Table1EditError(DataSet: TDataSet;
   E: EDatabaseError; var Action: TDataAction);
begin
   Locker.Unlock('employee', Table1['Emp_No']);
end;
```

Ahora puede ejecutar dos copias de la aplicación y comprobar que, efectivamente, cuando editamos una fila de la tabla de empleados bloqueamos el acceso a la misma desde la otra copia de la aplicación.

Para sustituir el servidor local por un servidor remoto, incluya la unidad de la aplicación servidora *LockServer_TLB*, que contiene la declaración de las interfaces y sus identificadores globales, dentro del proyecto cliente. Cambie la declaración de la variable *Locker* del formulario principal del siguiente modo:

```
type
   TForm1=class(TForm)
      // …
   private
      Locker: ILocker;
   end;
```

Y modifique de la siguiente manera la inicialización de la misma:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
   Locker := CoLocker.CreateRemoteObject('NombreServidor');
end;
```

Capítulo

41

Midas

EN ESTE CAPÍTULO NOS OCUPAMOS del modelo de desarrollo de aplicaciones en múltiples capas, y en particular de los servicios que ofrece la tecnología Midas de Inprise (o Borland, como prefiera). Enfocaremos el desarrollo de servidores de datos en la capa intermedia, y veremos qué componentes nos proporciona Delphi para desarrollar las partes del servidor y del cliente de la aplicación, estudiando los mecanismos particulares de este modelo para resolver los problemas que ocasiona el acceso concurrente.

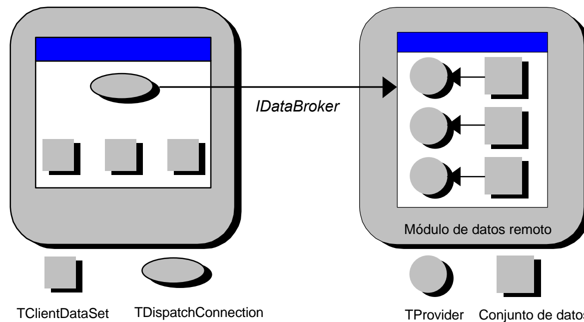
¿Qué es Midas?

Las siglas Midas quieren decir, con un poco de buena voluntad, *Multi-tiered Distributed Application Services*, que traducido viene a ser, poco más o menos, Servicios para Aplicaciones Distribuidas en Múltiples Capas. Midas no es una aplicación, ni un componente, sino una serie de servicios o mecanismos que permiten transmitir conjuntos de datos entre dos aplicaciones. En la primera versión de Midas, que apareció con Delphi 3.0, el vehículo de transmisión era DCOM, aunque podíamos utilizar un sustituto, también basado en COM, denominado OLEnterprise. Más adelante, con la versión de actualización 3.01, se incorporó el protocolo TCP/IP a esta lista. Con Delphi 4, podemos transmitir conjuntos de datos entre aplicaciones utilizando indistintamente COM/DCOM, OLEnterprise, TCP/IP y CORBA. En este libro trataremos solamente COM/DCOM y TCP/IP, por lo que todo lo que digamos de ahora en adelante sobre Midas se debe aplicar a estos protocolos de transporte.

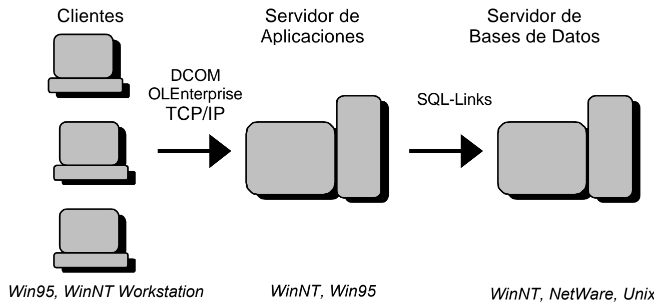
En el mecanismo básico de comunicación mediante Midas intervienen dos aplicaciones. Una actúa como servidora de datos y la otra actúa como cliente. Lo normal es que ambas aplicaciones estén situadas en diferentes ordenadores, aunque en ocasiones es conveniente que estén en la misma máquina, como veremos más adelante. También es habitual que el servidor sea un ejecutable, aunque si vamos a colocar el cliente y el servidor en el mismo puesto es posible, y preferible, programar un servidor DLL dentro del proceso.

A grandes rasgos, la comunicación cliente/servidor se establece a través de una interfaz COM nombrada *IDataBroker*, que es una interfaz de automatización. Esta interfaz permite el acceso a un conjunto de *proveedores*; cada proveedor es un objeto que soporta la interfaz *IProvider*. Los proveedores se sitúan en la aplicación servidora, y cada uno de ellos proporciona acceso al contenido de un conjunto de datos diferente. Esta estructura tiene su paralelo en la parte cliente. Los componentes derivados de la clase abstracta *TDispatchConnection* utilizan la interfaz *IDataBroker* de la aplicación servidora, y ponen a disposición del resto de la aplicación cliente los punteros a las interfaces *IProvider*. Cada clase concreta derivada de *TDispatchConnection* implementa la comunicación mediante un protocolo determinado: *TDComConnection*, *TSocketConnection*, *TCorbaConnection* y *TOLEnterpriseConnection*.

Nuestro viejo conocido, el componente *TClientDataSet*, puede utilizar una interfaz *IProvider* extraída de un *TDispatchConnection* como fuente de datos. A partir de estos componentes, la arquitectura de la aplicación es similar a la tradicional, con conjuntos de datos basados en el BDE. El siguiente esquema muestra los detalles de la comunicación entre los clientes y el servidor de aplicaciones:



Las más popular de las posibles configuraciones de un sistema de este tipo es la clásica aplicación en tres capas, cuya estructura se muestra en el siguiente diagrama:



En esta configuración los datos se almacenan en un servidor dedicado de bases de datos. El sistema operativo que se ejecuta en este ordenador no tiene por qué ser Windows: UNIX, en cualquiera de sus mutaciones, o NetWare pueden ser alternativas mejores, pues la concurrencia está mejor diseñada y, en mi opinión, son más estables, aunque también más difíciles de administrar correctamente. Hay un segundo ordenador, el servidor de aplicaciones, que actúa de capa intermedia. En esta máquina está instalado el BDE y los SQL Links, con el propósito de acceder a los datos del servidor SQL. El sistema operativo, por lo tanto, debe ser Windows NT Server, preferentemente, ó Windows NT Workstation e incluso Windows 95/98. La aplicación escrita en Delphi que se ejecuta aquí no tiene necesidad de presentar una interfaz visual; puede ejecutarse en segundo plano, aunque es conveniente disponer de algún monitor de control. Finalmente, los ordenadores clientes son los que se encargan de la interfaz visual con los datos; éstos son terminales relativamente baratas, que ejecutan preferentemente Windows 95/98 ó NT Workstation. En estos ordenadores no se instala el Motor de Datos de Borland, pues la comunicación entre ellos y el servidor intermedio se realiza a través de DCOM, TCP/IP, OLEnterprise o CORBA.

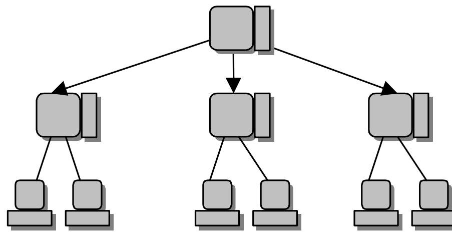
Cuándo utilizar y cuándo no utilizar Midas

Las bondades de Midas, pero sobre todo la propaganda acerca de la nueva técnica, ha llevado a muchos equipos de programación a lanzarse indiscriminadamente al desarrollo utilizando el modelo que acabo de presentar. Desde mi punto de vista, muchas de las aplicaciones planteadas no justifican el uso de este modelo. De lo que se trata no es de la conveniencia indiscutible de estratificar los distintos niveles de tratamiento de datos en una aplicación de bases de datos, sino de si es rentable o no que esta división se exprese físicamente. Es algo de sentido común el hecho de que al añadir una capa adicional de software o de hardware, cuya única función es la de servir de correa de transmisión, solamente logramos ralentizar la ejecución de la aplicación.

Olvidémonos por un momento de la estratificación metodológica y concentrémonos en el análisis de la eficiencia. ¿Cuál es la ventaja del modelo de dos capas, más conocido como modelo cliente/servidor? La principal es que gran parte de las reglas de empresa pueden implementarse en el ordenador que almacena los datos. Por lo tanto, para su evaluación los datos no necesitan viajar por la red hasta alcanzar el nodo en que residen las reglas. ¿Qué sucede cuando se añade una capa intermedia? Pues que en la mayoría de las aplicaciones no existen reglas de empresa lo suficientemente complejas como para justificar un nivel intermedio: casi todo lo que puede hacer un servidor Midas puede implementarse en el propio servidor SQL. Existe una excepción importante para este razonamiento: si nuestra aplicación requiere reglas de empresa que involucren simultáneamente a varias bases de datos. Por ejemplo, el inventario de nuestra empresa reside en una base de datos Oracle, pero el sistema de factu-

ración utiliza InterBase. Ninguno de los servidores SQL puede asumir por sí mismo la ejecución de las reglas correspondientes, por lo que la responsabilidad debe descargarse en un servidor Midas.

Existe, sin embargo, una técnica conocida como *balance de carga*, que en Delphi 3 solamente podía implementarse con OLEnterprise, pero que en Delphi 4 también puede aplicarse a otros protocolos. La técnica consiste en disponer de una batería de servidores de capa intermedia similares, que ejecuten la misma aplicación servidora y que se conecten al mismo servidor SQL. Los clientes, o estaciones de trabajo, se conectan a estos servidores de forma balanceada, de forma tal que cada servidor de capa intermedia proporcione datos aproximadamente al mismo número de clientes.



Se deben dar dos condiciones para que esta configuración disminuya el tráfico de red y permita una mayor eficiencia:

- El segmento de red que comunica al servidor SQL con los servidores Midas y el que comunica a los servidores Midas con los clientes deben ser diferentes físicamente.
- Los servidores Midas deben asumir parte del procesamiento de las reglas de empresa, liberando del mismo al servidor SQL.

Otra desventaja del uso de Midas consiste en que las aplicaciones clientes deben emplear conjuntos de datos clientes, y estos componentes siguen una filosofía optimista respecto al control de cambios. Ya es bastante difícil convencer a un usuario típico de las ventajas de los bloqueos optimistas, como para además llevar este modo de acción a su mayor grado. Los conjuntos de datos clientes obligan a realizar las grabaciones mediante una acción explícita (*ApplyUpdates*), y es bastante complicado disfrazar esta acción de modo que pase desapercibida para el usuario. Es más, le sugiero que se olvide de las rejillas de datos en este tipo de aplicación. Como ya sabe, si la curiosidad del usuario le impulsa a examinar el último registro de una tabla, no pasa nada pues el BDE implementa eficientemente esta operación. Pero un conjunto de datos clientes, al igual que una consulta, se ve obligado a traer todos los registros desde su servidor.

De todos modos, existen otras consideraciones aparte de la eficiencia que pueden llevarnos a utilizar servidores de capa intermedia. La principal es que los ordenadores en los que se ejecutan las aplicaciones clientes no necesitan el Motor de Datos de Borland ni, lo que es más importante, el cliente del sistema cliente/servidor. Conozco dos importantes empresas de *tele-marketing* que programan en Delphi; una de ellas tiene sus datos en Oracle y la otra en Informix. Es bastante frecuente que, tras contratar una campaña, tengan que montar de un día para otro treinta o cuarenta ordenadores para los nuevos operadores telefónicos y, después de instalar el sistema operativo, se vean obligados a instalar y configurar el BDE y el cliente de Oracle e Informix en cada uno de ellos. Evidentemente, todo resulta ser más fácil con los clientes de Midas, que solamente necesitan para su funcionamiento una DLL de 207 KB (Delphi 4).

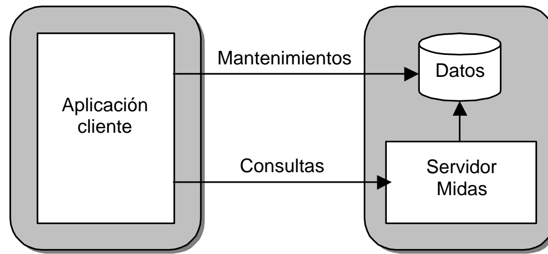
Midas y las bases de datos de escritorio

Los razonamientos anteriores suponen que nuestra base de datos reside en un sistema cliente/servidor. Curiosamente, una aplicación que funcione con Paradox o dBase puede beneficiarse mucho de utilizar un servidor Midas para determinadas tareas. No voy a sugerirle que utilice conjuntos de datos clientes para el mantenimiento de las tablas, pues estas operaciones se realizan más eficientemente accediendo directamente a los ficheros. Pero las consultas sobre la base de datos son excelentes candidatas a ser sustituidas por componentes *TClientDataSet*. Son las once de la mañana, y todos en la oficina se afanan en introducir datos de pedidos desde sus ordenadores a la Gran Base de Datos Central (que a pesar de las mayúsculas está en formato Paradox). A esa hora el Gran Jefe se digna en honrar con su presencia a sus empleados. Se repantinga en su sillón de cuero, enciende su Gran Ordenador y ejecuta el único comando que conoce de la aplicación: obtener un gráfico con la distribución de las ventas. Pero mientras el gráfico se genera, el rendimiento de la red cae en picado, las aplicaciones parecen moverse en cámara lenta, y los pobres empleados miran al cielo raso, suspiran y calculan cuántos meses quedan para las próximas vacaciones.

Lo que ha sucedido es que los datos del gráfico se reciben de un *TQuery*, mediante una consulta que implica el encuentro entre cinco tablas, un **group by** y una ordenación posterior. Y el encargado de ejecutar esa consulta es el intérprete local de SQL del Gran Ordenador. Esto significa que todos los datos de las tablas implicadas deben viajar desde la Gran Base de Datos Central hacia la máquina del Gran Jefe, y consumir el recurso más preciado: el ancho de banda de la red.

Mi propuesta es la siguiente: que las operaciones de mantenimiento sigan efectuándose como hasta el momento, accediendo mediante componentes *TTable* a la base de datos central. Pero voy a instalar un servidor Midas en el ordenador de la Gran Base de Datos, que contendrá la consulta que utiliza el gráfico, y exportará ese conjunto de

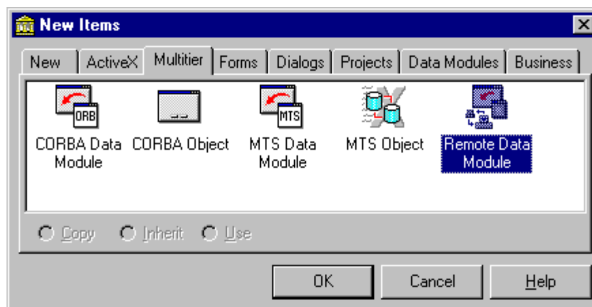
datos. Por otra parte, cuando la aplicación necesite imprimir el gráfico, extraerá sus datos de un conjunto de datos clientes que se debe conectar al servidor Midas. ¿Dónde ocurrirá la evaluación de la consulta? Está claro que en la misma máquina que contiene la base de datos y, como consecuencia, los datos originales no tendrán que viajar por la red. El conjunto resultado es normalmente pequeño, y será lo único que tendrá que desplazarse hasta el ordenador del Gran Jefe. ¿Resultado? Menos tráfico en la red (y empleados que miran menos al techo).



Un servidor Midas también puede encargarse de ejecutar operaciones en lote que no requieran interacción con el usuario. Estas operaciones se activarían mediante una petición emitida por un cliente, y se ejecutarían en el espacio de procesos del servidor. De esta forma, estaríamos implementando algo parecido a los procedimientos almacenados, pero sobre una base de datos de escritorio.

Módulos de datos remotos

Una vez sopesados los pros y los contras de las aplicaciones Midas, veamos como crearlas. Para desarrollar una aplicación en tres capas es necesario, en primer lugar, crear la aplicación intermedia que actuará como servidor de aplicaciones. De esta manera, la aplicación cliente conocerá la estructura de los datos con que va a trabajar. Es parecido a lo que ocurre en las aplicaciones en una o dos capas: necesitamos que las tablas estén creadas, y de ser posible con datos de prueba, para poder trabajar sobre su estructura.



Partiremos de una aplicación vacía; en principio no necesitamos el formulario principal, pero lo dejaremos con el propósito de controlar la ejecución del servidor. Primero creamos un módulo de datos remoto, por medio del Depósito de Objetos. El icono necesario se encuentra en la página *Multitier* del Depósito, bajo el castizo nombre de *Remote data module*. Al seleccionar el icono y pulsar el botón *Ok*, Delphi nos pide un nombre de clase y un modo de creación de instancias, al igual que para los objetos de automatización que hemos visto en el capítulo anterior:

Como nombre de clase utilizaremos *MastSql*; guardaremos la aplicación como *ServidorApl*, por lo que el identificador de clase que utilizaremos más adelante para referirnos al objeto de automatización será la concatenación de estos nombres: *ServidorApl.MastSql*. Como modo de instancias, utilizaremos *Multiple instance*, de forma tal que cuando se conecten varios clientes al servidor, la aplicación esté ejecutándose una sola vez. Por cada cliente, sin embargo, habrá un módulo remoto diferente, ejecutándose en su propio hilo. Y como estamos creando un servidor remoto, podemos ignorar el modelo de concurrencia.

