La guía de Delphi

por Francisco Charte

© Danysoft 2012

DERECHOS RESERVADOS

El contenido de esta publicación tiene todos los derechos reservados, por lo que no se puede reproducir, transcribir, transmitir, almacenar en un sistema de recuperación o traducir a otro idioma de ninguna forma o por ningún medio mecánico, manual, electrónico, magnético, químico, óptico, o de otro modo. La persecución de una reproducción no autorizada tiene como consecuencia la cárcel y/o multas.

LIMITACIÓN DE LA RESPONSABILIDAD

Tanto el autor como en Danysoft hemos revisado el texto para evitar cualquier tipo de error, pero no podemos prometerle que el libro esté siempre libre de errores. Por ello le rogamos nos remita por e-mail sus comentarios sobre el libro a attcliente@danysoft.com

DESCUENTOS ESPECIALES

Recuerde que Danysoft ofrece descuentos especiales a centros de formación y en adquisiciones por volumen. Para más detalles, consulte con Danysoft.

MARCAS REGISTRADAS

Todos los productos y marcas se mencionan únicamente con fines de identificación y están registrados por sus respectivas compañías.

Autor: Francisco Charte
Publicado en castellano por Danysoft
Avda. de la Industria, 4 Edif. 1 3º
28108 Alcobendas, Madrid. España.
902 123146 | www.danysoft.com

ISBN: 978-84-939910-1-2
Depósito Legal: M-9679-2012

Por acuerdo entre el Autor y Editor, este libro se ofrece sin coste. El contenido no se puede modificar, ni obtener beneficio por su redistribución, ni eliminar la información del autor y editor del mismo.

IMPRESO EN ESPAÑA

© Danysoft | Madrid, 2012

La guía de DELPHI por Francisco Charte

Prólogo

A mis hijos: David y Alejandro
A mi alma gemela en esta travesía: María Jesús

El libro que tienes en tus manos ha sido escrito pensando en los distintos tipos de desarrolladores que podrían estar interesados en la última versión de la herramienta estrella de Embarcadero: Delphi XE2.

Partiendo de que es complejo clasificar a los desarrolladores en categorías, dado que cada uno tiene necesidades específicas y diferenciadas, el objetivo ha sido ofrecer aquello que interesará a personas que nunca antes han usado Delphi, por una parte; a los que ya conocen este producto y su necesidad primordial es el desarrollo con bases de datos o crear aplicaciones web, por otra, y los que están interesados especialmente en las novedades de Delphi XE2 que hacen posible el desarrollo multiplataforma, por otra.

En la redacción de este libro he empleado mi experiencia previa sobre Delphi, un entorno de desarrollo con el que comencé a trabajar antes de que estuviese disponible la primera versión y sobre la que he publicado casi una veintena de libros, pero no es en ningún caso una actualización de títulos anteriores sino un libro completamente nuevo basado en la versión XE2.

Ha de tenerse en cuenta, no obstante, que éste no es un libro para aquellos que no tienen unos fundamentos previos sobre programación, concretamente con algún lenguaje orientado a objetos. La primera parte, en la que se introduce Delphi XE2 para aquellos que no han trabajo previamente con esta herramienta, asume conceptos básicos tanto a nivel de usuario: interacción con la interfaz gráfica del sistema operativo y sus aplicaciones; como de programador: qué son los tipos de datos o variables, las clases, la herencia, etc.

Para que este libro llegase hasta el lector ha sido necesario, aparte del mío propio, el trabajo de un equipo de personas en Danysoft al que quisiera transmitir mi agradecimiento, tanto por la confianza depositada como por el apoyo durante el tiempo que duró su elaboración. Asimismo agradezco el siempre incondicional apoyo de mi familia frente a una tarea que exige tanta dedicación.

Tabla de Contenidos

Prólogo..3

Tabla de Contenidos..5

Introducción..15
La historia de Delphi..16
Delphi XE2..19
Sobre este libro..20
Sobre el autor...22

Apartado I:
Delphi XE2..23

Capítulo 1: Características de Delphi XE2..25
¿Qué es Delphi XE2?...26
Compiladores Delphi...27
Bibliotecas de componentes...28
Herramientas y utilidades..29
Ediciones de Delphi y sus características...31

Actualizar Delphi XE2......32
Conclusión......34
A continuación......34

Capítulo 2: Introducción al entorno......35
MicroClassic......36
Inicio del proyecto......36
Selección de la plantilla adecuada......38
Configuración del proyecto......40
Guardar el proyecto y sus elementos......41
Diseño de la interfaz de usuario......42
Búsqueda y selección de componentes......42
Distribución de los componentes......43
Edición de propiedades......46
Estructura de la interfaz......49
La lista de ordenadores......52
Diseño de la ficha y descripción......54
La lista de imágenes......57
Gestión de eventos......60
Método general......60
Agregar ordenadores nuevos......62
Sincronizar la lista con la ficha (y viceversa)......63
Acciones de las opciones de menú......65
Un visor para las imágenes......66
Herramientas de depuración......70
Puntos de parada......70
Estado del proceso......71
Ejecución paso a paso......73
Examinar el contenido de variables......74
Despliegue de la aplicación......75
Conclusión......76
A continuación......77

Capítulo 3: Introducción al lenguaje......79
Sintaxis básica......80
Módulos de código en Delphi......80
Estructura de un programa......81
Estructura de un módulo (unit)......82
Referencias a módulos......83
Comentarios......84

Tipos de datos fundamentales......85
Números enteros y en coma flotante......86
Caracteres y cadenas de caracteres......87
Otros tipos de datos básicos......88

Notación para identificadores y literales....88
Enumeraciones y subrangos....90
Matrices....91
Registros....94
Conjuntos....96
Expresiones....97
Aritméticas, relacionales y lógicas....98
Conjuntos....99
Punteros....99
Otros tipos de expresiones....99
Sentencias....100
Condicionales....101
Iterativas....102
Procedimientos/Funciones....103
Control de excepciones....106
Otras sentencias....107
Clases y sus miembros....108
Definición de una clase....108
Visibilidad de los miembros de una clase....110
Construcción de objetos....111
La clase TColeccionable....112
La clase TOrdenador....119
Miembros de clase y el objeto self....121
Métodos anónimos....125
Tipos genéricos....128
Definir un tipo genérico....128
Instanciación y uso de un tipo genérico....130
Asistencia a la edición de código....131
Plantillas de código....131
Formatear el código....133
Navegar por el código....134
Refactorización del código....134
Acceso a versiones previas....135
Conclusión....136
A continuación....136

Apartado II:
Bases de datos....137

Capítulo 4: Fundamentos del acceso a datos en Delphi XE2....139
Aplicaciones y bases de datos....140
Acceso a datos en Delphi XE2....140
Estructura de una aplicación con acceso a datos....143
RDBMS, controladores y componentes....145
Herramientas de bases de datos en el entorno....146