A primera vista, el módulo incorporado al proyecto tiene el mismo aspecto que un módulo normal, pero no es así. Si abrimos el Administrador de Proyectos, veremos que Delphi ha incluido una biblioteca de tipos en el proyecto. Esto se debe a que el módulo de datos remoto implementa una interfaz, denominada *IDataBroker*, que Delphi define como descendiente de la interfaz de automatización *IDispatch*. Las interfaces se definen en la unidad correspondiente a la biblioteca de tipos:

```
type
   { Dispatch interface for MastSql Object }
   IMastSql = interface(IDataBroker)
      ['{90B69001-22BA-11D1-9412-00A024562074}']
   end;

   { DispInterface declaration for Dual Interface IMastSql }
   IMastSqlDisp = dispinterface
      ['{90B69001-22BA-11D1-9412-00A024562074}']
      function GetProviderNames: OleVariant; dispid 22929905;
   end;
```

Recuerde que esta unidad no debe modificarse directamente, sino a través del Editor de la Biblioteca de Tipos. La que sí podemos modificar es la unidad correspondiente al módulo de datos, en la cual se define el siguiente tipo de clase para el módulo:

```
type
   TMastSql = class(TRemoteDataModule, IMastSql)
   private
      { Private declarations }
   public
      { Public declarations }
   end;
```

Al final de esta unidad, en el código de inicialización, Delphi incluye la instrucción que crea la fábrica de clases:

```
initialization
   TComponentFactory.Create(ComServer, TMastSql,
      Class_MastSql, ciMultiInstance, tmApartment);
end.
```

Para controlar el número de clientes que se conectan al servidor, aprovecharemos que por cada cliente se crea un nuevo objeto de tipo *TMastSql*. En el formulario principal añadimos un componente *Label1*, y una propiedad *Connections*:

```
property Connections: Integer
   read FConnections write SetConnections;
```

La implementación de *SetConnections* es la siguiente:

```
procedure TForm1.SetConnections(Value: Integer);
begin
   FConnections := Value;
   Label1.Caption := Format('%d clientes conectados', [Value]);
end;
```

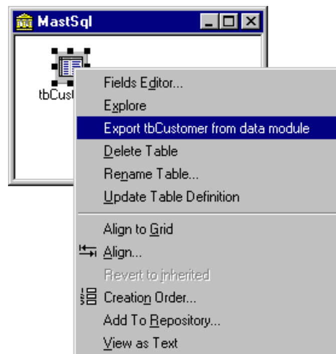
Regresamos al módulo de datos, incluimos la unidad principal en su cláusula **uses** y creamos estos métodos como respuesta a los eventos *OnCreate* y *OnDestroy* del módulo remoto:

```
procedure TMastSql.MastSqlCreate(Sender: TObject);
begin
   Form1.Connections := Form1.Connections + 1;
end;

procedure TMastSql.MastSqlDestroy(Sender: TObject);
begin
   Form1.Connections := Form1.Connections - 1;
end;
```

Proveedores

Las aplicaciones clientes deben extraer sus datos mediante una interfaz remota cuyo nombre es *IProvider*. El propósito del módulo de datos remoto, además de contener los componentes de acceso a datos, es exportar un conjunto de proveedores por medio de la interfaz *IDataBroker*. Por lo tanto, nuestro próximo paso es crear estos proveedores. Comenzaremos con el caso más sencillo: una sola tabla. Más adelante veremos como manejar relaciones *master/detail*, y cómo pueden manejarse transacciones de forma explícita.

Colocamos una tabla en el módulo de datos remotos, cambiamos su nombre a *tbClientes*, su base de datos a *dbdemos* y su nombre de tabla a *customer.db*; después la abrimos mediante su propiedad *Active*. Pulsamos el botón derecho del ratón sobre la tabla y elegimos el comando *Export tbClientes from data module*:



Con esta acción hemos provocado que una nueva propiedad, *tbClientes* de sólo lectura, aparezca en la interfaz del objeto de automatización. La declaración de clase de la interfaz se ha modificado de la siguiente forma:

```
type
   IMastSql = interface(IDataBroker)
      ['{90B69001-22BA-11D1-9412-00A024562074}']
      function Get_tbClientes: IProvider; safecall;
      property tbClientes: IProvider read Get_tbClientes;
   end;
```

La implementación del nuevo método de acceso, *Get_tbClientes*, es responsabilidad del módulo de datos, y se genera automáticamente:

```
function TMastSql.Get_tbClientes: IProvider;
begin
   Result := tbClientes.Provider;
end;
```

Los conjuntos de datos basados en el BDE tienen una propiedad *Provider*, del tipo *IProvider*, que ha hecho posible exportar directamente sus datos desde el módulo remoto. Ahora bien, es preferible realizar la exportación incluyendo explícitamente un componente *TProvider*, de la página *Data Access* de la Paleta de Componentes, conectándolo a un conjunto de datos y exportándolo mediante su menú de contexto. La ventaja de utilizar este componente es la posibilidad de configurar opciones y de crear manejadores para los eventos que ofrece, ganando control en la comunicación entre el cliente, el servidor de aplicaciones y el servidor de bases de datos.

Para deshacer la exportación de *tbClientes*, debemos buscar en primer lugar la Biblioteca de Tipos (*View|Type library*), seleccionar la propiedad *tbClientes* de la interfaz *MastSql* y eliminarla. Después, en la unidad del módulo de datos, hay que eliminar la función *Get_tbClientes*. Traemos entonces un componente *TProvider*, de la página *Midas*, y lo situamos sobre el módulo de datos. Cambiamos las siguientes propiedades:

Propiedad	Valor
Name	*Clientes*
DataSet	*tbClientes*
Options	Añadir *poIncFieldProps*

La opción *poIncFieldProps* hace que el proveedor incluya en los paquetes que envía al cliente información sobre propiedades de los campos, tales como *DisplayLabel*, *DisplayFormat*, *Alignment*, etc. De esta forma, podemos realizar la configuración de los campos en el servidor y ahorrarnos repetir esta operación en sus clientes. Más adelante estudiaremos otras propiedades de *TProvider*, además de sus eventos. Finalmente, pulsamos el botón derecho del ratón sobre el componente y ejecutamos el comando *Export Clientes from data module*.

Una vez que hemos exportado una interfaz *IProvider*, de una forma u otra, podemos guardar el proyecto y ejecutarlo la primera vez; así se registra el objeto de automatización dentro del sistema operativo. Recuerde que para eliminar las entradas del registro, se debe ejecutar la aplicación con el parámetro */unregserver* en la línea de comandos.

Servidores remotos y conjuntos de datos clientes

Es el momento de programar la aplicación cliente. Si el lector no dispone de una red con DCOM configurado de algún modo, no se preocupe, porque podemos probar el cliente y el servidor dentro de la misma máquina. Iniciamos una aplicación nueva, con un formulario vacío, y añadimos un módulo de datos de los de siempre. Sobre este módulo de datos colocamos un componente *TDCOMConnection*, de la página *Midas*, cuyas propiedades configuramos de este modo:

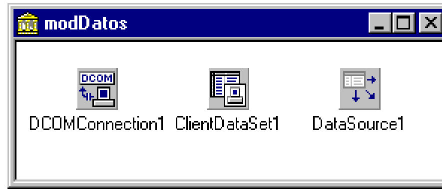
Propiedad	Valor	Significado
Computer		Nombre del ordenador donde reside el servidor. Se deja vacío si es local.
ServerName	*ServidorApl.MastSql*	Identificador de clase del objeto de automatización
ServerGUID		Lo llena automáticamente Delphi
Connected	*True*	Al activarla, se ejecuta el servidor de aplicaciones

Por cada proveedor que exporte el módulo de datos remoto y que nos interese, traemos al módulo local un componente *TClientDataSet*, de la página *Midas* de la Paleta de Componentes. Ya hemos visto en funcionamiento a este componente en el capítulo 38, pero trabajando con datos extraídos de ficheros locales. Ahora veremos qué propiedades, métodos y eventos tenemos que utilizar para que funcione con una interfaz *IProvider* como origen de datos. En concreto, para este proyecto traemos un *TClientDataSet* al módulo local, y asignamos las siguientes propiedades; después traemos un *TDataSource* para poder visualizar los datos:

Propiedad	Valor
RemoteServer	*DCOMConnection1*
ProviderName	*Clientes*
Active	*True*

Hay un par de propiedades que controlan la forma en que el conjunto de datos cliente extrae los registros del servidor de aplicaciones. *FetchOnDemand*, por ejemplo, debe ser *True* (valor por omisión) para que los registros se lean cuando sea necesario; si es *False*, hay que llamar explícitamente al método *GetNextPacket*, por lo que se recomienda dejar activa esta propiedad. *PacketRecords* indica el número de registros que se transfiere en cada pedido. En el caso en que es -1, el valor por omisión, todos los registros se transfieren en la primera operación, lo cual solamente es aconsejable para tablas pequeñas.

A partir de ahí, podemos crear y configurar los objetos de acceso a campo igual que si estuviésemos tratando con una tabla o una consulta. Si me hizo caso e incluyó la opción *poIncFieldProps* en el proveedor, se podrá ahorrar la configuración de los campos. He aquí el módulo de datos local después de colocar todos los componentes necesarios:



Ya podemos traer una rejilla, o los componentes visuales que deseemos, al formulario principal y enlazar a éste con el módulo local, para visualizar los datos extraídos del servidor de aplicaciones.

Grabación de datos

Si realizamos modificaciones en los datos por medio de la aplicación cliente, salimos de la misma y volvemos a ejecutarla, nos encontraremos que hemos perdido las actualizaciones. Las aplicaciones que utilizan *TClientDataSet* son parecidas a las que aprovechan las actualizaciones en caché: tenemos que efectuar una operación explícita de actualización del servidor para que éste reconozca los cambios producidos en los datos.

Ya hemos visto que *TClientDataSet* guarda los cambios realizados desde que se cargan los datos en la propiedad *Delta*, y que *ChangeCount* almacena el número de cambios. En las aplicaciones basadas en ficheros planos, los cambios se aplicaban mediante el método *MergeChangeLog*. Ahora, utilizado proveedores como origen de datos, la primera regla del juego dice:

"Prohibido utilizar MergeChangeLog con proveedores"

Para grabar los cambios en el proveedor, debemos enviar el contenido de *Delta* utilizando la misma interfaz *IProvider* con la que rellenamos *Data*. El siguiente método es la forma más fácil de enviar las actualizaciones al servidor:

```
function TClientDataSet.ApplyUpdates(MaxErrors: Integer): Integer;
```

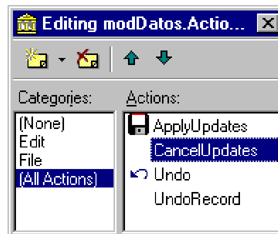
El parámetro *MaxErrors* representa el número máximo de errores tolerables, antes de abortar la operación:

- 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.

Estamos hablando, por supuesto, de Delphi 4. En Delphi 3, el valor –1 indicaba que no se toleraban errores. Otra diferencia importante es que el servidor en Delphi 4 inicia una transacción sobre la base de datos si no hemos iniciado una explícitamente. Por lo tanto, cuando hablamos de abortar la operación, en realidad estamos hablando de realizar un *Rollback* sobre la base de datos en el servidor. Más adelante veremos con detalle qué sucede cuando se producen errores, y cómo podemos controlar estos errores, tanto en el cliente como en el servidor Midas. Además, veremos cómo manejar transacciones más complejas que las simples transacciones automáticas de los proveedores Midas.

¿Cuándo debemos llamar al método *ApplyUpdates*? Depende del modo de edición y de los convenios de interfaz que usted establezca con sus usuarios. Si el usuario utiliza cuadros de diálogo para insertar o modificar registros, es relativamente sencillo asociar la grabación al cierre del diálogo. Las cosas se complican cuando las actualizaciones se realizan directamente sobre una rejilla. Personalmente, soy partidario en tales casos de que el usuario considere a la aplicación como una especie de procesador de textos, y que disponga de un comando *Guardar* para enviar los cambios al servidor. Realmente, ésta es la metáfora más apropiada para los conjuntos de datos clientes, que no mantienen una conexión directa con los datos originales.

Voy a mostrar cómo incluir el comando *Guardar* y los demás comandos de edición, como *Deshacer*, *Deshacer todo* y *Deshacer registro*, en una lista de acciones de Delphi 4. Colocamos un objeto de tipo *TActionList* (página *Standard*) en el módulo de datos, para que esté cerca del conjunto de datos cliente. Podemos traer también una lista de imágenes para asociarle jeroglíficos a las acciones. Estas son las acciones que debemos crear:



El evento *OnUpdate* de las acciones individuales se dispara para que podamos activar o desactivar la acción en dependencia del estado de la aplicación. Las cuatro acciones de nuestro ejemplo se activan cuando hemos realizado cambios sobre el conjunto de datos clientes, por lo que utilizaremos una respuesta compartida para el evento *OnUpdate* de todas ellas:

```
procedure TmodDatos.HayCambios(Sender: TObject);
begin
   TAction(Sender).Enabled := ClientDataSet1.ChangeCount > 0;
end;
```

La respuesta de cada acción se programa en el evento *OnExecute*:

```
procedure TmodDatos.ApplyUpdatesExecute(Sender: TObject);
begin
   ClientDataSet1.ApplyUpdates(0);
end;

procedure TmodDatos.CancelUpdatesExecute(Sender: TObject);
begin
   ClientDataSet1.CancelUpdates;
end;

procedure TmodDatos.UndoExecute(Sender: TObject);
begin
   ClientDataSet1.UndoLastChange(True);
end;

procedure TmodDatos.UndoRecordExecute(Sender: TObject);
begin
   ClientDataSet1.RevertRecord;
end;
```

Estas acciones que hemos preparado son objetos no visuales. Para materializarlas creamos un menú, y en vez de configurar sus comandos trabajosamente propiedad por propiedad, podemos asignar acciones en la propiedad *Action* de cada *TMenuItem*. La misma operación puede efectuarse sobre los botones *TToolButton* de una barra de herramientas.

Es conveniente, para terminar, programar la respuesta del evento *OnCloseQuery*, de forma similar a como lo haríamos en un procesador de textos:

```
resourcestring
   SGuardarCambios = '¿Desea guardar los cambios?';

procedure TwndMain.FormCloseQuery(Sender: TObject;
   var CanClose: Boolean);
begin
   with modDatos.ClientDataSet1 do
      if ChangeCount > 0 then
         case MessageDlg(SGuardarCambios, mtConfirmation,
            mbYesNoCancel, 0) of
            mrNo: { Nada };
            mrCancel: CanClose := False;
         else
            ApplyUpdates(0);
            CanClose := ChangeCount = 0;
         end;
end;
```

Resolución

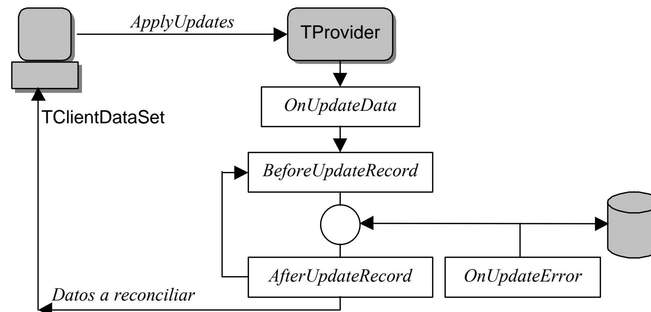
El algoritmo empleado para grabar los cambios efectuados en el cliente mediante el servidor remoto es un ballet muy bien sincronizado y ensayado, cuyas *primas ballerinas* son el cliente y el proveedor, pero en el que interviene todo un coro de componentes. La obra comienza cuando el cliente pide la grabación de sus cambios mediante el método *ApplyUpdates*:

```
ClientDataSet1.ApplyUpdates(0);
```

Internamente, el conjunto de datos echa mano de su interfaz *Provider*, ejecuta el siguiente método remoto, y se queda a la espera del resultado para efectuar una operación que estudiaremos más adelante, llamada *reconciliación*:

```
Provider.ApplyUpdates(Delta, MaxErrors, Result);
```

Como vemos, al servidor Midas se le envía *Delta*: el vector que contiene los cambios diferenciales con respecto a los datos originales. Y ahora cambia el escenario de la danza, pasando la acción al servidor:



El proveedor utiliza un pequeño componente auxiliar, derivado de la clase *TCustomResolver*, que se encarga del proceso de grabar las diferencias en la base de datos. En la jerga de Borland/Inprise, este proceso recibe el nombre de *resolución*. El algoritmo de resolución puede concretarse de muchas formas diferentes. El componente *TProvider*, por ejemplo, aplica las modificaciones mediante sentencias SQL que lanza directamente a la base de datos, saltándose el conjunto de datos que tiene conectado en su propiedad *DataSet*. Así se logra mayor eficiencia, pero se asume que *DataSet* apunta a un conjunto de datos del BDE. El componente *TDataSetProvider*, que también se encuentra en la página *Midas*, aplica las actualizaciones utilizando como intermediario a su conjunto de datos asociado. Es más lento, pero puede utilizarse con cualquier tipo de conjunto de datos. Y si usted tiene la suficiente paciencia, puede crearse su proveedor personalizado, derivando clases a partir de *TBaseProvider* y de *TCustomResolver*.

Veamos la secuencia de eventos que se disparan durante la resolución. El primer evento se activa una sola vez, antes de que comience la acción. Se trata de *OnUpdate-Data*, y puede utilizarse para editar los datos antes de que sean grabados. Su tipo es el siguiente:

```
type
   TProviderDataEvent = procedure(Sender: TObject;
      DataSet: TClientDataSet) of object;
```

El parámetro *DataSet* contiene un conjunto de datos cliente creado al vuelo por el proveedor, que contiene la información pasada en *Delta*. Podemos recorrer este conjunto de datos, a diferencia de lo que sucede en otros eventos que prohiben mover la fila activa. ¿Qué podemos hacer aquí? Tenemos la oportunidad de modificar registros o completar información. El ejemplo que viene en el Manual del Desarrollador muestra cómo llenar un campo con el momento en que se produce la grabación, para los registros nuevos:

```
procedure TRemMod.Provider1UpdateData(Sender: TObject;
   DataSet: TClientDataSet);
begin
   DataSet.First;
   while not DataSet.EOF do
   begin
      if DataSet.UpdateStatus = usInserted then
      begin
         DataSet.Edit;
         DataSet['FechaAlta'] := Now;
         DataSet.Post;
      end;
      DataSet.Next;
   end;
end;
```

Pero también se puede utilizar el evento para controlar el contenido de las sentencias SQL que va a lanzar el proveedor al servidor de datos. Antes de llegar a este extremo, permítame que mencione la propiedad *UpdateMode* del proveedor, que funciona de modo similar a la propiedad homónima de las tablas. Con *UpdateMode* controlamos qué campos deben aparecer en la cláusula **where** de una modificación (**update**) y de un borrado (**delete**). Como expliqué en el capítulo 31, nos podemos ahorrar muchos conflictos asignando *upWhereChanged* a la propiedad, de forma tal que solamente aparezcan en la cláusula **where** las columnas de la clave primaria y aquellas que han sufrido cambios.

¿Necesita más control sobre las instrucciones SQL generadas para la resolución? Debe entonces configurar, campo por campo, la propiedad *ProviderFlags* de los mismos, que ha sido introducida por Delphi 4:

Opción	Descripción
pfInWhere	El campo no aparece en la cláusula **where**
pfInUpdate	El campo no aparece en la cláusula **set** de las modificaciones
pfInKey	Utilizado para releer el registro
pfHidden	Impide que el cliente vea este campo

Analizando fríamente la lista de opciones anterior, vemos que a pesar de la mayor complejidad, no hay muchas más posibilidades reales que las básicas de *UpdateMode*. Además, los nombres de las dos primeras opciones inducen a pensar completamente en lo contrario de lo que hacen. Cosas de la vida.

Después de haber sobrevivido a *OnUpdateData*, el componente de resolución aplica los cambios en un bucle que recorre cada registro. Antes de cada grabación se activa el evento *BeforeUpdateRecord*, se intenta la grabación y se llama posteriormente al evento *AfterUpdateRecord*. *BeforeUpdateRecord* sirve para los mismos propósitos que *OnUpdateData*, pero esta vez los cambios que realizamos afectarán sólo al registro activo. Este es su prototipo:

```
type
   TBeforeUpdateRecordEvent = procedure(Sender: TObject;
      SourceDS: TDataSet; DeltaDS: TClientDataSet;
      UpdateKind: TUpdateKind; var Applied: Boolean) of object;
```

Nuevamente, podemos utilizar este evento para modificar la forma en que se aplicará la actualización. El registro activo de *DeltaDS* es el que contiene los cambios. Por ejemplo, este es un buen momento para asignar claves únicas de forma automática. Pero también podemos optar por grabar nosotros mismos el registro, o eliminarlo. Muchas bases de datos están diseñadas con procedimientos almacenados como mecanismo de actualización. Si durante la respuesta a *BeforeUpdateRecord* ejecutamos uno de esos procedimientos almacenados y asignamos *True* al parámetro *Applied*, el proveedor considerará que el registro ya ha sido modificado, y pasará por alto el registro activo.

Finalmente, después de la grabación exitosa del registro, se dispara el evento *AfterUpdateEvent*, que es similar al anterior, pero sin el parámetro *Applied*:

```
type
   TAfterUpdateRecordEvent = procedure(Sender: TObject;
      SourceDS: TDataSet; DeltaDS: TClientDataSet;
      UpdateKind: TUpdateKind) of object;
```

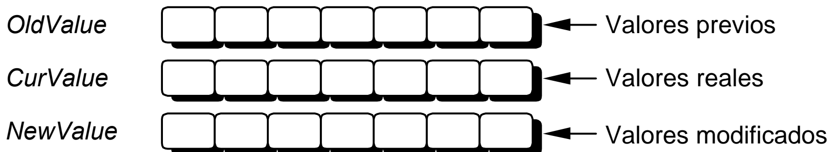
Este evento es de poco interés para nosotros.

Control de errores durante la resolución

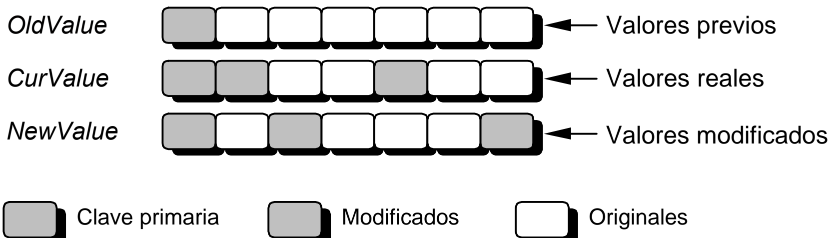
Cada vez que se produce un error durante la resolución se dispara el evento *OnUpdateError*, cuya declaración en Delphi 4 es:

```
type
   TUpdateKind = (ukModify, ukInsert, ukDelete);
   TResolverResponse = (rrSkip, rrAbort, rrMerge, rrApply,
      rrIgnore);
   TResolverErrorEvent = procedure(Sender: TObject;
      DataSet: TClientDataSet; E: EDatabaseError;
      UpdateKind: TUpdateKind; var Response: TResolverResponse)
      of object;
```

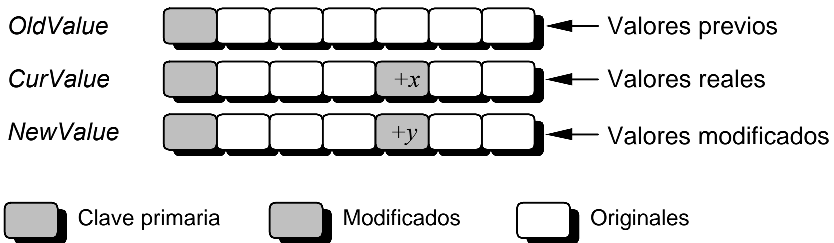
En términos generales, se recomienda realizar la recuperación de errores en las aplicaciones clientes, donde el usuario puede intervenir con mayor conocimiento de causa. Sin embargo, existen situaciones especiales en las que los conflictos pueden resolverse automáticamente en el servidor. Por supuesto, debemos en lo posible tratar estos errores durante la resolución. La corrección de errores se hace más fácil, tanto en la resolución como en la posterior reconciliación, gracias a la existencia de una propiedad asociada a los campos, *CurValue*, que indica cuál es el valor del campo encontrado en el registro de la base de datos, si se produce un fallo del bloqueo optimista. El siguiente esquema representa la relación entre las propiedades *OldValue*, *CurValue* y *NewValue*:



Supongamos por un momento que el valor de *UpdateMode* del proveedor sigue siendo *upWhereAll*. Si dos usuarios realizan actualizaciones en diferentes columnas, sin llegar a afectar a la clave primaria, estamos ante una situación conocida como *actualizaciones ortogonales* (*orthogonal updates*), y se produce un error. Cuando el segundo usuario que intenta grabar recibe el correspondiente error, el estado de los campos puede representarse mediante el siguiente diagrama:



Otro caso frecuente se produce cuando dos aplicaciones diferentes intentan modificar la misma columna, y el valor numérico almacenado en la misma representa un acumulado:



Todos estos tipos de conflicto pueden solucionarse automáticamente dentro del evento *OnUpdateRecord* del servidor. Veamos, por ejemplo, cómo puede resolverse un conflicto con una actualización acumulativa sobre el salario de un empleado. Comenzaré definiendo una función auxiliar que compruebe que solamente se ha cambiado el salario en una actualización:

```
function SalaryChanged(ADataSet: TDataSet): Boolean;
var
   I: Integer;
   Changed: Boolean;
begin
   Result := False;
   for I := ADataSet.FieldCount - 1 downto 0 do
      with ADataSet.Fields[I] do
      begin
         Changed := not VarIsEmpty(CurValue)
            and not VarIsEmpty(NewValue);
         if Changed <> (CompareText(FieldName, 'SALARY') <> 0) then
            Exit;
      end;
   Result := True;
end;
```

La función se basa en que *CurValue* y *NewValue* valen *UnAssigned* cuando el valor que representan no ha sido modificado. Recuerde que *VarIsEmpty* es la forma más segura de saber si un *Variant* contiene este valor especial. La regla anterior tiene una curiosa excepción: si *CurValue* y *NewValue* han sido modificados con el mismo valor, *CurValue* también contiene *UnAssigned*:

```
procedure TMastSql.EmpleadosUpdateError(Sender: TObject;
   DataSet: TClientDataSet; E: EUpdateError;
   UpdateKind: TUpdateKind; var Response: TResolverResponse);
var
   CurSal: Currency;
begin
   if (UpdateKind = ukModify) and SalaryChanged(DataSet) then
```

```
        with DataSet.FieldByName('SALARY') do
        begin
           if VarIsEmpty(CurValue) then
              CurSal := NewValue
           else
              CurSal := CurValue;
           NewValue := CurSal + NewValue - OldValue;
           Response := rrApply;
        end;
  end;
```

Reconciliación

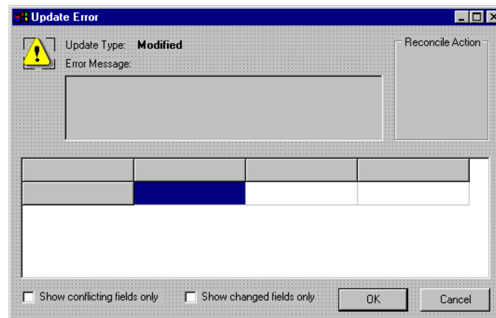
Como todas las modificaciones a los datos tienen lugar en la memoria de la estación de trabajo, la grabación de las modificaciones en el servidor presenta los mismos problemas que las actualizaciones en caché, como consecuencia del comportamiento optimista. El mecanismo de resolución de conflictos, denominado *reconciliación*, es similar al ofrecido por las actualizaciones en caché, y consiste en un evento de *TClientDataSet*, cuyo nombre es *OnReconcileError* y su tipo es el siguiente:

```
type
   TUpdateKind = (ukModify, ukInsert, ukDelete);
   TReconcileAction = (raSkip, raAbort, raMerge, raCorrect,
      raCancel, raRefresh);
   TReconcileErrorEvent = procedure(DataSet: TClientDataSet;
      E: EReconcileError; UpdateKind: TUpdateKind;
      var Action: TReconcileAction) of object;
```

En contraste con las actualizaciones en caché, hay más acciones posibles durante la reconciliación de los datos en este modelo. Los valores del tipo *TReconcileAction* son:

Valor	Significado
raSkip	El registro no se actualiza, pero permanece en la lista de modificaciones.
raAbort	Se aborta la operación de reconciliación.
raMerge	Se mezclan los cambios con los del registro actual.
raCorrect	Se reintenta la grabación.
raCancel	No se actualiza el registro, definitivamente.
raRefresh	Se relee el registro.

La forma más conveniente de reconciliar datos es presentando un cuadro de diálogo de reconciliación, definido como plantilla en la página *Dialogs* del Depósito de Objetos. Para incluirlo en nuestra aplicación, vaya a dicha página del Depósito y seleccione el icono *Reconcile error dialog*. Al proyecto se le añade entonces un formulario de diálogo, con el nombre *TReconcileErrorForm*.



Nadie es perfecto: hay que traducir los mensajes del diálogo. Es importante además utilizar el menú *Project|Options* para eliminar a este formulario de la lista de ventanas con creación automática. Para garantizar que realizamos esta operación, la variable que normalmente aparece en la interfaz, y que se utiliza en la creación automática, ha sido eliminada de la plantilla. La creación y ejecución del diálogo es tarea de la siguiente función, declarada en la interfaz de la unidad asociada:

```
function HandleReconcileError(DataSet: TDataSet;
   UpdateKind: TUpdateKind;
   ReconcileError: EReconcileError): TReconcileAction;
```

El uso típico de la misma durante el evento *OnReconcileError* es como sigue:

```
procedure TDataModule2.tbEmpleadosReconcileError(
   DataSet: TClientDataSet; E: EReconcileError;
   UpdateKind: TUpdateKind; var Action: TReconcileAction);
begin
   Action := HandleReconcileError(DataSet, UpdateKind, E);
end;
```

Si el programador no quiere complicarse demasiado la vida, o no confía en la sensatez de sus usuarios, puede programar un evento de reconciliación más elemental:

```
procedure TDataModule2.tbEmpleadosReconcileError(
   DataSet: TClientDataSet; E: EReconcileError;
   UpdateKind: TUpdateKind; var Action: TReconcileAction);
begin
   ShowMessage('Hay errores');
   Action := raAbort;
end;
```

En tal caso es conveniente que los cambios efectuados queden señalados en la rejilla de datos (si es que está utilizando alguna):

```
procedure TForm1.DBGrid1DrawColumnCell(Sender: TObject;
   const Rect: TRect; DataCol: Integer; Column: TColumn;
   State: TGridDrawState);
begin
   if ClientDataSet1.UpdateStatus <> usUnmodified then
```

```
      DBGrid1.Canvas.Font.Style := [fsBold];
    DBGrid1.DefaultDrawColumnCell(Rect, DataCol, Column, State);
  end;
```

> Si va a utilizar la propiedad *UpdateStatus* de los registros del conjunto de datos clientes, es aconsejable que llame a *Refresh* después de grabar los cambios exitosamente, pues la grabación no modifica el valor de esta propiedad.

Relaciones master/detail y tablas anidadas

Hay dos formas de configurar relaciones *master/detail* en un cliente Midas. Podemos exportar dos interfaces *IProvider* desde el servidor, correspondientes a dos tablas independientes, enganchar a las mismas dos componentes *TClientDataSet* en el módulo cliente, y establecer entonces la relación entre ellos. Haga esto si quiere sabotear la aplicación.

La mejor forma de establecer la relación es configurarla en las tablas del servidor Midas, en el módulo remoto, y exportar solamente el proveedor de la tabla maestra. A partir de Delphi 4, el proveedor puede enviar las filas de detalles junto a las filas maestras (la opción más simple y sensata), o esperar a que el cliente pida explícitamente esos datos, mediante la opción *poFetchDetailsOnDemand* de la propiedad *Options*. Cuando programe el cliente, utilice dos *TClientDataSet*. El primero debe conectarse al proveedor de la tabla maestra. Si trae todos los objetos de acceso a campos del componente, verá que se crea también un campo de tipo *TDataSetField*, con el nombre de la tabla de detalles. Este campo se asigna a la propiedad *DataSetField* del segundo conjunto de datos. Y ya tenemos una relación *master/detail* entre los dos componentes.

Hay dos importantes ventajas al utilizar este enfoque para tablas dependientes. La primera: menos tráfico de red, pues la relación se establece en el servidor, y los datos viajan sólo por demanda. La segunda: en teoría, todas las modificaciones deben acumularse en el *log* del conjunto maestro y deben poder aplicarse en una sola transacción cuando se apliquen las actualizaciones (*ApplyUpdates*) de este conjunto. En teoría. ¿Me promete no matar al mensajero? Bueno, pues esto no funciona muy bien en la práctica, utilizando la versión 4.01. En ocasiones, los cambios se graban en el conjunto anidado en vez de utilizar el conjunto maestro. Y créame que lo siento.

Envío de parámetros

Cuando aplicamos un filtro en un *TClientDataSet* la expresión se evalúa en el lado cliente de la aplicación, por lo cual no se limita el conjunto de registros que se trans-

fieren desde el servidor. Existe, sin embargo, la posibilidad de aplicar restricciones en el servidor Midas si utilizamos la propiedad *Params* del conjunto de datos clientes. La estructura de esta propiedad es similar a la propiedad del mismo nombre del componente *TQuery*, pero su funcionamiento es diferente, ya que requiere coordinación con el servidor de aplicaciones.

Si el proveedor al cual se conecta el conjunto de datos clientes está exportando una consulta desde el servidor, se asume que los parámetros del *TClientDataSet* coinciden en nombre y tipo con los de la consulta. Cuando cambiamos el valor de un parámetro en el cliente, el valor se transmite al servidor y se evalúa la consulta con el nuevo parámetro. Hay dos formas de asignar un parámetro en un conjunto de datos cliente. La primera es cerrar el conjunto de datos, asignar valores a los parámetros deseados y volver a abrirlo:

```
ClientDataSet1.Close;
ClientDataSet1.Params.ParamByName('CIUDAD').AsString :=
   'Cantalapiedra';
ClientDataSet1.Open;
```

La otra es asignar el parámetro sin cerrar el conjunto de datos, y aplicar el método *SendParams*:

```
ClientDataSet1.Params.ParamByName('CIUDAD').AsString :=
   'Cantalapiedra';
ClientDataSet1.SendParams;
```

El método *FetchParams* realiza la acción contraria a *SendParams*: lee el valor actual de un parámetro desde el servidor de aplicaciones.

¿Qué sucede cuando el conjunto de datos que se exporta desde el servidor es una tabla? La documentación de Delphi establece que, en ese caso, el nombre de los parámetros del *TClientDataSet* debe corresponder a nombres de columnas de la tabla, y que la tabla restringe el conjunto de registros activos como si se tratara de un filtro sobre dichos campos. Sin embargo, este humilde programador ha examinado el código fuente (procedimiento *SetParams* del componente *TProvider*), y ha encontrado que Delphi intenta establecer un rango en el servidor. Hay una diferencia clave entre lo que dice la documentación y lo que sucede en la práctica, pues el rango necesita que los campos afectados sirvan de criterio de ordenación activo. A pesar de todo, no he logrado que esta técnica funcione correctamente con el *Update Pack 1* de Delphi 4. Si esto no es flagrante *bug*, yo no me llamo Ian Marteens.