El Explorador de datos....147
Manipulación de la información de esquema de una base de datos....151
Manipulación interactiva de los datos....152
Módulos de datos....153
MyBase....156
El componente TClientDataSet....157
Definición interactiva de columnas....159
Crear y almacenar el dataset....162
Integrar MyBase en la aplicación....164
Distribución de una aplicación MyBase....166
Conclusión....167
A continuación....167

Capítulo 5: LiveBindings....169
Componentes visuales con conexión a datos....170
Introducción a LiveBindings....172
Tipos de LiveBindings....174
Participantes y expresiones de un LiveBinding....175
Conexión genérica entre propiedades....177
Enlace simple entre componentes FMX....177
Enlace estándar....179
Enlace compuesto....183
Enlace con componentes de acceso a datos....187
Enlazando componentes simples....188
Enlaces TBindLink....193
Enlaces TBindPosition....193
Enlaces TBindList....196
Enlace de las imágenes de un ordenador....201
Conclusión....201
A continuación....202

Capítulo 6: dbExpress....203
Componentes dbExpress....204
Conexión con la base de datos....206
Almacenar y recuperar los parámetros de conexión....209
Configuración de la conexión al ejecutar....210
Obtener información de conexiones y controladores dbExpress....211
Ejecución de comandos....216
El componente TSQLDataSet....217
Consultas y parámetros....219
Navegación y actualización....222
Almacenamiento local de datos y cambios....223
Generación de comandos de actualización....225
Resolución de conflictos de actualización....227
El componente TSimpleDataSet....230

MicroClassic colaborativo....232
Instalación del RDBMS....232
Creación de la base de datos....234
Generadores de secuencias y su uso....237
Cambios en el módulo de datos....239
Cambios en la interfaz de usuario....242
Mejoras a la aplicación....243
Controladores dbExpress....245
Conclusión....246
A continuación....246

Capítulo 7: DataSnap....247
Componentes DataSnap....248
Un servidor DataSnap simple....251
Modo de ejecución del servidor....252
Características del servidor....253
Configuración de puertos....255
Otros parámetros....256
Inicio del servidor y métodos administrativos....258
Tiempo de vida de los objetos en el servidor....259
Configuración TCP/IP....261
Componentes de acceso a datos....263
Métodos expuestos por el servidor....264
Información sobre las conexiones activas....266
Ejecución del servidor....268
Desarrollo de un cliente básico....270
Proxys de acceso a los servicios....276
Ciclo de vida de objetos en el servidor....279
Configuración por defecto....280
Creación de un objeto por llamada....281
Creación de un único objeto global....281
MicroClassic distribuido....282
MicroClassicServer....282
MicroClassicClient....285
Configuración de seguridad....288
Filtros activos en el transporte....289
Identificación de usuarios....290
Conclusión....294
A continuación....295

Apartado III:
Aplicaciones web....297

Capítulo 8: Aplicaciones IntraWeb....299
Tipos de proyectos web en Delphi....300

Actualización de IntraWeb....302
Estructura de una aplicación IntraWeb....304
Inicialización de una aplicación IntraWeb....304
El controlador del servidor....305
Formularios....307
Sesión de usuario....308
Clases de navegadores....309
Componentes IntraWeb....311
IntraWeb en la práctica....313
Información de sesión....315
La página principal....317
La lista de conexiones....320
Ejecución del proyecto....321
Asociar formularios y URLs....324
Identificación de usuarios....325
Conclusión....328
A continuación....328

Capítulo 9: Interfaces web para aplicaciones DataSnap....329
Servidores web y de aplicaciones....330
Interfaz IntraWeb y servidor DataSnap....331
Conexión con el servidor DataSnap....331
Acceso a los datos desde la interfaz....333
Diseño de la interfaz – LiveBindings....334
Diseño de la interfaz – Controles DB....339
Servidor DataSnap y REST....340
El asistente DataSnap REST Application....341
Rutas de acceso a la aplicación....342
Generación del contenido....344
Configuración del servidor web....346
Proxys JavaScript de acceso a servicios....348
Solicitudes de archivos....349
Interfaz del servidor....351
Ejecución de la aplicación....352
Acceso a los servicios REST....355
Generación de HTML a partir de datos....356
Conclusión....359
A continuación....359

Apartado IV:
Aplicaciones multiplataforma....361

Capítulo 10: Aplicaciones de 64 bits....363
32 bits versus 64 bits....364
Aplicaciones de 64 bits en Windows....366

Plataformas de un proyecto..368
Perfiles remotos..370
Instalación del Asistente de plataforma..371
Configuración de paserver..372
Definición del perfil remoto en Delphi..374
Despliegue del proyecto..377
Ejecución y depuración remota..379
Código Delphi de 64 bits..381
Tipos de datos nativos..381
Punteros..384
Conclusión..385
A continuación..385

Capítulo 11: Interfaces FMX..387
FireMonkey y el hardware gráfico..388
Direct2D versus GDI+ en Windows..389
Tipos de interfaces FMX..391
Componentes FMX – Aspectos generales..393
Distribución de controles en la ventana..393
Todos los controles son contenedores..397
Colores, bordes y fondos..400
Estilos visuales..401
Composición del estilo de un control..402
Reutilizar un estilo en otros controles..406
Estilos por defecto..408
Cómo definir un estilo nuevo..409
Estilos predefinidos..413
Cambiar de estilo en ejecución..415
Animaciones y transformaciones..418
Tipos de animaciones..419
Configuración básica de una animación..420
Control de la animación..422
Rotación y escalado..424
Animaciones y transformaciones en la práctica..425
Efectos visuales y filtros..430
La jerarquía TEffect..431
Configuración de un efecto visual..433
Interfaces 3D..435
Estructura de una interfaz 3D..437
Objetos tridimensionales..439
Materiales..445
Luces..448
Cámaras..450
Objetos 2D en un mundo 3D..452
Conclusión..454

A continuación....454

Capítulo 12: Delphi XE2 y MacOSX....455
El compilador para MacOS X....456
Configuración de la máquina remota....457
Instalación de paserver....457
Puesta en marcha de paserver....459
Configuración en la máquina local....461
Despliegue de módulos redistribuibles....463
Conclusión....465
A continuación....465

Capítulo 13: Aplicaciones para otras plataformas....467
Desarrollo para iOS en Delphi....468
Compilación de código Delphi para iOS....468
Conversión de proyectos Delphi a Xcode....469
Preparación de la máquina MacOS X....471
Compartir carpetas y escritorio....474
Creación de un proyecto para iOS en Delphi....475
Ejecución del proyecto en Xcode....478
Desarrollo para otras plataformas....481
DataSnap Mobile Connectors....481
Obtención de un conector concreto....483
Uso del conector....485
Conclusión....486
A continuación....486