Extendiendo la interfaz del servidor

En Delphi 3, la resolución de las modificaciones efectuadas en el cliente no transcurría, por omisión, dentro de una transacción que garantizara su atomicidad. Esto era

una fuente de problemas cuando editábamos objetos complejos, distribuidos entre varias tablas, como puede suceder al trabajar con una relación *master/detail*. Para resolver esta dificultad, no quedaba más remedio que definir métodos dentro del objeto de automatización y exportarlos, de modo que las actualizaciones en el cliente hicieran uso de los mismos.

Como he dicho antes, Delphi 4 sí utiliza transacciones automáticamente, cuando no las hemos iniciado de forma explícita. Ahora bien, ¿cómo puede una aplicación cliente remota iniciar una transacción mediante objetos situados en el servidor? La respuesta es simple: el módulo de datos remoto implementa, en definitiva, una interfaz de automatización. Si el servidor exporta otros métodos de automatización que los predefinidos, el cliente puede ejecutarlos a través de la siguiente propiedad de la conexión al servidor:

```
property TDispatchConnection.AppServer: Variant;
```

De modo que regresamos al servidor de aplicaciones y abrimos la biblioteca de tipos, con el comando de menú *View|Type library*. En el nodo correspondiente a la interfaz *IMastSql* añadimos tres métodos:

```
procedure StartTransaction; safecall;
procedure Commit; safecall;
procedure Rollback; safecall;
```

En la unidad de implementación del módulo remoto proporcionamos el cuerpo a los métodos anteriores:

```
procedure TMastSql.StartTransaction;
begin
   with tbClientes.Database do
   begin
      if not IsSqlBased then TransIsolation := tiDirtyRead;
      StartTransaction;
   end;
end;

procedure TMastSql.Commit;
begin
   tbClientes.Database.Commit;
end;

procedure TMastSql.Rollback;
begin
   tbClientes.Database.Rollback;
end;
```

Ahora desde el cliente podemos ejecutar secuencias de instrucciones como la siguiente:

```
// …
DComConnection1.AppServer.StartTransaction;
try
   // Aplicar cambios en varios conjuntos de datos
   DComConnection1.AppServer.Commit;
except
   DComConnection1.AppServer.Rollback;
   raise;
end;
// …
```

La técnica anterior tiene muchas aplicaciones, como poder activar procedimientos almacenados situados en el módulo remoto desde las aplicaciones clientes. Más adelante veremos cómo y cuándo aprovechar las interfaces duales para acelerar las llamadas al servidor.

Para aquellos lectores que deben seguir un tiempo más con Delphi 3, incluyo un método para aplicar cambios dentro de una transacción que se utilizaba con esa versión. El método, que añadiremos a la interfaz *IMastSql*, debía tener el siguiente prototipo:

```
procedure ApplyTransaction(var Data: OleVariant);
```

Data es un *buffer* con los cambios realizados en un conjunto de datos clientes; cuando el cliente quiera grabar sus cambios debe pasar el contenido de su propiedad *Delta* en el mismo. Dentro de la definición de clase del módulo de datos se crea automáticamente una entrada para este método, para el que suministramos la siguiente implementación:

```
procedure TMastSql.ApplyTransaction(var Data: OleVariant);
var
   ErrorCount: Integer;
begin
   with tbClientes.Database do
   begin
      if not IsSqlBased then TransIsolation := tiDirtyRead;
      StartTransaction;
      try
         if not VarIsNull(Data) then
         begin
            Data := Clientes.ApplyUpdates(Data, 0, ErrorCount);
            if ErrorCount > 0 then
               SysUtils.Abort;
         end;
         Commit;
      except
         Rollback;
         // ¡¡¡No hay raise!!!
      end;
   end;
end;
```

Terminadas las modificaciones en el servidor de aplicaciones, pasamos a la aplicación cliente. Seleccionamos el módulo de datos de la misma, y añadimos el siguiente método:

```
procedure TDataModule2.ApplyUpdates;
var
   Data: OleVariant;
begin
   ClientDataSet1.CheckBrowseMode;
   if ClientDataSet1.ChangeCount > 0 then
      Data := ClientDataSet1.Delta
   else
      Data := Null;
   // ¡ESTO ES DELPHI 3!
   RemoteServer1.AppServer.ApplyTransaction(Data);
   if not VarIsNull(Data) then
      ClientDataSet1.Reconcile(Data)
   else
   begin
      ClientDataSet1.Reconcile(Data);
      ClientDataSet1.Refresh;
   end;
end;
```

La ejecución de *CheckBrowseMode* garantiza que se graben localmente los registros que estén todavía en fase de edición o inserción. A continuación se prepara el *buffer* con los cambios realizados, en la variable local *Data*. Para ejecutar el método del servidor remoto, se utiliza la propiedad *AppServer* del componente *TRemoteServer* de Delphi 3. Este componente se sigue incluyendo en Delphi 4 por compatibilidad con las versiones anteriores, pero se ha vuelto obsoleto. Luego hay que realizar la reconciliación en el cliente, en función de los valores retornados en el *buffer* de datos. Por último, solamente cuando no se han detectado errores, se actualizan los datos del cliente mediante una llamada a *Refresh*.

La metáfora del maletín

Paco McFarland, empleado de la mítica Anaconda Software, inicia todos los lunes por la mañana desde casa, al terminar de cepillarse los dientes, una sesión de acceso remoto telefónico entre su portátil y el servidor de aplicaciones de la compañía. Una vez establecida la conexión, ejecuta una aplicación que accede a los datos de los clientes de Anaconda y los guarda en el disco duro del portátil; a continuación, y aunque la línea la paga la empresa, corta el cordón umbilical de la conexión. El trabajo de Paco consiste en realizar visitas programadas a determinados clientes y, como resultado de las mismas, actualizar los datos correspondientes en el portátil. Al terminar el día, y antes de ingerir sus vitaminas y analgésicos, Paco vuelve a enchufar el portátil al servidor de aplicaciones e intenta enviar los registros modificados al ordenador de la empresa. Por supuesto, es posible que encuentre conflictos en determinados registros, pero para esto cuenta con técnicas de reconciliación apropiadas.

La historia anterior muestra en acción el "modelo del maletín" (*briefcase model*). Este modelo puede implementarse en Delphi utilizando el mismo componente *TClientDataSet*. Para soportar el modelo, el componente ofrece los métodos *LoadFromFile* y *SaveToFile*; estas rutinas trabajan con ficheros "planos" en formato ASCII. Una aplicación cliente puede incluir opciones para guardar sus datos en un fichero de este tipo y para recuperar los datos posteriormente, sobre todo en el caso en que no se detecta un servidor de aplicaciones al alcance del ordenador.

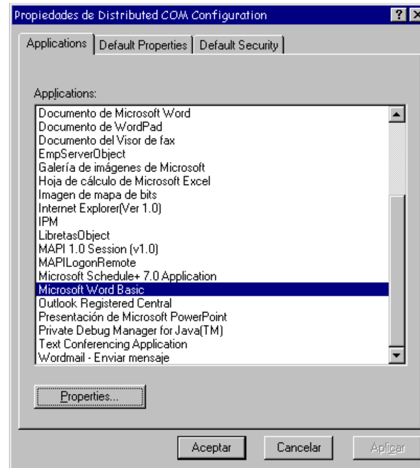
Tipos de conexión

Todos los ejemplos anteriores utilizan el componente *TDCOMConnection* para establecer la comunicación con el servidor Midas. He utilizado dicho componente como punto de partida pensando en el lector que realiza sus pruebas cómodamente en casa, en su ordenador personal, y que al no disponer de una red para probar técnicas sofisticadas de comunicación, sitúa al cliente Midas y a su servidor en el único nodo disponible. Veamos ahora los distintos tipos de conexiones que pueden establecerse y los problemas de configuración de cada uno de ellos.

El primer tipo de conexión se basa en COM/DCOM. Ya hemos visto que cuando el cliente y el servidor residen en la misma máquina es éste el tipo de protocolo que utilizan para comunicarse, y no hace falta ninguna configuración especial. Así que vamos directamente a DCOM. Para poder utilizar este protocolo, debe tener los siguientes elementos:

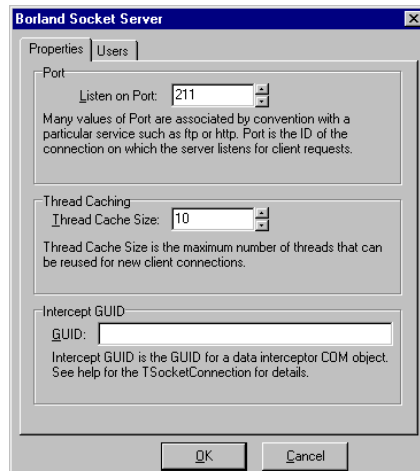
- Una red Windows/Windows NT en la que los ordenadores controlen el acceso a los recursos compartidos mediante el nombre de usuarios.
- Si las estaciones de trabajo tienen instalado Windows 95/98, el requisito anterior obliga a que estén conectadas a un dominio NT. Así que, en cualquier caso, necesitará un ordenador que actúe como controlador de dominio de Windows NT.
- Los ordenadores que tienen Windows 95 necesitan un parche de Microsoft para activar DCOM, tanto si van a actuar como clientes o como servidores. Si se trata de Windows 98, NT Server o Workstation, el soporte DCOM viene incorporado.

Una de las rarezas de DCOM es que no se configura, como sería de esperar, mediante el Panel de Control. Hay que ejecutar, desde la línea de comandos o desde *Inicio|Ejecutar*, el programa *dcomcnfg.exe*. La versión para Windows 95 permite activar o desactivar DCOM en el ordenador local, principalmente:



La versión de *dcomcnfg* para Windows NT se utiliza para controlar la seguridad del acceso a las aplicaciones situadas en el servidor. El control de acceso es la principal ventaja que ofrece DCOM respecto a otros protocolos. Sin embargo, a veces es un poco complicado echar a andar DCOM en una red existente, configurada de acuerdo a las prácticas viciosas y perversas de algunos "administradores de redes".

En tales casos, la forma más sencilla de comunicación es utilizar el propio protocolo TCP/IP, instalando en el ordenador que contiene el servidor Midas una aplicación que actúa como *proxy*, o delegada, y que traduce las peticiones TCP/IP en llamadas a métodos COM dentro del servidor. Esta aplicación se llama *scktsrvr.exe*, y se encuentra en el directorio *bin* de Delphi:

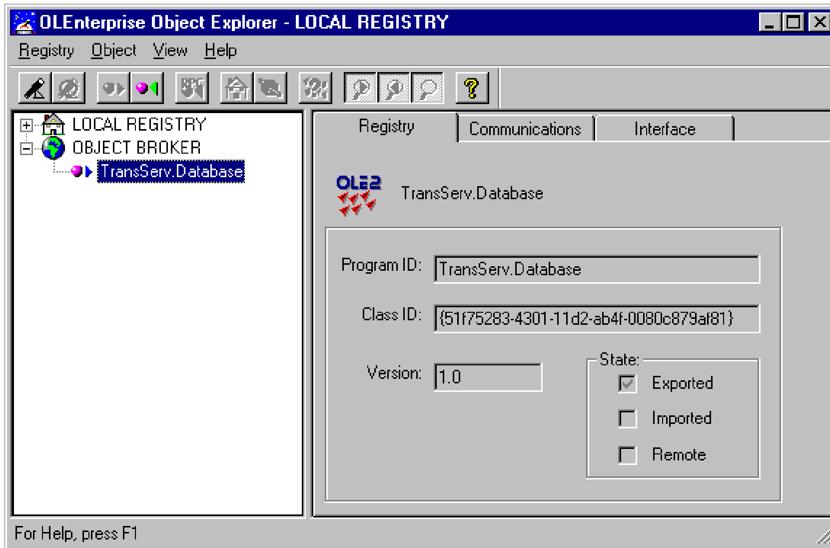


La aplicación al ejecutarse se coloca en la bandeja de iconos; la imagen anterior corresponde en realidad al diálogo de propiedades de la misma. También hay una versión, *scktsrvc.exe*, que se ejecuta como un servicio en Windows NT. Si queremos utilizar TCP/IP con Midas, debemos ejecutar cualquiera de estas dos versiones antes de conectar el primer cliente al servidor. Da lo mismo que la máquina que contiene el servidor ejecute Windows 95, 98 o NT. ¿Qué modificaciones debemos realizar en las aplicaciones clientes y servidoras para que se conecten vía TCP/IP? En el servidor, ninguna. En el cliente debemos sustituir el componente *TDCOMConnection* por un *TSocketConnection*. El uso de este tipo de conexión puede disminuir el tráfico en la red. DCOM envía periódicamente desde el servidor a su cliente notificaciones, que le permiten saber si sus clientes siguen activos y no se ha producido una desconexión. Así pueden ahorrarse recursos en caso de fallos, pero las notificaciones deben viajar por la red. *TSocketConnection* evita este tráfico, pero un servidor puede perder a un cliente y no darse cuenta. Este tipo de conexión tampoco soporta el control de acceso basado en roles de DCOM.

OLEnterprise es el tercer mecanismo de comunicación. Está basado en RPC, funciona en cualquiera de las variantes de Windows, y permite implementar de forma fácil estas tres deseables características:

- Transparencia de la ubicación: El cliente no tiene por qué conocer el ordenador exacto donde está situado su servidor. Todo lo que tiene que especificar es el nombre del servicio (*ServerGUID*) y el nombre del ordenador donde se ejecuta el Agente de Objetos (*Object Broker*). Este último es el software central de OLEnterprise, que lleva un directorio de servicios para toda la red.
- Balance de carga: Gracias a su arquitectura, el Agente de Objetos puede decidir a qué servidor se conecta cada cliente. Al principio de este capítulo mencionábamos la posibilidad de habilitar una batería de servidores Midas redundante para reducir el tráfico en red. Bien, el balance de carga de OLEnterprise es la técnica más completa y segura para aprovechar esta configuración.
- Seguridad contra fallos: El uso de una batería de servidores redundantes permite que, cuando se cae un servidor, sus clientes puedan deshacer las transacciones pendientes, si es que existen, y conectarse a otro servidor de la lista.

La gran desventaja de OLEnterprise es que debemos instalarlo tanto en los servidores como en los clientes, y se trata de una instalación bastante pesada. La imagen de la página siguiente corresponde al *Object Explorer*, la herramienta de OLEnterprise que permite registrar para su uso global a las aplicaciones situadas en los servidores de capa intermedia:



Por último, el componente *TCorbaConnection* permite la conexión a un módulo de datos CORBA. En esta edición del libro, sin embargo, no estudiaremos esta tecnología.

Balance de carga simple

Con Delphi 4 no hace falta OLEnterprise para disponer de un mecanismo sencillo de balance de la carga de los servidores. La técnica está basada en el componente *TSimpleObjectBroker*, que es un descendiente de la clase más general *TCustomObjectBroker*. La idea consiste en especificar una lista de servidores, dando sus nombres o sus direcciones IP, dentro de la propiedad *Servers* de un *TSimpleObjectBroker*. Los componentes de conexión, como *TDCOMConnection* y *TSocketConnection*, en vez de especificar el nombre o dirección de un ordenador deben conectarse al agente por medio de la propiedad *ObjectBroker*.

¿Qué sucede cuando se intenta realizar la conexión? El componente de conexión pide un servidor al agente de objetos, y éste devuelve un servidor de la lista de acuerdo a su propiedad *LoadBalanced*. Si vale *True*, el servidor se elige aleatoriamente. En caso contrario, se selecciona el primer servidor de la lista. ¿Qué sentido tiene entonces utilizar este componente? Bien, en vez de confiar en el azar, usted puede configurar el orden de la lista de servidores en un cliente determinado, una vez instalada la aplicación. Por ejemplo, los ordenadores del departamento comercial se deben conectar siempre a *SVENTAS*, mientras que Investigación y Desarrollo siempre se conectará a *SID*. Por supuesto, esto implica desarrollar código especial para poder realizar esta reconfiguración persistente en tiempo de ejecución.

Interfaces duales en Midas

Si el mecanismo de comunicación que utilizamos es DCOM, podemos utilizar la interfaz dual del módulo remoto en el cliente para acelerar las llamadas a las extensiones exportadas por el módulo, en vez de utilizar la interfaz *IDispatch* que devuelve la propiedad *AppServer* de *TDCOMConnection*. Si nuestro módulo implementaba *IMastSql*, podíamos realizar llamados de la siguiente forma:

```
var
   MastSql: IMastSql;
begin
   MastSql := DComConnection1.AppServer as IMastSql;
   MastSql.StartTransaction;
   try
      // Aplicar cambios en varios conjuntos de datos
      MastSql.Commit;
   except
      MastSql.Rollback;
      raise;
   end;
end;
```

Si el protocolo no es DCOM, no podemos realizar esta mejora al algoritmo, pues OLEnterprise y el Servidor de Sockets se limitan a realizar el *marshaling* para la interfaz *IDispatch*. Sin embargo, algo podemos hacer: utilizar la interfaz *IMastSqlDisp*, que ha sido definida en el módulo remoto como un tipo **dispinterface**:

```
var
   MastSql: IMastSqlDisp;
begin
   MastSql := DComConnection1.AppServer;
   MastSql.StartTransaction;
   // …
end;
```

Como esta interfaz asocia códigos de identificación a los métodos, evitamos el uso de *GetIDOfNames* para recuperar estos códigos, aunque estemos obligados a seguir utilizando implícitamente a *Invoke*. Algo es algo.

Capítulo

42

Creación de instalaciones

EL TOQUE FINAL DEL DESARROLLO de una aplicación es la generación de un programa para su instalación. Para poder distribuir aplicaciones de bases de datos creadas mediante Delphi necesitamos distribuir también el Motor de Datos de Borland, y ello implica la necesidad de configurar este motor de la forma más automática posible. También podemos querer distribuir la base de datos inicial con la que trabaja la aplicación, más los ficheros de ayuda y la documentación en línea. A partir de Delphi 3 se hace necesario también distribuir los paquetes, para las aplicaciones que hacen uso de ellos, lo que añade más complejidad a una instalación típica.

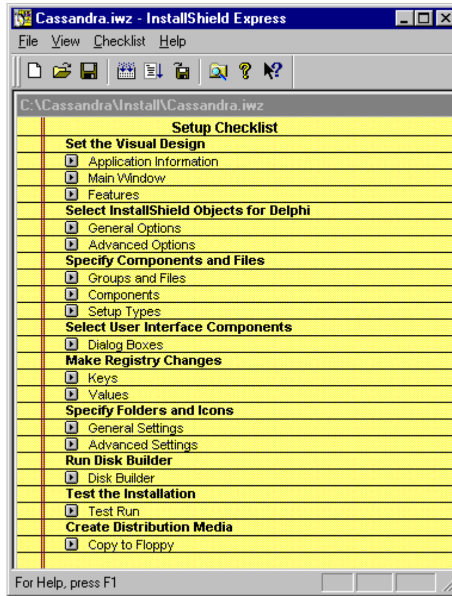
En Delphi 1 todas estas tareas tenían que ser realizadas por el programador, o había que comprar un instalador ajeno a Delphi. La parte más complicada era la instalación y configuración automática del BDE. Aunque el CD-ROM incluía un instalador del BDE, que podía ejecutarse desde una instalación personal, la configuración posterior del motor de datos debía realizarse mediante llamadas de bajo nivel al API del BDE, o dejarse bajo la responsabilidad del usuario.

Afortunadamente, a partir de Delphi 2, Borland/Inprise incluye en el CD-ROM una versión reducida del programa de creación de instalaciones InstallShield. A continuación estudiaremos cómo crear instaladores con esta aplicación. Al final del capítulo mencionaré las posibilidades de la versión completa de este producto.

Los proyectos de InstallShield Express

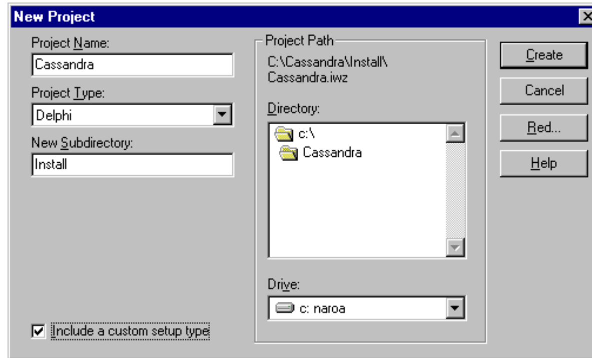
InstallShield no se instala automáticamente junto con Delphi, en ninguna de las versiones de éste. En las versión 3 y 4 de Delphi, se debe ejecutar la opción correspondiente del programa de instalación global, que aparece al insertar el CD-ROM en el ordenador. En Delphi 2, hay que ejecutar el programa *setup*, del subdirectorio *isxpress\disk1*. Una vez que el programa se ha copiado en el ordenador, podemos acceder al mismo por medio de un acceso directo que se coloca directamente en el menú *Inicio|Programas*, de Windows.

InstallShield Express nos permite generar las instalaciones mediante una serie de diálogos de configuración en los cuales debemos indicar qué ficheros debemos copiar, qué parámetros de Windows debemos cambiar y que interacción debe tener el programa de instalación con el usuario. Hay que aclarar este punto, porque existen versiones de InstallShield en las cuales las instalaciones se crean compilando un *script* desarrollado en un lenguaje de programación al estilo C. Evidentemente, el trabajo con estas versiones es mucho más complicado.



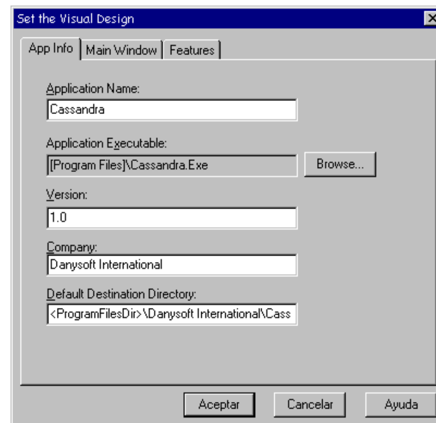
Para comenzar a trabajar con InstallShield necesitamos crear un *proyecto*, que contendrá los datos que se suministrarán al generador. Los proyectos se almacenan en ficheros de extensión *iwz*. Cuando se crea un proyecto nuevo, con el comando de menú *File*|*New*, hay que indicar el nombre y el directorio donde queremos situar este fichero. Si el directorio no existe, podemos crearlo tecleando su nombre en el cuadro de edición *New subdirectory*. También podemos abrir un proyecto existente, con el comando *File*|*Open*, o seleccionándolo de la lista de los últimos proyectos cargados, del menú *File*.

En el cuadro de diálogo de creación de proyectos hay una casilla importante, con el título *Include a custom setup type*. Debemos marcar esta casilla si queremos que el usuario de la instalación pueda seleccionar los componentes de la aplicación que quiere instalar y los que no. Si no incluimos esta posibilidad al crear el proyecto, se puede indicar más adelante, pero nos costará más trabajo hacerlo.



La presentación de la instalación

Una vez creado el fichero de proyecto, los primeros datos que InstallShield necesita son los datos generales de la aplicación y los relacionados con la presentación del programa de instalación. Los datos de la aplicación se suministran en la sección *Application information*, en la que se nos piden los siguientes datos:

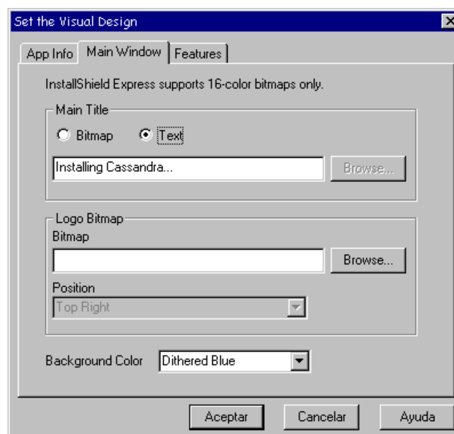


Dato	Observaciones
Application Name	El nombre de la aplicación. Puede ser diferente del nombre del ejecutable, e incluye la ruta.
Application Executable	El nombre del fichero y su ubicación en el ordenador en que estamos trabajando.
Version	El número de versión de nuestra aplicación.
Company	El nombre de nuestra compañía.
Default Destination Directory	El directorio de instalación por omisión.

Los datos sobre el nombre de la aplicación, la empresa y la versión se utilizan para deducir otros parámetros de la instalación. Por ejemplo, el nombre del directorio de instalación inicial se construye a partir del directorio de programas de Windows, más el nombre de la empresa seguido por el nombre de la aplicación. Los datos de la aplicación, por otra parte, se almacenan en una clave del registro de configuración de Windows que se construye de forma similar, incluyendo además la versión. De esto hablaremos más adelante.

En la página *Main Window*, del mismo cuadro de diálogo, se ajustan las propiedades visuales de la ventana de presentación del instalador. Las propiedades configurables son tres: un texto, que aparece en cursivas en la esquina superior izquierda de la ventana, un logotipo, del que podemos indicar su posición, y el color de fondo de la ventana. El texto de la presentación es, casi siempre, el mismo nombre que le hemos dado a la aplicación (*Application Name*), y recomiendo dejar el fondo de la ventana con el típico gradiente azul, a no ser que su aplicación se ocupe de la historia de los enanitos verdes, de Eric El Rojo, o del Submarino Amarillo de los Beatles.

En la opción *Logo Bitmap* podemos indicar la ubicación de un fichero de imagen, que puede ser un mapa de bits o un metafichero (extensiones *bmp* y *wmf*). Los gráficos que podemos utilizar con la versión de InstallShield para Delphi están limitados a 16 colores. Realmente, el código necesario para mostrar mapas de bits de 256 colores es un poco complicado, pues hay que tener en cuenta la existencia de paletas asociadas a los gráficos, y la posibilidad de que dos gráficos diferentes, con paletas disímiles, estén en pantalla al mismo tiempo. Tenemos una foto de un coche y una foto de una chica. La primera imagen utiliza una paleta en la que abundan los tonos metálicos mientras que en la segunda predominan suaves matices de rosa, marrón, amarillo y azul. En un sistema limitado a 256 colores, cuando ambas imágenes se visualicen simultáneamente sucederá que la chica parecerá un coche recién pintado, o que el coche mostrará unos sospechosos tonos carnales.



Por último, en la página *Features* tenemos una casilla para indicar si queremos la desinstalación automática de nuestra aplicación. No conozco motivo alguno para desactivar esta casilla.

Las macros de directorios

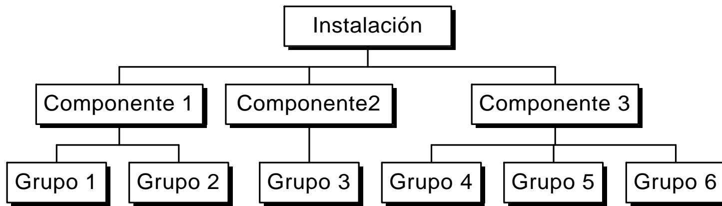
En muchas de las opciones que veremos a continuación que se refieren a directorios del ordenador donde se instala la aplicación, se pueden utilizar *macros* de directorios. Estas son nombres simbólicos, o variables de cadenas, y nos evitan las dependencias que causan los nombres absolutos. Por ejemplo, diferentes ordenadores pueden tener instalado Windows 95 en directorios diferentes. Para no tener que teclear *c:\windows* en una opción y hacer dependiente la instalación de este directorio, se puede utilizar la macro <*WINDIR*>, que al ejecutarse la instalación es sustituida por el nombre del directorio donde reside el sistema operativo en el ordenador de destino.

Las macros de directorios de InstallShield Express son las siguientes:

Macro	**Significado**
<*INSTALLDIR*>	El directorio de instalación, suministrado por el usuario.
<*WINDIR*>	El directorio principal de Windows.
<*WINSYSDIR*>	El directorio de sistema de Windows.
<*WINDISK*>	El disco del directorio principal de Windows.
<*WINSYSDISK*>	El disco del directorio de sistema de Windows.
<*WINSYS16DIR*>	El directorio donde residen las DLLs de 16 bits.
<*ProgramFilesDir*>	El directorio de los archivos de programa.
<*CommonFilesDir*>	El directorio de los archivos comunes.

Grupos y componentes

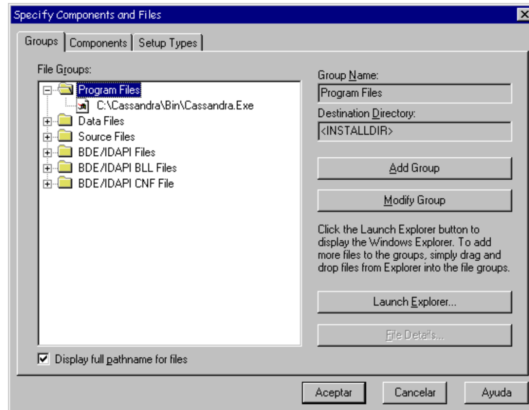
La parte más importante de la especificación de un proyecto de InstallShield se configura en la sección *Specify Components and Files*. Aquí es donde se indican los ficheros que se copian durante la instalación y en qué directorios del ordenador de destino se deben instalar. También se pueden agrupar los ficheros en grupos y componentes, de modo que el usuario pueda escoger cuáles de estos grupos deben ser instalados y cuáles no. InstallShield agrupa el conjunto de ficheros a instalar según el siguiente esquema:



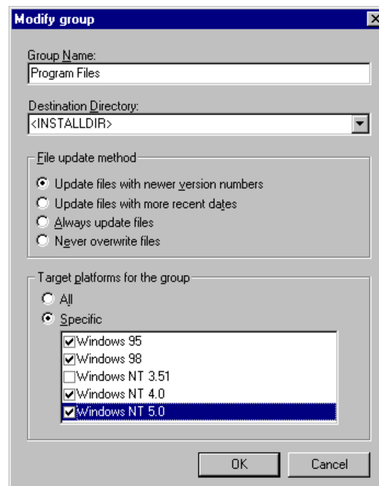
Un grupo está formado por una serie de ficheros que se instalan todos en el mismo directorio, mientras que un componente es una colección de grupos. Por ejemplo una aplicación determinada puede estar constituida por dos componentes: los ficheros de la aplicación y los ficheros de soporte. Los ficheros de la aplicación pueden estar divididos en tres grupos: los ejecutables, propiamente hablando, que residirían en el directorio raíz de la instalación, los ficheros de ayuda, que podrían ubicarse en un subdirectorio llamado *ayuda*, y las DLLs necesarias, que se copiarían al directorio de Windows del ordenador. Por su parte, los ficheros de soporte pueden repartirse a su vez en grupos: el código fuente del programa en un directorio, documentación adicional en otro, etc.

Los grupos y componentes iniciales de un proyecto dependen de la forma en que éste ha sido creado, de acuerdo al estado de la casilla *Include a custom setup type* del diálogo de creación de proyectos. Si no se permiten instalaciones personalizadas, se crea un único grupo, *Program files*, cuyos ficheros se copian al directorio de instalación. Este grupo forma parte del único componente, *Application files*. Por el contrario, cuando existe la opción de instalar componentes de la aplicación a la medida, se crean tres grupos de ficheros, a modo de orientación: *Program files*, *Help files*, y *Sample files*. No es necesario dividir nuestros ficheros en estos tres grupos, pero para muchas aplicaciones el esquema es válido. Se crean tres componentes a la vez, cada uno con uno de los grupos anteriores: *Application Files*, *Help and Tutorial Files* y *Sample Files*. El único grupo que no está vacío es *Program files*, que contiene el ejecutable de la aplicación, que se ha especificado en la sección *Application information*.

El único método mediante el cual se añadían ficheros a un grupo en las versiones de InstallShield para Delphi 2 y 3 era, cuando menos, curioso: había que arrastrar los ficheros desde el Explorador de Windows hasta la carpeta que representa al grupo. Era divertido observar a programadores que no tenían mucha habilidad con el ratón tratando de hacer diana en el minúsculo icono del grupo, o forcejeando con las ventanas de InstallShield y del Explorador de modo que ambas fueran visibles simultáneamente. El cuadro de diálogo de definición de grupos tiene un botón, *Launch Explorer*, que sirve para ejecutar el Explorador de Windows:



En la versión de InstallShield que viene con Delphi 4 tenemos un botón *Insert files* para añadir ficheros a un grupo de una forma menos arriesgada, aunque sigue conservándose la posibilidad de arrastrar y soltar ficheros. En esta versión se ha añadido la posibilidad de poder especificar qué sucede cuando la instalación detecta que ya existen los ficheros que se quieren copiar al directorio del grupo:



Podemos indicar que los ficheros se actualicen cuando tengan un número de versión más antiguo, o según la fecha, o que se actualicen siempre o nunca. También podemos especificar que un grupo se instale solamente para determinados sistemas operativos.

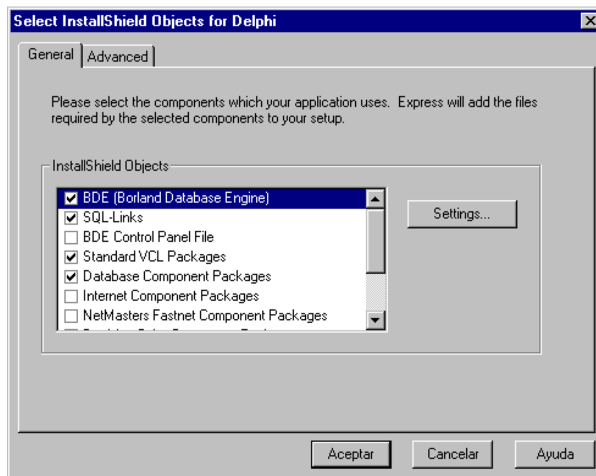
Una vez que se han definido los componentes y grupos de ficheros de una aplicación, debemos configurar los tipos de instalaciones. La versión *Express* de InstallShield solamente permite tres tipos de instalaciones: *Typical*, *Compact* y *Custom*. En el primer tipo se incluyen los componentes de uso más frecuente, en el segundo tipo

de instalación se incluyen los componentes mínimos que necesita la aplicación, mientras que el tipo personalizado incluye inicialmente a todos los componentes, para que el usuario pueda posteriormente configurar la instalación a su gusto.

Instalando el BDE y los SQL Links

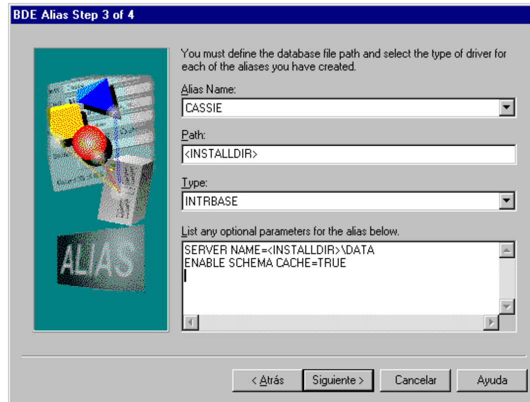
La primera versión de Delphi, que no incluía el generador de instalaciones, ofrecía en un directorio del CD-ROM un par de discos con una instalación "prefabricada" del BDE. El programador debía enlazar su propio instalador con estos discos, ejecutando el programa *setup*, y tenía que ocuparse de la creación de alias persistentes al finalizar la copia de los ficheros. Para ponerlo más difícil, Delphi 1 no tenía funciones de alto nivel para la creación de alias, y estábamos obligados a utilizar nuestras queridas funciones del BDE, aquellas que comienzan con el prefijo *Dbi*.