Apartado V:
Informes y documentación....487

Capítulo 14: Informes con FastReport....489
Componentes FastReport....490
Diseño de un informe....492
Conexión con los datos a usar....492
Asistentes y plantillas para informes....494
Personalización del informe....497
Previsualización e impresión 500
Personalizar la vista previa....501
Informes FastReport en aplicaciones FMX....502
Conclusión....504
A continuación....504

Capítulo 15: Documentation Insight....505
Documentación XMLDoc....506
Editar la documentación con Documentation Insight....507
Edición manual de la documentación....510

Generar documentación electrónica....................513
Conclusión....................517

Indice....................519

Introducción

Delphi XE2 es la versión 16 de Delphi, una herramienta de desarrollo presentada por Borland en 1995. Si bien no es algo que sea imprescindible conocer para poder comenzar a usar una determinada herramienta, narrar la historia de ésta cuando se escribe un nuevo libro es algo que se ha convertido prácticamente en una tradición. No faltaremos aquí a la misma y, aunque brevemente, haremos un recorrido cronológico que nos permita saber cuál es el origen de Delphi XE2.

Una vez situado en su contexto histórico, nos centraremos en las características distintivas de Delphi XE2. La más reciente versión de este producto, disponible desde otoño de 2011, puede ser considerada la más importante revisión de los últimos años tanto por la cantidad de novedades que incluye como por el calado de éstas.

Esta introducción concluirá describiendo someramente los contenidos del libro, a fin de guiar al lector hacia las partes que más pudieran interesarle, y ofreciendo una breve reseña general sobre el autor y su experiencia con Delphi en particular.

La historia de Delphi

Hace ahora algo más de 30 años, en 1981, un joven estudiante danés llamado Anders Hejlsberg pensó que sería buena idea crear para el Nascom[1] una versión del lenguaje Pascal[2] que sustituyese al BASIC de Microsoft que dicho ordenador incorporaba en memoria ROM. En apenas 12 kilobytes de memoria incluyó un sencillo editor de líneas, el compilador y las bibliotecas necesarias, de forma que todo podía ser introducido en una EPROM que se insertaba en el zócalo donde originalmente estaba el intérprete de BASIC.

Hejlsberg llamó a su producto BLS Pascal, siendo BLS el acrónimo de *Blue Label Software* que era el nombre de su empresa. A esa implementación le siguieron COMPAS Pascal, versión de 8 bits para CP/M presentada en 1982, y PolyPascal, una versión de 16 bits para CP/M-86 y MS-DOS presentada en 1984.

En julio de 1983, encontrándose de vacaciones en España, Hejlsberg recibió una llamada de su compañero Preben Madsen: una nueva empresa, llamada Borland y que desarrollaba sus productos con DR Pascal, estaba interesada en licenciar COMPAS Pascal 3.0 siempre que desarrollasen una versión de 16 bits. Según Hejlsberg, el trabajo de esta nueva versión comenzó el 22 de julio de 1983 y finalizó en apenas cuatro meses. El 20 de noviembre de 1983 se presentaba Borland Turbo Pascal 1.0 que, básicamente, era COMPAS Pascal 3.02 con el añadido de un editor a pantalla completa (basado en WordStar[3]) y un menú que facilitaba las operaciones de edición, compilación, carga y almacenamiento de programas.

Aunque en principio Hejlsberg continuó con su empresa, llamada PolyData, vendiendo una versión propia de 16 bits llamada PolyPascal, finalmente se incorporó a Borland como diseñador jefe de Turbo Pascal, un producto que haría evolucionar durante aproximadamente una década.

---

1 Nascom era una empresa que a finales de los 70 vendía kits de ordenador, basados en una placa en la que se incluía el procesador, memoria ROM y RAM y una interfaz de vídeo estándar. Más información sobre estos ordenadores en `http://www.nascomhomepage.com`.

2 Por entonces Pascal era un lenguaje prácticamente nuevo, derivado de Algol que era el lenguaje empleado habitualmente para enseñar programación.

3 WordStar, de la empresa MicroPro, está considerado el primer editor de textos accesible para el público en general al estar disponible para microordenadores con CP/M y DOS.

Entre 1984 y 1988 van lanzándose sucesivas versiones de Turbo Pascal: 2.0, 3.0, 4.0 y 5.0, cada una de ellas con diversas mejoras. En 1989, con la versión 5.5, se comienzan a introducir características de orientación a objetos y a hablar de *Object Pascal*. En los años siguientes se presentarían las versiones 6.0 y 7, esta última redenominada como *Borland Pascal with Objects*, así como las versiones 1.0 y 1.5 de Turbo Pascal para Windows 3.1, esta última en 1991. Las versiones para Windows contaban con una biblioteca de clases propia, denominada OWL (*Object Windows Library*), que Borland proponía como alternativa a la propia de Microsoft: MFC (*Microsoft Foundation Classes*).

Tomando como base el lenguaje Object Pascal, que había evolucionado en las últimas versiones del producto, y el compilador para Windows con el que ya se contaba, se desarrolla un nuevo entorno de trabajo con un enfoque RAD (*Rapid Application Development*) inspirado en Visual Basic[4] y una nueva biblioteca de componentes visuales llamada VCL (*Visual Component Library*). El nuevo producto, lanzado el 14 de febrero de 1995, recibe el nombre Delphi[5] 1.0, si bien en principio se iba a denominar AppBuilder e internamente era conocido como VBK (*Visual Basic Killer*) dejando claro cuál era su objetivo.

Apenas un años después, en febrero de 1996, se presenta Delphi 2.0, una versión de 32 bits adaptada al nuevo Windows 95. En 1997, 1998 y 1999 aparecen las versiones 3.0, 4.0 y 5.0 con mejoras en el lenguaje, nuevos componentes de acceso a datos, soporte para COM[6] y CORBA[7] y adaptación a Windows 98.

En 2001, con la versión 6 de Delphi, se introduce la nueva biblioteca de componentes multiplataforma CLX y la primera versión de Kylix, una adaptación de Delphi para el sistema operativo Linux que, a pesar de contar con dos versiones adicionales en 2001 y 2002, pasaría a la historia con más pena que gloria. Paralelamente a Kylix 3.0, en 2002 también aparece Delphi

---

4 Visual Basic 1.0 se lanzó a principios de 1991, poco después que Windows 3.0, y en 1993 la versión 3.0 era uno de los entornos de programación más populares para Windows.

5 Uno de los aspectos sobre el que más empeño se puso en el desarrollo de Delphi fue el acceso a bases de datos. Hace 20 años el término *base de datos* era casi sinónimo de Oracle (Oráculo). Haciendo un juego de palabras: *Para hablar al Oráculo hay que ir a Delphos*, se decidió usar *Delphi* como nombre del nuevo producto.

6 *Component Object Model*: El modelo de componentes introducido por Microsoft en Windows 95.

7.0 que, al igual que la versión previa, ponía especial énfasis en el desarrollo web y la introducción de herramientas de modelado, un campo en el que Borland quería adentrarse y que le llevó a cambiar su nombre por Inprise

Tras un largo tiempo en desarrollo en 2003 se lanza Delphi 8 for .NET, una versión dirigida a la plataforma .NET introducida por Microsoft en 2000. Ésta sería la última versión de Delphi con denominación de versión consecutiva. En 2004 y 2005 aparecen Delphi 2005 y Delphi 2006, desarrollados sobre la base de Delphi 8 incidiendo en el apoyo a la plataforma .NET lo que les lleva incluso a incorporar soporte para el lenguaje C#[8].

Tras deshacer su cambio de nombre a Inprise, y volviendo a denominarse Borland, la empresa toma la decisión de deshacerse de la división dedicada a herramientas de desarrollo. Para ello comienza separándola bajo un nombre distinto: CodeGear, con autonomía propia. Esto se traduce en una reorientación del producto más importante de esa división, que no es otro que Delphi. El enfoque vuelve de nuevo a Win32 como plataforma objetivo más importante, facilitando la creación de aplicaciones para Windows Vista en CodeGear Delphi 2007 (versión 11.0 de Delphi). Un año después, en agosto de 2008, se lanza CodeGear Delphi 2009 con importantes novedades en el lenguaje y nuevos componentes.

En 2008 Borland finalmente vende CodeGear a la empresa Embarcadero, comprometida desde el principio a seguir evolucionando y mejorando una herramienta tan emblemática como Delphi para un gran número de desarrolladores a lo largo de todo el mundo. Ese compromiso queda reflejado en las versiones Delphi 2010[9], XE y XE2 presentadas en 2009, 2010 y 2011, respectivamente. En ellas se continua con la reorientación iniciada en CodeGear hacia Win32 y la incorporación de novedades en el lenguaje, además de nuevas características como el desarrollo para la nube o la inclusión de nuevos modelos de componentes multiplataforma.

---

7 *Common Object Request Broker Architecture*: Un modelo de componentes distribuidos estándar compatible entre sistemas y lenguajes de programación.

8 Paradójicamente C# fue desarrollado por Anders Hejlsberg en Microsoft, empresa a la que se incorporó en 1996, como parte de la plataforma .NET.

9 Aunque por numeración consecutiva a Delphi 2010 le correspondería la versión 13, en Embarcadero decidieron saltársela y pasar a la 14, de ahí que Delphi XE2 sea la versión 16.

Delphi XE2

La última versión de Delphi, disponible desde septiembre de 2011, es heredera de un avanzado lenguaje de programación: una versión de Pascal orientada a objetos a la que se han agregado multitud de extensiones en los últimos años; unas extensas bibliotecas de servicios y componentes: la RTL y la VCL, y un compilador y un entorno que han ido evolucionando paulatinamente para adaptarse a las innovaciones de cada nueva versión de Windows.

En todos y cada uno de esos apartados esta versión XE2 aporta importantes mejoras y novedades, entre las que destacaría las siguientes:

- Tanto el compilador como el lenguaje se han adaptado para poder generar aplicaciones de 64 bits, un aspecto importante para programas que han de tratar con grandes volúmenes de datos aprovechando la memoria con que cuentan los actuales ordenadores. En realidad existen compiladores independientes para 32 y 64 bits.
- La VCL no solamente se ha preparado para entornos de 64 bits e incluye nuevos componentes, sino que ofrece características que afectan globalmente al diseño de interfaces de usuario como son los estilos. Es posible tanto elegir entre estilos de interfaz preconfigurados como definir otros propios, consiguiendo que nuestras aplicaciones tenga una apariencia personalizada.
- Se incluyen nuevos controladores de acceso a bases de datos, entre los cuales seguramente el más importante sea el que permite usar cualquier fuente ODBC[10]. Por regla general las empresas de productos RDBMS, como pueden ser Oracle, IBM o Microsoft, siempre ofrecen controladores ODBC mucho antes que otro tipo de controladores.
- La reestructuración de las clases relacionadas con proyectos para la nube simplifican el diseño de soluciones con una base común pero dirigidas a distintos proveedores, como pueden ser Amazon o Microsoft Azure.

---

10 *Open Database Connectivity*: Es un estándar para el acceso a bases de datos independiente del fabricante del RBDMS, el sistema operativo o el lenguaje de programación empleado.

- Sin duda la novedad más importante de esta versión XE2 es FireMonkey, nombre de una nueva biblioteca de componentes visuales que abre las puertas al desarrollo de aplicaciones multiplataforma en Windows (32 y 64 bits), Mac OSX e iOS aprovechando toda la potencia que ofrecen las actuales GPU[11]. Además de FireMonkey, para hacer posible la generación de aplicaciones dirigidas a distintas plataformas en el entorno se ha incluido un compilador Windows de 32 bits, otro de 64 bits y un compilador para Mac OSX, así como una RTL compatible con esas tres plataformas.
- Tanto en la VCL como en FireMonkey las conexiones de datos entre componentes son más sencillas y potentes que nunca antes gracias a LiveBindings, otra de las novedades importantes de Delphi XE2.
- Además de incluir versiones de 64 bits de los paquetes de componentes de terceros, como IntraWeb, Indy o TeeChart, también se han incorporado novedades de otros fabricantes como Documentation Insight, que facilita la documentación del código fuente, o FastReport para el diseño y generación de informes.

Aparte de los aquí enumerados, Delphi XE2 aporta cambios, ya sean mejoras o novedades, en prácticamente todos los elementos del producto: el compilador, el entorno, los componentes, el lenguaje, etc.

Sobre este libro

Como se apuntaba en el prólogo, el objetivo de este libro es facilitar el trabajo con Delphi XE2 a diferentes tipos de usuarios y, por ello, no está centrado exclusivamente en el estudio de las novedades específicas de esta versión, enumeradas en el punto anterior. Esto no significa, sin embargo, que dichas novedades no se traten con la profundidad que merecen, más bien al contrario.

---

11 *Graphics Processing Unit*: Denominación común del hardware gráfico actual, en el que existen núcleos de procesamiento especializados con un alto grado de paralelismo.

En el índice de contenidos puede apreciarse que los capítulos se han agrupado en cinco partes diferenciadas. La primera de ellas está dirigida a aquellos que, teniendo experiencia con otros lenguajes y herramientas, toman contacto por primera vez con Delphi. En los tres capítulos que la componen se describen las características generales de Delphi y se explican de manera concisa los aspectos más importantes tanto del entorno de trabajo como del lenguaje. No se trata de una referencia detallada de cada opción disponible o cada instrucción del lenguaje, sino de ofrecer una vía rápida que permita comenzar a trabajar con Delphi XE2 en el menor tiempo posible. A pesar de que ya conozcas versiones previas de Delphi en estos capítulos seguramente también encuentres información de interés, como la nueva denominación diferenciada para módulos pertenecientes a RTL, VCL y FireMonkey o novedades incluidas en el entorno.