Con InstallShield, solamente hay que llenar los datos de la sección *General Options*. En la primera página del cuadro de diálogo hay casillas de verificación para indicar cuáles componentes de Delphi hay que incluir con la instalación. En Delphi 2, estos componentes son tres: el Motor de Datos, los SQL Links y las DLLs de ReportSmith. En Delphi 3, que ya había abandonado a ReportSmith, solamente había casillas para los dos primeros componentes. El InstallShield de Delphi 4 añade casillas para instalar los paquetes predefinidos por Borland:



Cuando el programador escoge instalar el BDE, automáticamente se ejecutan los diálogos de configuración de la instalación del motor de datos. El primero de estos diálogos nos permite elegir entre una instalación completa o parcial. Es recomendable efectuar siempre una instalación completa, pues una parcial puede estropear una copia ya instalada del BDE. En el segundo diálogo podemos introducir los nombres

de los alias persistentes que queremos crear, mientras que en el tercero indicamos si deseamos que la configuración nueva esté disponible tanto para las aplicaciones de 16 bits como para las de 32. Por último, por cada alias persistente a crear tenemos que suministrar su configuración: su controlador y sus parámetros, al igual que cuando creamos alias mediante el BDE. En el parámetro *PATH*, y en la lista de parámetros opcionales, se pueden utilizar las macros de directorio.



Aunque un alias SQL no necesita el parámetro *PATH* hace falta especificar "algo" para este valor, pues en caso contrario InstallShield no puede crear el alias. Utilice, por ejemplo, la macro de directorio *<INSTALLDIR>*.

Por su parte, la instalación de los SQL Links nos deja seleccionar con qué formatos deseamos trabajar. En este caso, no hay problemas por no realizar una instalación completa.

En la sección *Advanced Options* podemos ver qué ficheros y grupos se van a instalar con la aplicación Este diálogo no permite realizar modificaciones, pues es solamente informativo. Si el programador hace posibles las instalaciones personalizadas, los grupos que se crean en *General options* no se añaden automáticamente a los componentes creados. Hay que regresar a la sección *Components* para incluir los nuevos grupos dentro de alguno de los componentes existentes.

Configuración adicional del BDE

Lamentablemente, con InstallShield solamente podemos configurar alias para el BDE, mientras que nuestras instalaciones pueden necesitar otros cambios en la configuración global y de los controladores del Motor de Datos. Por ejemplo, las aplicaciones diseñadas para Paradox necesitan configurar los parámetros *NET DIR* y *LOCAL SHARE* para su correcto funcionamiento.

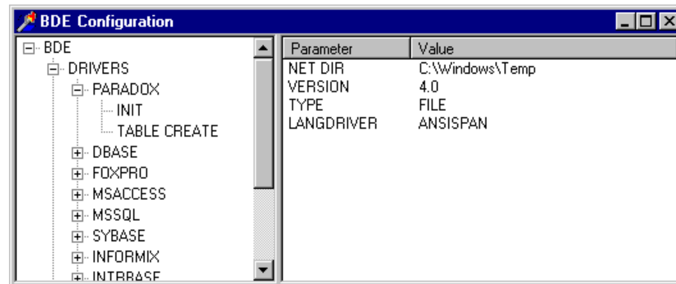
Al final de este capítulo veremos cómo programar DLLs que puedan ejecutarse desde la versión Express de InstallShield (no es la que viene con Delphi). En una de estas extensiones podemos añadir código para configurar el BDE. Si no utilizamos la versión Express, estamos obligados a realizar la configuración de los parámetros de controladores desde nuestra propia aplicación, quizás la primera vez que se inicie.

Y seguimos con problemas, pues Delphi no permite configurar a alto nivel dichos parámetros. Por lo tanto, tenemos que recurrir a llamadas directas al API del BDE. El estudio de esta interfaz no es objetivo de este libro, pero como esta tarea se nos puede presentar con frecuencia, aquí le muestro una unidad que puede ayudarnos a conseguir nuestros objetivos:

```
unit BDEChange;

interface

function GetBDEInfo(const Path, Param: string): string;
procedure SetBDEInfo(const Path, Param, Value: string);

procedure SetLocalShare(const Value: string);
procedure SetNetDir(const Value: string);

implementation

uses BDE, SysUtils;

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;

procedure SetBDEInfo(const Path, Param, Value: string);
var
   hCur: HDbiCur;
   Desc: CFGDesc;
begin
   DbiOpenCfgInfoList(nil, dbiReadWrite, cfgPersistent,
      PChar(Path), hCur);
   try
      while DbiGetNextRecord(hCur, dbiNoLock, @Desc, nil) =
         DBIERR_NONE do
         if StrIComp(Desc.szNodeName, PChar(Param)) = 0 then
         begin
            StrCopy(Desc.szValue, PChar(Value));
            DbiModifyRecord(hCur, @Desc, True);
            Break;
         end;
   finally
      DbiCloseCursor(hCur);
```

```
      end;
   end;

   function GetBDEInfo(const Path, Param: string): string;
   var
      hCur: HDbiCur;
      Desc: CFGDesc;
   begin
      Result := '<Not found>';
      DbiOpenCfgInfoList(nil, dbiReadOnly, cfgPersistent,
         PChar(Path), hCur);
      try
         while DbiGetNextRecord(hCur, dbiNoLock, @Desc, nil) =
            DBIERR_NONE do
            if StrIComp(Desc.szNodeName, PChar(Param)) = 0 then
            begin
               Result := StrPas(Desc.szValue);
               Break;
            end;
      finally
         DbiCloseCursor(hCur);
      end;
   end;

   end.
```

La idea básica es que el BDE organiza sus parámetros en una estructura en forma de árbol. Cada rama del árbol puede verse como una pseudo tabla, cuyos registros consisten en pares parámetro/valor. La siguiente aplicación, cuyo código incluimos en el CD-ROM que acompaña al libro, muestra dicha estructura:

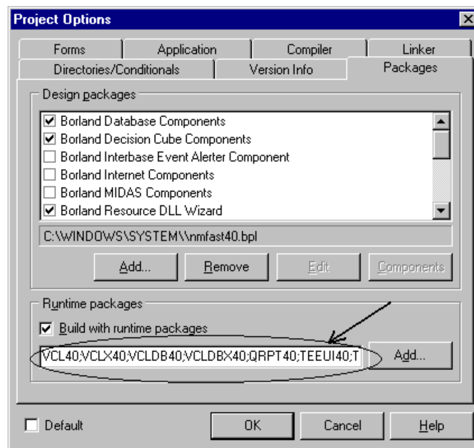


Instalación de paquetes

Como hemos visto en la sección anterior, es muy fácil instalar los paquetes predefinidos con la versión de InstallShield que acompaña a Delphi 4. Sin embargo, podemos vernos en la necesidad de instalar otros paquetes adicionales o estar obligados a seguir un tiempo con Delphi 3.

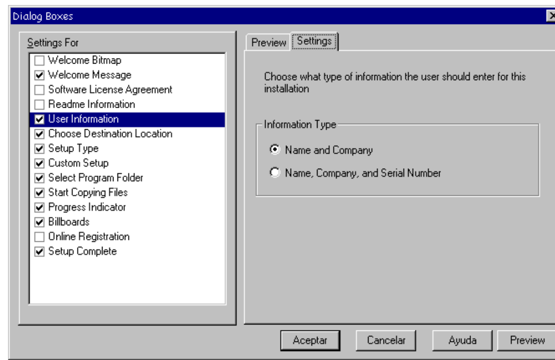
En esta situación, es aconsejable crear un grupo de ficheros y darle un nombre, digamos que *Paquetes*. En su propiedad *Destination Directory* debemos especificar la macro *<WINSYSDIR>*. A este grupo arrastramos los ficheros de paquetes de la versión de Delphi con la que estamos trabajando. ¿Qué ficheros son estos? En Delphi 3, tenemos que buscar los ficheros de extensión *dpl* que se encuentran en el directorio de sistema de Windows. En Delphi 4, los ficheros necesarios tienen la extensión *bpl*. Debe tener cuidado, no obstante, porque los paquetes de C++ Builder comparten la misma extensión. Pero es fácil distinguir entre ellos, pues los paquetes de C++ Builder (para la versión que existe en el momento de la escritura de este libro) incluyen en su nombre un "35", indicando el número de versión de la VCL: *vcl35.bpl*, por ejemplo. Los de Delphi 4 van media versión por delante: *vcl40.bpl*.

Sin embargo, es más fácil determinar cuáles son exactamente los paquetes necesarios si abrimos el diálogo de las opciones de proyecto de Delphi, y nos fijamos en el contenido del cuadro de edición *Runtime packages*:



Interacción con el usuario

La sección *Dialog boxes* permite configurar la mayor parte de los cuadros de diálogos con los que el programa instalador realiza su interacción con el usuario final de nuestra aplicación. En esta sección existen entradas para cada uno de los diálogos típicos de una instalación. En la mayoría de los casos, la configuración consiste en decir si vamos a mostrar o no el diálogo en cuestión; para algunos diálogos, tenemos la posibilidad de configurar los elementos que lo constituyen, o especificar valores iniciales para los datos con que trabajan. Siempre se puede, además, tener una idea del aspecto final del diálogo pulsando el botón *Preview*.



A continuación describiremos los elementos de presentación de InstallShield:

- *Welcome Bitmap*: la imagen de bienvenida.
  No se muestra por omisión. Si se activa este diálogo, hay que especificar un mapa de bits de 16 colores en la página *Settings*.
- *Welcome Message*: el mensaje de bienvenida.
  El contenido de este cuadro de diálogo no se puede cambiar.
- *Software License Agreement*: el acuerdo de licencia de la aplicación.
  No se muestra por omisión. Cuando se activa, el texto a mostrar se extrae de un fichero que se especifica en la pagina *Settings*. El diálogo tiene un botón *No*, por si el usuario decide no aceptar los términos de la licencia.
- *Readme Information*: información antes de instalar.
  No se muestra por omisión. Cuando se activa, el texto a mostrar se extrae de un fichero que se especifica en la pagina *Settings*.
- *User Information*: información acerca del usuario.
  Este diálogo pide el nombre del usuario y la compañía a la que pertenece. Se inicializa con los valores almacenados por Windows en su registro, y los valores suministrados se almacenan en una clave dedicada a la aplicación, como explicaré más adelante. En la pagina *Settings* puede indicarse si queremos, además, el número de serie del programa.
- *Choose Destination Location*: ubicación final de la aplicación.
  Permite especificar el directorio de instalación. En la pagina *Settings* puede cambiarse el directorio por omisión.
- *Setup Type*: tipo de instalación.
  Si queremos que el usuario pueda personalizar su instalación, debemos activar esta opción. Si al crear el proyecto, decidimos incluir esta posibilidad, la casilla correspondiente ya estará activada.
- *Custom Setup*: instalaciones personalizadas.
  Esta casilla se activa y se desactiva en sincronía con la del diálogo anterior; se incluye sólo por compatibilidad con otras versiones de InstallShield. El diálogo que aparece permite al usuario escoger qué componentes desea instalar si ha ele-

gido una instalación personalizada. También permite cambiar el directorio de instalación.

- *Select Program Folder*: seleccionar la carpeta.
  Permite cambiar el nombre de la carpeta donde se colocan los iconos del programa, o elegir una carpeta existente. En la pagina *Settings* se puede indicar el nombre inicial de la carpeta.
- *Start Copying Files*: mensaje de inicio de la copia.
  Este diálogo no es configurable, y muestra los valores que se utilizarán para la instalación, antes de llevarla a cabo.
- *Progress Indicator*: indicador de progreso.
  Es el diálogo que muestra el porcentaje efectuado de la instalación. No es configurable.
- *Billboards*: carteles.
  Mientras InstallShield copia los ficheros en el ordenador, se pueden mostrar imágenes para información o entretenimiento del usuario. En la pagina *Settings* hay que teclear el nombre del directorio donde se encuentran las imágenes, que deben ser mapas de bits o metaficheros de 16 colores (extensiones *bmp* y *wmf*). Los ficheros de imágenes deben, por convenio, nombrarse *setup1.bmp* (ó *wmf*), *setup2.bmp*, etc. InstallShield determina automáticamente los intervalos entre imágenes.
- *Online Registration*: registro en línea.
- *Setup Complete*: instalación terminada.
  Con este diálogo, permitimos que el usuario pueda lanzar uno de nuestros ejecutables al finalizar la aplicación, o que pueda reiniciar el ordenador.

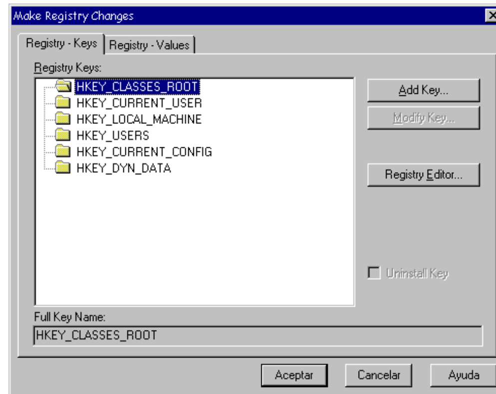
Las claves del registro de Windows

El registro de Windows 95 y Windows NT almacena los datos de configuración de los programas instalados en el sistema, y sustituye a la multitud de ficheros *ini* necesarios en versiones anteriores del sistema operativo. Podemos crear e inicializar secciones y valores en el registro mediante la sección *Make Registry Changes*. Antes de ver cómo podemos añadir claves y valores debemos conocer las claves que InstallShield llena automáticamente en el registro. La más importante de todas es la cadena de desinstalación, que se guarda en la clave siguiente:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\
       Current Version\Uninstall\<Aplicación>
```

Bajo esta clave se almacenan las cadenas *DisplayName* y *UninstallString*. La primera representa el nombre que se muestra en el Panel de Control, en el comando de agregar o quitar programas. La segunda es el comando que hay que ejecutar para eliminar

la aplicación del sistema. El nombre de la aplicación es, por supuesto, el suministrado por el programador en el apartado *Application name*.



Antes de Windows 95, los programas para MS-DOS y Windows 3.x necesitaban modificar el fichero *autoexec.bat* para añadir directorios a la variable *path*. Esto ya no es necesario, pues Windows 95 ofrece una clave especial en el registro, bajo la cual la aplicación que lo desee puede guardar su directorio de ejecución:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\
        Current Version\App Paths\<Aplicación>
```

Por otra parte, si el programador solicita información sobre los datos del usuario, en la sección *Dialog Boxes*, con la opción *User information*, InstallShield almacena los datos que suministra el usuario bajo la clave:

```
HKEY_LOCAL_MACHINE\SOFTWARE\<Compañía>\<Aplicación>\<Versión>
```

Bajo la clave anterior se almacenan las cadenas *User*, *Company* y *Serial*; esta última si ha sido especificada en la configuración de *User information*.

Si nuestra aplicación necesita crear claves y valores adicionales, debemos utilizar la sección *Make Registry Changes*. En la primera página se crean las claves; si la clave no pertenece a las predefinidas por InstallShield, es necesario marcar la casilla *Uninstall key*, para que la clave se borre al desinstalar el programa. Para asignar valores dentro de claves nuevas o ya existentes, utilice la segunda página del cuadro de diálogo.

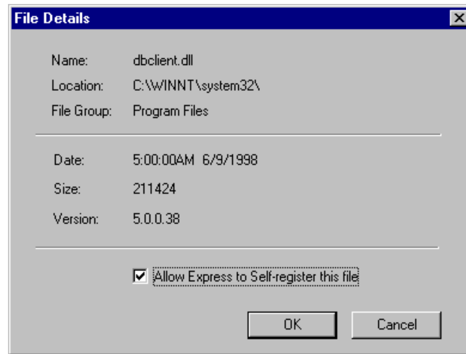
Cómo se registran los componentes ActiveX

Uno de los problemas más frecuentes que plantea una instalación es registrar los servidores COM que necesita la aplicación a instalar, en particular, los controles ActiveX. Como vimos en el capítulo 39, todos los servidores COM son capaces de

registrarse a sí mismos automáticamente. La pregunta es: ¿cómo puede saber InstallShield si determinado ejecutable o DLL necesita ser registrado para su correcto funcionamiento?

InstallShield busca dentro de la información de versión del fichero una cadena con el contenido *OleSelfRegister*. Si la encuentra, el fichero se registra. Recuerde que si se trata de una DLL, debe ejecutarse la función exportada *DllRegisterServer*, y que si es un ejecutable, debe lanzarse con el parámetro */regserver*.

Ahora bien, no todos los servidores COM incluyen dicha cadena en la información de versión. Para estos casos, la versión de InstallShield que viene con Delphi 4 permite indicar, en las propiedades del fichero que se activan en la ventana de grupos y componentes, si queremos o no registrarlo:



Si no dispone de esta versión, puede intentar registrar manualmente el servidor al inicio de su aplicación. Entre los ejemplos de Delphi se encuentra un programa llamado *tregsvr*, que muestra cómo registrar ejecutables y DLLs.