La segunda parte está centrada en el acceso a bases de datos desde aplicaciones Delphi XE2, con capítulos dedicados a dbExpress, DataSnap y LiveBindings. En ellos se tratan tanto los mecanismos heredados como los introducidos como novedad en Delphi XE2, por lo que están dirigidos tanto a nuevos usuarios de esta herramienta como a los ya experimentados. Lo mismo se aplica a la tercera parte, con capítulos dedicados al desarrollo para la web.

Muchas de las novedades más importantes de Delphi XE2 son tratadas en la cuarta parte del libro, dedicada al desarrollo de aplicaciones para múltiples plataformas. Aquí encontrarás información sobre los detalles específicos que necesitas conocer al programar para Windows de 64 bits, Mac OS X o iOS[12] usando la nueva biblioteca de componentes FireMonkey.

La quinta y última parte del libro, la más breve también, describe algunas funciones nuevas de Delphi XE2 no abordadas en capítulos previos, como son el diseño de informes con FastReport o la documentación de código mediante Documentation Insight.

Como puede apreciarse, la estructura de los contenidos se ha efectuado de tal forma que el lector pueda rápidamente determinar qué capítulos son los que debe leer en cada momento según esté interesado en el trabajo con bases de datos, desarrollo web o desarrollo multiplataforma.

---

12 Sistema operativo que utilizan casi todos los productos de Apple a excepción de los iMac: iPod, iPhone e iPad.

Sobre el autor

Habiendo tomado contacto por primera vez con un ordenador a principios de la década de los ochenta, rápidamente me dí cuenta de que quería dedicar mi futuro a programar ese tipo de máquinas. Para ello aprendí multitud de lenguajes a lo largo de los años, algunos bien conocidos como C, C++, Pascal, FORTRAN, BASIC, COBOL o Java y otros de ámbito más reducido como ensamblador de varios procesadores, BAL, Prolog o Lisp. Soy ingeniero informático por la Universidad de Jaén y máster en Soft Computing e Inteligencia Computacional por la Universidad de Granada, pero mi vertiente de estudiante es un continuo en el tiempo en un intento de mantenerme siempre al día en un campo tan dinámico como éste.

Además de trabajar como desarrollador, desde 1985 he publicado más de un centenar de libros sobre sistemas operativos y programación con distintas editoriales, he escrito varios cientos de artículos en más de una decena de revistas divulgativas y enseñado también a unos cuantos cientos de alumnos tanto en centros privados como públicos.

Comencé usando Turbo Pascal 1.0 para CP/M en un MSX2 en 1986. Era la misma herramienta que Borland había lanzado para MS-DOS, en su versión para el microprocesador Z80 de 8 bits. Desde entonces Pascal se convirtió en uno de mis lenguajes preferidos y, tras cambiar mi microordenador de 8 bits por un PC, usé prácticamente todas las versiones posteriores de Turbo Pascal y Borland Pascal.

A finales del verano de 1984 los representantes de Borland en España me mostraron de manera confidencial un producto que iban a lanzar en pocos meses, proponiéndome escribir un libro sobre el mismo. Ya había trabajado anteriormente con ellos en un libro sobre Borland C++. Me facilitaron en un paquete de disquetes aquel producto, que llamaban VBK en tono de broma y que se convertiría en Delphi. Desde entonces publiqué uno o más títulos monográficos para cada versión de Delphi, en total 17 libros, y algunos más en los que Delphi también era una parte importante.

La redacción de este nuevo libro sobre Delphi, como no podría ser de otra manera, se apoya sobre mi experiencia previa, en una triple vertiente: como usuario de Delphi desde que existe, como escritor y, sobre todo, como docente. Con ello he intentado que este libro sea pedagógico, fácil de leer y útil para el lector que es el propósito final. Espero haberlo conseguido.

Apartado I: Delphi XE2

La primera parte de este libro abarca tres capítulos y está pensada para, por una parte, ofrecer a aquellos que no conocen Delphi una visión general sobre esta herramienta de desarrollo, su entorno de trabajo y su lenguaje de programación. Por otra, también servirá a los usuarios de versiones previas del producto como rápido repaso de lo que ya conocen y, sobre todo, como toma de contacto con las importantes novedades que incorpora la versión XE2 en cada uno de esos apartados.

- **Capítulo 1: Características de Delphi XE2**
- **Capítulo 2: Introducción al entorno**
- **Capítulo 3: Introducción al lenguaje**

Capítulo 1: Características de Delphi XE2

Delphi XE2 es un producto diseñado por una empresa para que sirva como herramienta de trabajo a un colectivo concreto de profesionales: los desarrolladores de software. Éstos necesitan conocer su herramienta de trabajo lo mejor posible, con el objetivo de utilizarla adecuadamente y, además, sacarle el mayor provecho. Se trata, sin embargo, de una herramienta que, a diferencia de las usadas por otros colectivos, tiene un alto grado de complejidad.

El objetivo de este capítulo es ofrecerte una visión general de cuáles son las posibilidades de la herramienta que has elegido, a fin de que sepas qué podrás hacer con ella a medida que vayas conociéndola.

¿Qué es Delphi XE2?

Delphi XE2 es, como indicaba antes, una herramienta compleja, un producto de software que se compone, básicamente, de los siguientes elementos:

- **Compiladores**: Un conjunto de compiladores que, partiendo del mismo código fuente en lenguaje Delphi, generan código para distintas plataformas: Win32, Win64 y MacOS X.
- **Componentes**: Varias bibliotecas de componentes software prefabricados que simplifican todo tipo de operaciones: confección de interfaces de usuario, conexión a bases de datos, acceso a los servicios del sistema operativo, etc. Las tres bibliotecas más importantes son la RTL (*Run-Time Library*), la VCL (*Visual Component Library*) y la FMX (*FireMonkey*).
- **Herramientas auxiliares**: Aunque disponiendo de un compilador y una biblioteca de componentes es posible crear aplicaciones, normalmente también se precisan utilidades como son un depurador, diseñadores de interfaces de usuario y de informes, un gestor de proyectos, controladores de BDD, etc. Todas ellas, y algunas más, forman parte de Delphi XE2.
- **Entorno de trabajo**: Todo lo anterior se integra en un entorno o IDE (*Integrated Development Environment*) que facilita el control sobre el ciclo de vida de desarrollo, aunando la administración de los proyectos, edición de código y diseño de interfaces, configuración, compilación y depuración.

Gracias a la existencia del IDE, al que se dedicará el capítulo siguiente, por regla general nunca tendrás que vértelas directamente con los compiladores y resto de utilidades desde la línea de comandos, aunque es una posibilidad perfectamente factible.

> ADVERTENCIA
>
> La anterior es una enumeración genérica de los elementos que forman parte de Delphi XE2, pero ha de tenerse en cuenta que su disponibilidad en tu instalación dependerá de la edición del producto con la que estés trabajando.

Compiladores Delphi

Las anteriores versiones de Delphi incluían un único compilador[13], encargado de convertir el código fuente escrito en lenguaje Delphi a código ejecutable para Windows, concretamente para la plataforma Win32 al ser un compilador de 32 bits. A éste, que sigue estando presente, en la versión XE2 se agregan dos compiladores más: uno para Win64 y otro para MacOS X.

Las ediciones de 64 bits de Windows cada vez tienen una mayor cuota en los equipos que instalan el sistema operativo de Microsoft, especialmente desde la aparición de Windows Vista y Windows 7. La razón fundamental es que los equipos cada vez cuentan con mayor cantidad de memoria, es habitual disponer de 6GB, 8GB o incluso más, y la única forma de aprovechar ese importante recurso es contar con aplicaciones que tengan a su disposición un espacio de direccionamiento superior a 32 bits, ya que esté se encuentra limitado a un máximo de 4GB. Por ello era importante disponer de un compilador Delphi para Win64, con capacidad para usar toda la memoria que precisen las actuales aplicaciones.

Además de los dos compiladores para Windows, en Delphi XE2 también encontramos un nuevo compilador para MacOS X que abre las puertas al desarrollo de aplicaciones Delphi para dicha plataforma. Se trata de un compilador de 32 bits, esperándose la disponibilidad de uno de 64 bits en un futura revisión del producto.

Los compiladores de Delphi XE2 son compiladores cruzados (*cross-compilers*). Esto significa que se usan sobre una plataforma, que puede ser Win32 o Win64 (según donde instalemos el producto), y generan código para plataformas distintas. No es necesario, por tanto, tener versiones del producto para cada una de las configuraciones posibles.

Si bien el uso del compilador adecuado lo decidirá el IDE según los parámetros establecidos para cada proyecto, si tuviéramos que invocarlos desde la línea de comandos lo haríamos como `DCC32.EXE`, `DCC64.EXE` y `DCCOSX.EXE`, según quisiésemos generar código para Win32, Win64 o MacOS X, respectivamente.

---

13 Existieron compiladores de Delphi para Linux y la plataforma .NET, el primero en Kylix y el segundo en Delphi 8 .NET y alguna versión posterior, pero en ambos casos son opciones obsoletas desde hace tiempo y no incluidas en las últimas versiones del producto. En la actualidad si se quiere desarrollar para .NET puede recurrirse a Prism, otro producto de Embarcadero similar a Delphi.

Bibliotecas de componentes

Desarrollar aplicaciones complejas apoyándose exclusivamente en la funcionalidad de un lenguaje, aunque sea muy avanzado como es el caso de Delphi, y los servicios ofrecidos por el sistema es un trabajo verdaderamente arduo. Podría equipararse a construir un automóvil sirviéndose únicamente de herramientas y materiales básicos, sin recurrir a ningún componente prefabricado.

Las bibliotecas de componentes permiten automatizar las tareas comunes, para las que ya existen soluciones bien conocidas, sin necesidad de reimplementarlas en cada proyecto y, lo que es más importante, dejando que la mayor parte de nuestro tiempo se invierta en la funcionalidad específica de la aplicación que estemos desarrollando que, al final, es lo que realmente interesa. En Delphi XE2 disponemos de tres bibliotecas de componentes:

- **RTL** (*Run-Time Library*): Es la biblioteca en la que se alojan los elementos más básicos de Delphi, como son los tipos de datos fundamentales y otros como las cadena de caracteres o las fechas, así como métodos para operar sobre caracteres en distintas codificaciones, números en punto flotante o gestionar la memoria dinámica. Sus orígenes se remontan a los tiempos de Borland Pascal y ha ido actualizándose continuamente. La RTL, con la excepción de algunos módulos específicos para Windows y MacOS X, es compatible con las tres plataformas citadas en el punto anterior: Win32, Win64 y MacOS X. Además es la base sobre la que se apoyan las dos bibliotecas siguientes.
- **VCL** *(Visual Component Library):* Como su propio nombre indica se trata de una biblioteca de componentes que, con pocas excepciones, tienen una parte visual. Con ellos se diseñan interfaces de usuario, pero también se definen conexiones a bases de datos, se generan gráficos o transfieren datos a través de redes. La VCL se introdujo con la primera versión de Delphi y ha ido renovándose principalmente para ajustarse a las novedades de las sucesivas versiones de Microsoft Windows, sistema operativo al que está estrechamente ligada. En Delphi XE2 la VCL está disponible para plataformas Win32 y Win64, pero no para MacOS X.
- **FMX** *(FireMonkey):* Al igual que la VCL, ésta es una biblioteca de componentes visuales y, por tanto, su objetivo básico es facilitar el

diseño de interfaces de usuario. La principal diferencia respecto a la anterior es que está disponible para Win32, Win64, MacOS X y también iOS, por lo que conjuntamente con la RTL hace posible el desarrollo de aplicaciones multiplataforma con Delphi. FMX es novedad en la versión XE2, no existía en versiones previas del producto, y todo parece indicar que será la biblioteca preferente en el futuro[14].

La nueva FMX no ha sido desarrollada a partir de la VCL, sino que es una biblioteca totalmente nueva y diferente. No existe, por tanto, una conversión directa de proyectos VCL existentes a FMX con el objetivo de aprovechar el hecho de que FMX es multiplataforma. La adaptación puede realizarse manualmente, no obstante requerirá un esfuerzo considerable para proyectos de cierta envergadura.

A la hora de iniciar nuevos proyectos tendremos que decidir si en éstos queremos usar la VCL o bien FMX. En ambos casos se usará el mismo lenguaje de programación y la misma RTL, pero la mayor parte de los componentes disponibles serán distintos según se opte por una u otra biblioteca[15].

Herramientas y utilidades

Como apuntaba anteriormente, resulta factible crear aplicaciones contando únicamente con un compilador y las bibliotecas de componentes, recurriendo a la línea de comando y un editor de textos básico a fin de controlar la generación del proyecto y trabajar en el código fuente. Sin embargo salvo en casos puntuales, y muy simples, no resulta habitual hacerlo ya que tenemos a nuestra disposición múltiples herramientas. Algunas de ellas son:

- **Gestor de proyectos**: A la hora de generar una aplicación es necesario conocer detalladamente tanto la lista de archivos que componen el proyecto como los parámetros de configuración y

14 Embarcadero ha afirmado que seguirán evolucionando también la VCL, de cara a posteriores versiones de Delphi, pero las ventajas de FMX son suficientemente importantes como para decantarse por esta nueva biblioteca, al menos en los proyectos nuevos.