Iconos y carpetas

Para que el usuario final pueda acceder a la aplicación recién instalada, InstallShield crea iconos y carpetas que se incluyen en el menú de inicio de Windows 95 y NT 4, o en el Administrador de Programas de WinNT 3.5x. Por omisión, InstallShield crea un icono para la aplicación especificada en *Application name*, pero se pueden crear todos los iconos necesarios y configurarlos en las secciones *General settings* y *Advanced settings* de *Specify Folders and Icons*.

Para añadir un nuevo icono, teclee el nombre del comando en la primera página del diálogo, en el apartado *Run command*, o seleccione uno de los ficheros de la instalación. En *Run command parameters* hay que indicar los parámetros que se le suministran a la aplicación asociada al icono. Por último, en *Description* se configura el texto asociado al icono. La versión de Delphi 4 permite además indicar el modo de visualización inicial de la aplicación: normal, a pantalla completa o minimizada.

En la segunda página se completan los parámetros de cada icono. Estos son el directorio de trabajo (*Start in*), el icono (*Icon*), y la tecla de abreviatura (*Shortcut key*). También puede indicarse, mediante el grupo de selección *Folder*, en qué carpeta queremos situar el acceso directo. Por ejemplo, si activamos la opción *Start Menu Folder* el icono se situará directamente en el primer nivel del menú *Programas* de Windows. Esto es recomendable solamente cuando la aplicación instala un solo icono, pues en caso contrario se congestiona demasiado este menú desplegable.

Generando y probando la instalación

Cuando todos los parámetros de la instalación han sido suministrados, es hora de crear los discos de la instalación, utilizando la opción *Disk Builder*. Podemos, antes de generar la instalación, indicar el formato en que queremos los discos. InstallShield permite utilizar los siguientes formatos:

Opción	Formato
720KB	Discos de 3.5 pulgadas, simple densidad
1.2MB	Discos de 5.25 pulgadas, doble densidad
1.44MB	Discos de 3.5 pulgadas, doble densidad
2.88MB	Discos de 3.5 pulgadas, cuádruple densidad
120MB	Para discos de mayor capacidad
CD-ROM	Hasta 650MB

En cualquier caso, la escritura no se efectúa directamente sobre el medio seleccionado. El generador de instalaciones crea un subdirectorio bajo el directorio del proyecto, utilizando el nombre del formato. Por ejemplo, si seleccionamos discos de doble densidad de 3.5 pulgadas, el subdirectorio se llamará *144mb*. Bajo este subdirectorio, a su vez, se crearán dinámicamente otros subdirectorios llamados *Disk1*, *Disk2*, etc., con el contenido de cada uno de los discos de instalación. Una vez elegido el formato de los discos, pulsamos el botón *Build* para generar sus contenidos. Se puede utilizar este comando varias veces, con formatos diferentes, si necesitamos distribuir la aplicación en soportes de diferente tamaño.

Finalmente, puede probar la instalación desde la opción *Test Run*. Tenga en cuenta que el programa se instalará realmente en su ordenador, pues no se trata de simular la instalación.

La versión completa de InstallShield Express

InstallShield Express 2, la versión completa del producto que se puede comprar por separado, ofrece características interesantes para el programador. Estas son algunas de ellas:

- Se incluyen módulos adicionales de lenguajes, para cambiar el lenguaje de la instalación. Si quiere esta posibilidad, asegúrese que se trata de la versión 2. La primera versión de InstallShield Express solamente desarrolla instalaciones en inglés.
- Se pueden desarrollar instalaciones para plataformas de 16 y 32 bits.

- Los sistemas de desarrollo soportados son varios: Delphi, Borland C++ y C++ Builder, Visual Basic, Optima++, etc.
- Es posible ejecutar *extensiones* de InstallShield desde la instalación. Estas son DLLs o ejecutables desarrollados por el programador que realizan acciones imposibles de efectuar por InstallShield.
- Cuando se configura un grupo de ficheros se puede indicar la acción a realizar cuando ya existen los ficheros en el ordenador de instalación: sobrescribir siempre, utilizar la versión más moderna (observando la información de versión incorporada, si está disponible) o utilizar los ficheros más nuevos, de acuerdo a su fecha de creación.
- Se pueden modificar los ficheros *autoexec.bat* y *config.sys*. Esto es indispensable en instalaciones para 16 bits.
- Se pueden mezclar ficheros de extensión *reg* en el Registro. Esta técnica permite registrar fácilmente los controles ActiveX.
- Los accesos directos y entradas en el menú que se crean para las aplicaciones tienen más opciones de configuración. Podemos, por ejemplo, indicar que cierta aplicación se ejecute minimizada o maximizada, y colocar una aplicación en otra carpeta, como la de aplicaciones de inicio.
- Se pueden indicar ficheros temporales a la instalación, que se eliminan una vez finalizada la misma, dejar espacio en el primer disco para ficheros que se instalan sin descomprimir, y crear instalaciones autoejecutables desde un solo fichero *exe*.

En mi opinión, la compra de InstallShield Express 2 es una inversión necesaria y rentable para el programador profesional de Delphi.

Las extensiones de InstallShield Express

InstallShield Express permite llamar a ficheros ejecutables y a funciones situadas dentro de DLLs desde una instalación. Desde estos módulos podemos realizar tareas tales como terminar la configuración del BDE, pidiendo quizás información al usuario. Aquí mostraremos un pequeño ejemplo de cómo modificar el parámetro *NET DIR* al concluir una instalación. Por simplicidad, utilizaré un directorio predeterminado para dicho parámetro. El lector puede completar el ejemplo incluyendo un diálogo que permita al usuario seleccionar un directorio.

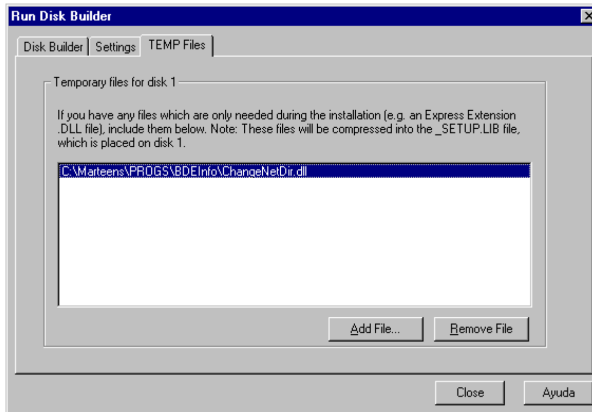
Creamos una DLL en Delphi, e incluimos en la misma la unidad *BDEChange* que hemos mostrado antes. Llamaremos a la DLL *ChangeNetDir*, y crearemos en su fichero principal la siguiente función:

```
function NetDir(HWnd: THandle;
   Source, Support, Install, Reserved: PChar): Char; stdcall;
begin
   Result := #1;
```

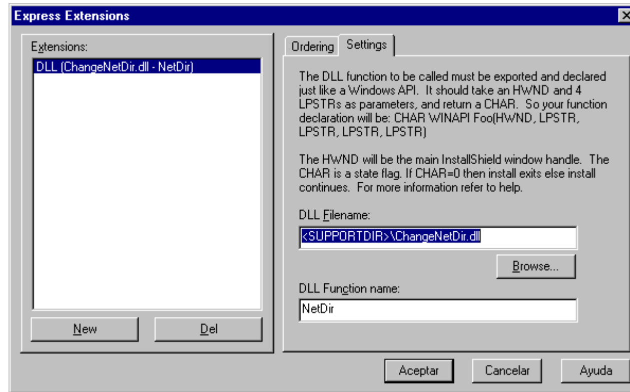
```
    DbiInit(nil);
    try
      SetNetDir('C:\Windows\Temp');
    finally
      DbiExit;
    end;
  end;
```

El prototipo de *NetDir* es el exigido por InstallShield Express para que una función externa pueda ser llamada. El parámetro *HWnd* indica la ventana que debe utilizarse como ventana madre si queremos lanzar algún cuadro de diálogo desde la DLL. *Source*, *Support* e *Install* son los nombres de los directorios utilizados por InstallShield, que no utilizaremos en el ejemplo. La función debe retornar un carácter. Como convenio, si retornamos el carácter nulo se aborta la instalación. Cualquier otro carácter es ignorado. Recuerde que esta función debe ser exportada, mediante la cláusula **exports** de la biblioteca.

Una vez compilada la DLL, debemos incluirla dentro de la instalación. Si se trata de una DLL temporal, que no será utilizada por la aplicación más adelante, lo mejor es comprimirla con el resto de los ficheros temporales de InstallShield. Esto se especifica en el apartado *Disk builder* de la versión Express:



Más adelante, en el apartado *Express extensions*, creamos una nueva extensión, indicando que se trata de una DLL. Tenemos que indicar en qué momento de la instalación se ejecutará la extensión. Lo más indicado es que se ejecute al final, cuando ya se ha instalado el BDE, que la extensión necesita. En la página *Settings* del diálogo de configuración de extensiones debemos indicar el nombre de la DLL y el directorio donde InstallShield debe buscarla. Como se trata de una DLL temporal, que hemos situado junto al núcleo de la instalación, debemos utilizar la macro de directorios *<SUPPORTDIR>*:



Y ya está lista la extensión para su ejecución durante la instalación de nuestra aplicación.

UNAS PALABRAS FINALES

DE NIÑO, DISFRUTABA BUSCANDO la última página de los libros, fundamentalmente novelas, que caían en mis manos. Me gustaba el "tono" especial que adoptaba el autor para dejar la última impresión a sus lectores. Cuando comencé a leer libros técnicos, fundamentalmente física y matemáticas, me angustiaba el invariable tono frío e impersonal del último párrafo; un libro podía terminar con una fórmula.

Por eso he decidido escribir algo aquí al final. Sé que alguna noche me levantaré insomne, arrancaré "La Cara Oculta" de su sitio en la estantería y buscaré en sus páginas postreras un mensaje lanzado por el autor (yo mismo, pero no Yo) en el pasado. Y estas palabras ajenas serán lo que encontraré:

...The time is gone, the song is over,
though I had something more to say...
Pink Floyd

INDICE ALFABÉTICO

A

Abort ·239, 592
Abrazo mortal ·488
abstract ·164
Acciones referenciales ·446
ACID ·635
Active ·304
Actualizaciones en caché ·639, 649
Actualizaciones ortogonales ·631, 862
AddExitProc ·743
AddIndex ·411
AddObject ·423
AddRef ·804
AddTerminateProc ·743
AfterCancel ·587, 844
AfterClose ·593
AfterDispatch ·767
AfterInsert ·587
AfterOpen ·593
AfterPost ·844
AfterPrint ·704
AfterScroll ·586
Agregados ·796
Alias ·110
  local ·110, 604
  persistente ·110
  persistentes ·620
AliasName ·605
Alignment ·373
AlignToBand ·706
AllowGrayed ·362
alter table ·448
AnsiString ·254
Aplicaciones
  icónicas ·184
Aplicaciones multicapas ·107
Append ·571, 575
AppendRecord ·577
ApplyRange ·410
ApplyUpdates ·649, 651, 661
AppServer ·870
as ·169, 271
Aserciones ·244
AsFloat ·329
AsInteger ·329
Assign ·573
Assigned ·209
AsString ·329
AutoCalcFields ·335
AutoDisplay ·363
AutoEdit ·306, 352, 366, 372, 572
automated ·141
AutoSize ·357
AutoStretch ·706

B

Bandas ·699, 703
BatchMove ·549
BeforeAction ·387
BeforeClose ·587
BeforeDelete ·372
BeforeDispatch ·767
BeforeEdit ·587, 589, 844
BeforeInsert ·592
BeforeOpen ·305, 547, 593
BeforePost ·494, 588
BeforePrint ·704, 705
BeforeScroll ·586
BeginUpdate ·620
Bibliotecas de Enlace Dinámico ·68, 733
  Carga dinámica ·745
  Excepciones ·744
  Exportación de funciones ·736
  Importación de funciones ·739
  Proyectos ·735
Bibliotecas de recursos ·749
Bibliotecas de tipos ·817
Bibliotecas de Tipos
  editor ·43
BLOCK SIZE ·89, 113

BlockReadSize ·320
Bloqueos
en transacciones ·641, 650
oportunistas ·114
optimistas ·628
pesimistas ·628
Bookmark ·310
Business rules · *Véase* Reglas de empresa
ButtonStyle ·378

C

CachedUpdates ·540, 650
Campos blob ·367, 388, 440
LoadFromFile ·367
LoadFromStream ·368
SaveToFile ·367
SaveToStream ·368
Campos calculados ·335, 403
internos ·794
Campos de búsqueda ·337, 338, 377, 403
Cancel ·404, 574
CancelRange ·408
CancelUpdates ·653
CanModify ·539, 552
CGI ·761
ChangeCount ·856
check ·443
CheckBrowseMode ·574, 638, 870
Ciclo de mensajes ·49, 177, 191, 234
ápex ·180
Ciclo de reintentos infinito ·596
class ·132, 170
Class completion ·48
Close ·304
CloseIndexFile ·393
CoCreateGuid ·803
CoCreateInstance ·807
Code Explorer ·46
Code Insight ·44
CoInitFlags ·842
Cola de mensajes ·177
Columns ·374, 379
Commit ·636
CommitUpdates ·651, 661
Componentes de impresión ·705
Compresión de tablas ·583
Conjunto de datos
estado ·321
Conjuntos de atributos ·340, 341
Conjuntos de datos
clientes ·299, 785
estados ·321, 571, 585
fila activa ·308
marcas de posición ·310
master/detail ·313
tablas ·303
ConstraintErrorMessage ·345
Constraints ·278
constructor ·144
Constructores ·142, 160, 240
virtuales ·167
Consultas dependientes ·545
Consultas heterogéneas ·538
Consultas paramétricas ·543
contains ·70
Controladores de automatización · 814, 833
Controles de datos ·306
Controles dibujados por el propietario ·358
ControlStyle ·388
CopyToClipboard ·363
CORBA ·800
CoRegisterClassObject ·809
Create ·144, 152, 168
CreateComObject ·807
CreateDataSet ·786
CreateForm ·267, 281
CreateOleObject ·811, 830
CreateProcess ·189
CreateSemafore ·626
CreateTable ·347
CreateWindow ·176
CreateWindowEx ·176
Currency ·249, 403
Cursores ·302
Microsoft SQL Server ·510

Oracle ·524
Cursores unidireccionales ·104, 138
CurValue ·862
CustomConstraint ·344
CutToClipboard ·363

D

Data Migration Wizard ·54
Database Desktop ·53, 457, 464
Database Explorer ·52, 340
DatabaseError ·334, 589
DatabaseName ·303, 537, 604
DatasetCount ·607
Datasets ·607
DayOfWeek ·250
dBase ·91, 392
DBIERR_UNKNOWNSQL ·600
DbiRegenIndexes ·584
DCL ·435
DCOM ·107, 809, 845
DDL ·*Véase* SQL
Decision Cube ·721
DecodeDate ·250
DecodeTime ·250
default ·200, 204
DEFAULT DRIVER ·122, 304
DefaultDrawColumnCell ·379
DefaultExpression ·344, 588
Delegación ·353
delete ·475
Delete ·582, 591, 651
DeleteIndex ·411
DeleteSQL ·667
DeleteTable ·348
Depósito de Objetos ·284, 735, 851, 864
compartido ·287, 296
configuración ·287
Destroy ·144, 152, 160, 243
destructor ·146
Destructores ·142
Diccionario de Datos ·53, 340
dominios ·449
Directivas de compilación
{$H+} ·254
ASSERTIONS ·244
VER80 ·29
DisableControls ·312
DispatchMessage ·179
dispinterface ·832, 875
DisplayFormat ·330, 340
DisplayLabel ·327, 354, 373
DisplayValues ·331
distinct ·461, 468, 471, 539
DLL ·*Véase* Bibliotecas de Enlace Dinámico
DllProc ·743
DML ·*Véase* SQL
Dominios ·343, 448, 673
DRIVER FLAGS ·638
DriverName ·605
DropDownAlign ·362
DropDownRows ·362, 378
DropDownWidth ·362
DroppedDown ·360

E

EAbort ·239, 628
EDatabaseError ·334, 597
EDBEngineError ·597
Edit ·366, 571, 627
EditFormat ·331
EditMask ·331, 355
Editor de bibliotecas de tipos ·818
Editor de Campos ·323, 335, 572
Editor de Código ·44, 66
Editor de Columnas ·374
Editor de Enlaces ·314
EditRangeEnd ·411
EditRangeStart ·411
EDivByZero ·226
EmptyTable ·583
ENABLE INTEGERS ·522
ENABLE SCHEMA CACHE ·115, 562, 611
EnableControls ·312, 318
EncodeDate ·250
EncodeTime ·250
Encuentro natural ·462
Encuentros externos ·473

ENoResultSet ·552
EOutOfMemory ·232
Event log ·51
Eventos
  activación recursiva ·219
  compartidos ·215
  de detección de errores ·586, 595
  de transición de estados ·585
  emisor y receptor ·214
Excepción silenciosa ·239, 592
Exception ·225, 231
ExceptObject ·227, 231
Exclusive ·305, 583
ExecSql ·548, 550
ExitProc ·742
Explorador de Bases de Datos ·*Véase* Database Explorer
Explorador de Código ·*Véase* Code Explorer
Explorador de Objetos ·*Véase* Object Browser
exports ·68, 737
external ·739