15 Si bien Delphi XE2 en principio no permite combinar componentes VCL y FMX en un mismo proyecto, esto no significa que no sea posible hacerlo de forma *manual*.

procesamiento asociados a cada uno de ellos. Cada archivo ha de ser entregado a la herramienta que corresponda y con los parámetros adecuados, uniendo la salida producida por todos ellos para obtener el ejecutable final. Gracias al gestor de proyectos de Delphi ese trabajo se automatiza, almacenando toda la información en un archivo de proyecto cuyo contenido se edita mediante una cómoda interfaz de usuario.

- **Editores de código fuente**: Para escribir código fuente, indistintamente del lenguaje de que se trate, es posible recurrir a cualquier editor de textos. Incluso el sencillo Bloc de notas de Windows serviría para escribir código Delphi. Es mucho más cómodo hacerlo, sin embargo, con un editor especializado con resalte sintáctico (se distingue con colores los diferentes elementos del lenguaje) y asistencia a la escritura de código: ayuda contextual sobre parámetros, listas de métodos y propiedades de objetos, opciones de refactorización, etc. Delphi XE2 incorpora editores específicos para varios lenguajes, entre ellos lógicamente Delphi.
- **Diseñadores**: Ciertos elementos de una aplicación, el caso más típico es su interfaz de usuario, requieren un trabajo muy minucioso que implica el uso de decenas (o incluso cientos) de elementos y el ajuste preciso de cada uno de ellos, estableciendo propiedades que afectan tanto a su aspecto visual como a su funcionalidad. Dicho trabajo se simplifica a través de los diseñadores, herramientas que facilitan la disposición de esos elementos y la configuración de sus características. Delphi XE2 ofrece diseñadores de interfaces de usuario y también para objetos como pueden ser las tablas de una base de datos o los informes.
- **Depuradores**: Las herramientas previas permiten trabajar sobre el proyecto y generarlo, obteniendo la aplicación final. Por regla general ésta presentará fallos o comportamientos no esperados, debiendo procederse a su depuración. Delphi XE2 cuenta tanto con un depurador local como con depuradores remotos, de forma que es posible ejecutar paso a paso y controlar desde una máquina un proyecto que está ejecutándose en otra.

Además de las anteriores, también forman parte de Delphi XE2 utilidades para configurar la conexión con bases de datos, herramientas de configuración de pruebas (*unit testing*), plantillas para múltiples tipos de proyectos, etc., así como una extensa documentación de referencia.

Ediciones de Delphi y sus características

La disponibilidad de ciertos componentes y herramientas de Delphi, a fin de poder usarlos en nuestros proyectos, dependerá de la edición del producto con la que estemos trabajando. Existen cinco ediciones distintas ajustadas a las necesidades de diferentes tipos de profesionales.

En la web de Danysoft[16] se ofrece una relación detallada de las características de cada una de las ediciones existentes. El nombre de las ediciones y los aspectos más importantes son los indicados a continuación:

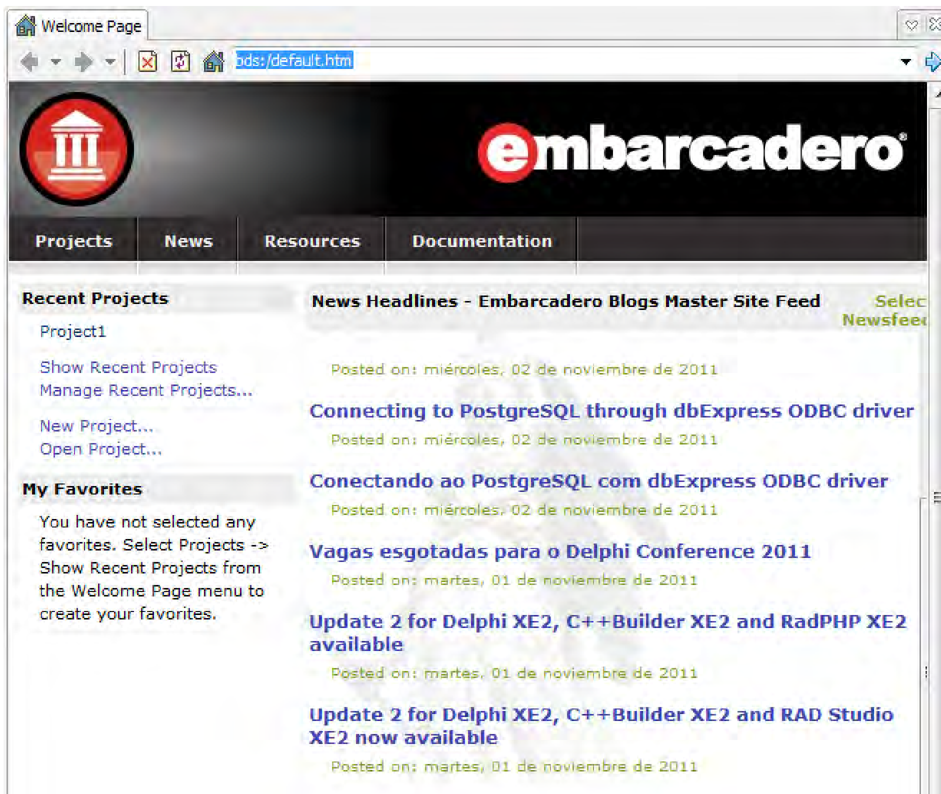
- **Starter**: La edición más básica únicamente permite desarrollar aplicaciones para Windows de 32 bits, tanto con la VCL como con FMX, con pocas posibilidades de acceso a bases de datos y menos opciones de edición de código. La principal ventaja es que su licencia de uso permite trabajar con esta edición gratuitamente a profesionales y pequeños grupos siempre que la facturación de las aplicaciones desarrolladas con ella no alcance un cierto límite.
- **Professional**: Contempla el desarrollo para las plataformas Win32, Win64, MacOS X e iOS, así como el acceso a bases de datos locales con RDBMS como InterBase y MySQL. También es posible crear aplicaciones web y para la nube.
- **Enterprise**: Agrega a las capacidades de la edición anterior controladores para acceder a los RDBMS más importantes: Oracle, SQL Server, Informix, DB2, Firebird, etc; la posibilidad de desarrollar aplicaciones distribuidas con DataSnap y herramientas adicionales como FinalBuilder y opciones de modelado UML.
- **Ultimate** y **Architect**: Las ediciones superiores de Delphi XE2 van dirigidas específicamente a profesionales que han de diseñar y optimizar soluciones de bases de datos, con herramientas de control de código y perfilado SQL, modelado y diseño visual de bases de datos.

---

16 http://www.danysoft.com/free/rad-studio-features-matrix.pdf.

Además de como producto independiente Delphi XE2 puede también adquirirse como parte de RAD Studio XE2, un paquete en el que se incluyen además C++ Builder XE2, para los que prefieren el lenguaje C++; RadPHP XE2 que facilita el desarrollo de aplicaciones web y para dispositivos móviles, y Prism XE2 que es la versión de Delphi[17] dirigida al desarrollo sobre la plataforma Microsoft .NET.