F

Fábricas de clases ·809
FetchAll ·651
FetchOnDemand ·855
FetchParams ·867
Ficheros asignados en memoria ·111, 734, 752
FieldByName ·328, 538, 572
FieldDefs ·346, 347, 396, 786
FieldName ·327
Fields ·328, 572
Fields editor ·*Véase* Editor de Campos
FieldValues ·329, 573
FILL FACTOR ·113
Filter ·414
Filtered ·414, 425
FilterOptions ·415
Filtros ·413
  TClientDataSet ·789, 866
finalization ·742
FindComponent ·271
FindField ·328
FindFirst ·425
FindKey ·398, 401
FindLast ·425
FindNearest ·398, 401
FindNext ·425
FindPrior ·425
Flat ·385
FlushBuffers ·594
FocusControl ·354
ForceNewPage ·714
foreign key ·446, 496
FormatDateTime ·251, 331
FormatFloat ·330
FormStyle ·267
Formularios activos ·819
Free ·144, 152, 160, 243
FreeNotification ·199
Funciones de conjuntos ·466
Funciones de respuesta ·174
Funciones definidas por el usuario ·460, 746

G

GetEnumName ·268, 365
GetIDsOfNames ·813
GetIndexNames ·396
GetMessage ·179, 188
GetNextPacket ·855
GetOrdProp ·206
GetPropInfo ·206
GetStrProp ·206
GetTickCount ·596
GotoCurrent ·310, 431
GotoKey ·404
GotoNearest ·404
grant ·453
group by ·465, 467, 539

H

HandleShared ·612
having ·467, 539
Herencia visual ·274, 291
Hint ·385
holdlock ·510, 528

HTML ·758
campos ocultos ·772
etiquetas ·759
HTTP ·757

I

ICopyHook ·826
Image Editor ·55, 384
ImportedConstraint ·344
IndexDefs ·347, 396, 786
IndexFieldNames ·314, 382, 397, 398, 420, 563, 788
IndexFiles ·392
IndexName ·314, 393, 398, 563, 788
Indices ·391
agrupados ·394
creación ·449
de expresiones ·393
de producción ·392
en dBase ·392, 405, 410
en InterBase ·394
en MS SQL Server ·394
en Oracle ·395
en Paradox ·391
reconstrucción y optimización · 451
regeneración ·584
TClientDataSet ·795
InfoPower ·413
inherited ·160, 183, 219, 293
initialization ·63, 742
insert ·475
Insert ·571, 575
InsertRecord ·577
InsertSQL ·667
InstallShield ·73
claves del registro ·890
Extensiones ·895
fichero de proyecto ·878
grupos y componentes ·881
información de la aplicación ·879
instalación ·877
instalación del BDE ·884
macros de directorios ·881
versión profesional ·894
Integridad referencial ·82, 445
Acciones referenciales ·446
propagación en cascada ·591
simulación ·495, 514
InterBase ·95, 394
alertadores de eventos ·498
check ·443
computed by ·441
configuración en el BDE ·117
excepciones ·497
external file ·441
generadores ·493
procedimientos almacenados ·483
roles ·454
set statistics ·451
tipos de datos ·439
usuarios ·451
InterBase Server Manager ·451
interface ·802, 818
Interfaces ·155
IClassFactory ·809, 811
IClassFactory2 ·810
IDataBroker ·812, 846
IDispatch ·812
IProvider ·846
IUnknown ·803, 811
Interfaces duales ·815
Internacionalización ·749
Invoke ·813
IProvider ·787, 853
is ·168, 740
ISAPI ·761
IsEmpty ·309
IShellLink ·823
IsMasked ·355
IsMultiThread ·765, 842
IsNull ·330, 336

K

KeepConnections ·609
KeyExclusive ·408
KeyPreview ·360

L

Ley de Murphy ·223
library ·735

like ·459, 544
Listas de acciones ·857
LoadFromFile ·871
LoadMemo ·357
LoadPicture ·363
Local ·539
LOCAL SHARE ·111, 885
Locate ·419, 422, 564
LockServer ·810
LockWindowUpdate ·275
LogChanges ·791
LoginPrompt ·609
Lookup ·403, 419
Lookup fields · *Véase* Campos de búsqueda
LookupCache ·338

M

Maletín, modelo ·870
Marcas de posición ·310
Marshaling ·809
MasterFields ·313
MasterSource ·313
MAX DBPROCESSES ·119
MaxLength ·356
MaxValue ·334
Mensajes ·176
  CM_GETDATALINK ·389
  EM_UNDO ·177
  WM_ACTIVATE ·217
  WM_CLOSE ·217
  WM_DESTROY ·177, 181
  WM_ERASEBKGND ·182
  WM_GETMINMAXINFO ·278
  WM_PAINT ·177, 178, 182, 282
  WM_QUIT ·181, 190
  WM_SIZE ·182, 217
  WM_TIMER ·178
  WM_USER ·185
MergeChangeLog ·793
message ·182
Métodos de clase ·170, 274
Métodos virtuales ·155, 164
Microsoft SQL Server ·97, 394, 677
  configuración en el BDE ·118
  Cursores · *Véase*
  dispositivo ·502
  Indices ·508
  integridad referencial ·507
  SQL Enterprise Manager ·501
Midas ·845
  Balance de carga ·848, 874
  Interfaces duales ·875
MinValue ·334
ModalResult ·274, 580, 581
Modelo de Objetos Componentes · 800
  Bibliotecas de tipos ·817
  GUID ·802
  in-process servers ·808
  Interfaces ·800, 801
  marshaling ·809
  Puntos de conexión ·816
Modified ·573, 651, 659
ModifySQL ·667
Módulos de datos ·283, 294
  remotos ·850
Módulos Web ·763
Motor de Datos ·103
  Administrador ·51, 108

N

NET DIR ·112, 622, 885
NewValue ·667, 862
Next ·591
Notification ·199
Now ·250
NSAPI ·761
Null ·415, 421, 460, 573

O

object ·209
Object Browser ·51
OBJECT MODE ·121, 532
Object Repository · *Véase* Depósito de Objetos
ODBC ·106, 112
OldValue ·667, 670, 862
OLE · *Véase* Modelo de Objetos Componentes
OLEnterprise ·107, 845, 873

OleSelfRegister ·892
OnAction ·766
OnAddReports ·714
OnCalcFields ·335, 403, 420, 585
OnCellClick ·381, 388
OnChange ·333, 355
OnClose ·218, 267, 377
OnCloseQuery ·218, 580, 638, 658
OnCreate ·294, 376
OnDataChange ·365
OnDeleteError ·586, 595, 669
OnDestroy ·294
OnDrawColumnCell ·379, 388, 654
OnDrawItem ·358
OnEditButtonClick ·378
OnEditError ·586, 595, 628, 844
OnException ·236
OnExit ·355
OnFilterRecord ·414, 418, 585
OnGetText ·331
OnHint ·214
OnHTMLTag ·770
OnIdle ·188, 214
OnLogin ·609
OnMeasureItem ·358
OnNewRecord ·344, 586, 587
OnPaint ·184
OnPassword ·622
OnPopup ·400
OnPostError ·586, 595, 602, 629, 669
OnPrint ·712
OnReconcileError ·864
OnResize ·184, 280
OnServerYield ·586
OnSetText ·331
OnStartup ·622
OnStateChange ·364
OnTitleClick ·381
OnUpdateData ·365
OnUpdateError ·586, 669, 862
OnUpdateRecord ·586, 666, 668
OnValidate ·333, 585, 588
Open ·304
OpenIndexFile ·393
Oracle ·99, 395, 679
  configuración en el BDE ·120
  cursores ·524
  Procedimientos almacenados ·522
  secuencias ·526
  SQL*Plus ·517
  tipos de datos ·520
  tipos de objetos ·528
  triggers ·525
order by ·308, 464, 539
OutputDebugString ·51
overload ·136, 145, 161, 265
override ·156

P

Packages ·70
  activación ·72
  colecciones ·74
  editor ·43
  Instalación ·887
  QuickReport ·699
PacketRecords ·855
Paradox ·88, 391
ParamByName ·544, 553
Parámetros
  por omisión ·140
Params ·544, 553, 605
PASSWORD ·610
PasteFromClipboard ·363
PATH ·606, 885
PChar ·253
PeekMessage ·188
Perro ·213
  de Codd ·80, 84, 600
PickList ·377
Pila de ejecución ·226
Plantillas de código ·44
Plantillas de componentes ·293
Post ·321, 574, 651
PostMessage ·184
PostQuitMessage ·181
Preparación de consultas ·546
Prepare ·546
primary key ·444
private ·141, 163

Privilegios ·453
Procedimientos almacenados ·88, 479, 618
ProcessMessages ·188
ProgIdToClassId ·811
Propiedades
  Code Completion ·48
  por omisión ·202
  vectoriales ·200
protected ·141, 163
ProviderFlags ·860
Proyecto
  fichero de ·58, 281
Proyectos
  grupos de ·74
public ·141, 163, 195
published ·141, 196, 203
Punteros
  a funciones ·207
  a métodos ·209
Punto de ruptura ·49
Puntos de conexión ·816

Q

QueryInterface ·803
QuickReport ·697, 828
  Componentes de impresión ·705
  expresiones ·706
  Informes compuestos ·714
  Informes con grupos ·709
  Informes master/detail ·712
  plantillas y asistentes ·700
  Previsualización ·715
QuotedStr ·418

R

raise ·196, 225, 227, 233
Rangos ·391, 406
read ·197
ReadOnly ·305, 352
RecNo ·789
Reconciliación ·864
record ·131
RecordCount ·309, 591
Recursos ·67
  ficheros de ·56, 203
Referencias de clases ·166, 274
Refresh ·410, 636
RefreshLookupList ·338
RegisterClass ·175
RegisterClassEx ·175
Registros de transacciones ·645
Reglas de empresa ·585
Reglas de Marteens ·233, 235, 595
Rejillas de selección múltiple ·383
Relación de referencia ·336
Relación master/detail ·313, 336, 398, 430, 447, 592, 639, 653, 712, 868
Release ·804
Remote Procedure Calls ·809
RenameTable ·348
Replicación ·644
ReportSmith ·698, 884
RequestLive ·539, 561
Required ·334
requires ·70
ResetAfterPrint ·707
Resolución ·859
resourcestring ·56, 385
Restricciones de integridad ·80, 442
  clave artificial ·444
  clave externa ·82, 445
  clave primaria ·81, 444
  claves alternativas ·444
revoke ·453
Rollback ·636
RowsAffected ·548
ROWSET SIZE ·121

S

SavePoint ·790
SaveToFile ·871
SCHEMA CACHE DIR ·611
select ·458
  selección única ·469
  subconsultas correlacionadas ·471
SelectedField ·376
SelectedIndex ·376
SelectedRows ·383
SelectNext ·276
Self ·138, 275

SendMessage ·184
SendParams ·867
Serializabilidad ·640
SERVER NAME ·117
Servidores de automatización ·812
Sesiones ·302, 613
Session ·613
Sessions ·613
SetFields ·577
SetLength ·256, 258, 576
SetOrdProp ·206
SetRange ·408
SetRangeEnd ·410
SetRangeStart ·410
SetStrProp ·206
ShareMem ·740
Shell_NotifyIcon ·185
ShortString ·252
Show ·180, 268
ShowException ·234
ShowModal ·180, 268, 424
ShowWindow ·180
Sobrecarga ·161
SQL
  Lenguaje de Control de Datos · 435
  Lenguaje de Definición de Datos · 435
  Lenguaje de Manipulación de Datos ·435, 475
SQL Links ·105, 438
SQL Monitor ·55, 559
SQL Net ·120
SQLPASSTHRUMODE ·605
SQLQRYMODE ·538
StartTransaction ·636, 639, 658
State ·321, 571, 588
store ·203
Stored procedures · *Véase* Procedimientos almacenados
StoredDefs ·786
StoredProcName ·553
Streams ·264
suspend ·486

T

Tabla de Métodos Virtuales ·158, 168, 213
Tablas
  anidadas ·301
TableName ·304
TableType ·304, 348
TActionList ·857
TADTField ·532
TAggregateField ·326
TApplication ·180, 188, 236, 267
TArrayField ·532
TAutoObject ·813
TBatchMove ·54
TBDEDataSet ·300
TBlobStream ·264, 368
TBookmark ·310
TBookmarkList ·383
TBookmarkStr ·310
TBooleanField ·329, 331, 362
TCalendar ·363
TChart ·722
TClientDataSet ·785, 846, 855
TClientSocket ·799
TColumn ·374
TComObjectFactory ·826
TComponent ·154, 168, 199, 294
TControl ·154, 294
TControlClass ·342, 355
TCorbaConnection ·874
TCriticalSection ·840
TCustomMaskEdit ·355
TDataAction ·596
TDatabase ·110, 294, 303, 342, 603, 636, 651
  herencia visual ·611
  InTransaction ·637
  IsSqlBased ·604
  Temporary ·603
  TransIsolation ·640
TDataLink ·353
TDataModule ·294
TDataSet ·299
TDataSetField ·326, 378, 532

TDataSetPageProducer ·783
TDataSetTableProducer ·770
TDataSource ·306, 351, 363, 385, 571
TDateTime ·249
TDateTimePicker ·363
TDBChart ·721, 726
  series ·726
TDBCheckBox ·354, 362
TDBComboBox ·357
TDBCtrlGrid ·388
TDbDataSet ·553, 653
TDBDataSet ·301
TDBEdit ·354, 355
  OnChange ·355
TDbGrid ·372
TDBGrid ·307
  ShowPopupEditor ·378
TDbGridColumns ·374
TDBImage ·354, 363, 388
TDBListBox ·357
TDBLookupComboBox ·339, 354, 360, 378, 398
TDBLookupListBox ·360, 398
TDBMemo ·354, 356, 388
TDBNavigator ·307, 384
TDBRadioGroup ·357, 362, 388
TDBRichEdit ·356, 388
TDBText ·357
TDCOMConnection ·854, 871, 874
TDecisionCube ·727
TDecisionGraph ·727
TDecisionGrid ·727, 729
TDecisionPivot ·727, 730
TDecisionQuery ·727
TDecisionSource ·727
TDrawGrid ·372
TField ·325
  IsNull ·330
  Name ·327
  OldValue ·670
  OnChange ·333
  Value ·329
TFieldClass ·342
TFieldDataLink ·353
TFieldDef
  ChildDefs ·348
TFieldDefs ·346
TFieldType ·347
TFileStream ·264
TGraphicControl ·154, 357
TInterfacedObject ·806
TLabel ·354
TLineSeries ·723
TList ·263
TMemoryStream ·264, 368, 720
TMessage ·182
TNestedTable ·316, 532
TNumericField ·325
TObject ·152, 153
TObjectField ·326
TPageControl ·725
TPageProducer ·769
TPersistent ·153, 203
TPopupMenu ·186, 400, 416
TProvider ·854
TQRCompositeReport ·714
TQRDetailLink ·713
TQrExpr ·706
TQRGroup ·709
TQuery ·399, 458, 537, 540, 546, 548, 566, 710, 726
TQueryTableProducer ·770
TQuickRep ·710
  Preview ·701
  Print ·701
  PrintBackground ·701
TQuickReport ·701
Transacciones ·579, 625, 634
  aislamiento ·639
  arquitectura multigeneracional · 643
  commit work ·636
  en bases de datos remotas ·867
  lecturas repetibles ·640
  rollback work ·636
  serializabilidad ·640
  start transaction ·636

Transaction logs ·*Véase* Registros de transacciones
Transact-SQL ·479
triggers ·512
TransIsolation ·608, 640, 650
TranslateMessage ·179
TReconcileErrorForm ·864
TReferenceField ·532
TReport ·698
Triggers ·88, 489, 586, 661, 676, 799
new /old ·491
new/old ·589, 668
Oracle ·525
Transact-SQL ·512
try/except ·229, 233, 552, 595, 602, 629
cláusulas on ·231
try/finally ·227, 233, 270, 310
TServerSocket ·799
TSession ·613
AddAlias ·620
AddStandardAlias ·620
ConfigMode ·620
DeleteAlias ·621
GetAliasDriverName ·617
GetAliasName ·617
GetAliasParams ·617
GetDatabaseNames ·617
Get-DriverNames ·617
GetDriverParams ·617
GetStoredProcNames ·618
GetTableNames ·618
ModifyAlias ·621
NetDir ·622
PrivateDir ·622
TSimpleObjectBroker ·874
TSocketConnection ·873, 874
TSplitter ·280
TStoredProc ·481, 553
TStream ·264
TStringField ·329
TStringGrid ·372
TStringList ·261
TStrings ·261, 357
TTabControl ·409
TTable ·303
Constraints ·345
FlushBuffers ·594
TThread ·616
FreeOnTerminate ·616
TUpdateSQL ·540, 667
TVarRec ·259
TWebActionItem ·765
TWebDispatcher ·765
TWebModule ·764
TWinControl ·154, 166
TypeInfo ·205, 268, 365

U

UnAssigned ·260
UndoLastChange ·790
Unidades ·58, 62
Unidades de importación ·739
UniDirectional ·541
Uniformidad referencial ·194
unique ·444
update ·475
UpdateMode ·630, 666, 670, 860
UpdateObject ·667
UpdateRecord ·367
UpdateRecordTypes ·655
UpdatesPending ·650, 659
UpdateStatus ·653, 866
UpperCase ·419
URL ·758
USER NAME ·609
uses ·59, 64

V

Valores nulos ·460
Value ·329
ValueChecked ·362
ValueUnchecked ·362
VarArrayOf ·421
Variant ·260, 329, 421, 573
VarIsNull ·422
VarToStr ·421
Vectores
dinámicos ·258
Vectores abiertos ·256

variantes ·259, 401
Ventana
clase ·175
handle ·175
procedimiento de ·180, 191
virtual ·155
Visible ·180, 267, 373
VisibleButtons ·385
Vistas ·451, 476, 675
Visual Query Builder ·555
VMT · *Véase* Tabla de Métodos Virtuales

W

WaitForSingleObject ·189
where ·459
Windows ISQL ·437, 481
with check option ·477
write ·197