Actualizar Delphi XE2

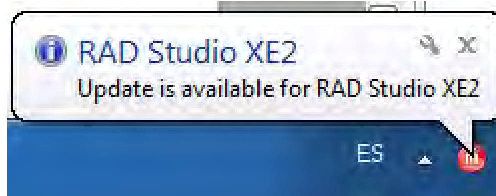


Una vez adquirida e instalada la edición de Delphi que más se adecue a nuestras necesidades no hemos de olvidar una importante tarea: mantenernos al día de las actualizaciones disponibles e instalarlas en cuanto nos sea posible. En la página de inicio de Delphi (imagen superior) se ofrece

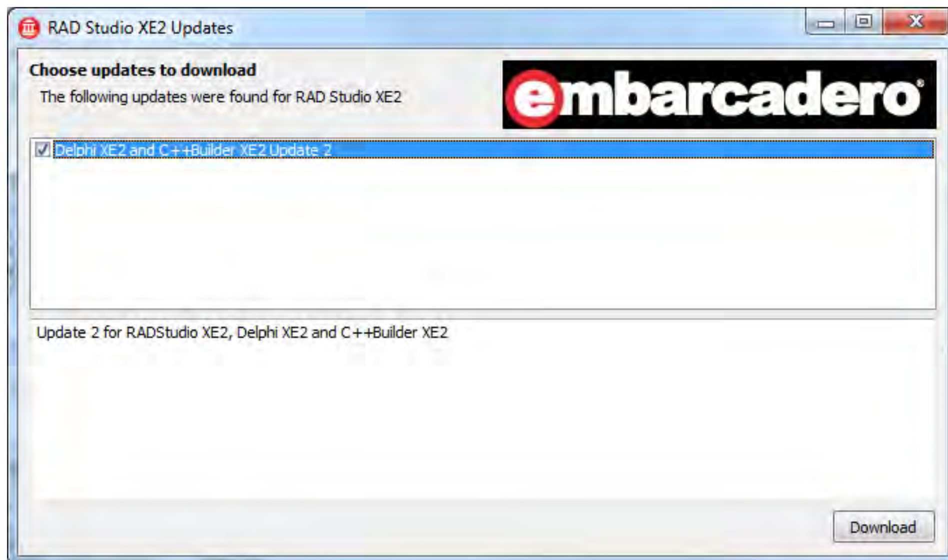
17 En realidad Prism ofrece un lenguaje de programación propio, conocido como Oxygene, muy similar a Delphi pero con ciertas características específicas.

un flujo cronológico de noticias relacionadas con el producto, entre las que aparecerán los anuncios de dichas actualizaciones. Embarcadero tiene el compromiso de corregir con celeridad los fallos que se detecten en el producto, algo que queda reflejado en la aparición de tres actualizaciones entre finales de 2011 y enero de 2012.

Si durante la instalación del producto se marcó la opción Check for Updates Automatically no tendremos que preocuparnos de nada, ya que el propio producto nos avisará con un mensaje en la Barra de tareas de Windows cuando haya actualizaciones disponibles.



En cualquier caso, siempre podemos elegir la opción Check for Updates que se instaló en el grupo de programas Embarcadero RAD Studio XE2 para abrir la utilidad que puede verse en la imagen inferior. En ella aparecerán las actualizaciones existentes que no tengamos instaladas. No hay más que marcarlas y proceder con la descarga y posterior instalación.



Una vez finalizado el proceso de instalación es posible que se abra en el navegador por defecto un documento con indicaciones adicionales. En los Update 2 y 3, por ejemplo, hay varios elementos que no son instalados automáticamente y en dicho documento se indica cuáles son y cómo hay que proceder para completar el proceso de actualización.

Mantener actualizado Delphi XE2 evitará que nos encontremos con fallos ya solucionados y, por tanto, nos ahorrará tiempo y quebraderos de cabeza.

Conclusión

Delphi XE2 es una potente herramienta de desarrollo, con un entorno que se ejecuta sobre Windows y que permite construir aplicaciones para plataformas Win32, Win64, MacOS X e iOS. Las bibliotecas de componentes que incorpora, en especial FMX, facilita el diseño de interfaces de usuario atractivas, capaces de aprovechar toda la potencia de los actuales adaptadores gráficos. También es posible desarrollar proyectos de aplicaciones web y para la nube. En todos los casos se cuenta con múltiples mecanismos de acceso a bases de datos, especialmente en las ediciones más altas del producto, contacta con Danysoft para más información.

A continuación

Los compiladores, bibliotecas de componentes y resto de herramientas y utilidades de Delphi XE2 están integrados en un entorno que es necesario conocer en profundidad para poder aprovechar todas las posibilidades que ofrece. No hay una receta rápida o mágica para dominar un entorno tan complejo en pocas horas o días. Será el trabajo continuo con el mismo lo que nos permitirá ir conociéndolo y aprendiendo a exprimirlo al máximo.

El capítulo siguiente trata de facilitar la primera toma de contacto con dicho entorno, por lo que va dirigido principalmente a aquellos que no lo conocen. Si has usado versiones previas de Delphi seguramente te bastará una lectura rápida a fin de familiarizarte con los aspectos que son nuevos en la versión XE2, especialmente los componentes FMX que se usarán.

Capítulo 2: Introducción al entorno

Para ser un manitas del bricolaje no basta con tener una buena caja llena de herramientas, también es necesario elegir la más adecuada para cada tarea y saber cómo utilizarla correctamente. A la hora de desarrollar aplicaciones con Delphi XE2 nuestra caja de herramientas es el entorno o IDE (*Integrated Development Environment*), repleto de opciones que conoceremos a medida que vayamos trabajando en la construcción de nuestros proyectos.

Este capítulo nos servirá para familiarizarnos con los elementos fundamentales del entorno, para lo cual desarrollaremos una sencilla aplicación que nos hará ir paso a paso por el proceso habitual de construcción de un proyecto, diseño, compilación, depuración, etc.

MicroClassic

Tenemos un cliente que cuenta entre sus aficiones el coleccionismo de ordenadores clásicos, concretamente microordenadores de las décadas de los 70 y 80 del siglo pasado, así como sus periféricos, soportes de almacenamiento, software y todo tipo de bibliografía de la época: libros, manuales, revistas, publicidad, etc. Necesita una aplicación específica para mantener toda la información asociada a su colección, debidamente clasificada y con opciones que le permitan realizar búsquedas, obtener listados, incluir fotografías, etc.

Por el momento nos concentraremos en los elementos más básicos de este proyecto, dejando de lado aspectos como que la información debería almacenarse en una base de datos, que la interfaz de usuario podría ser multiplataforma o incluso que la aplicación podría ser accesible a través de la web.

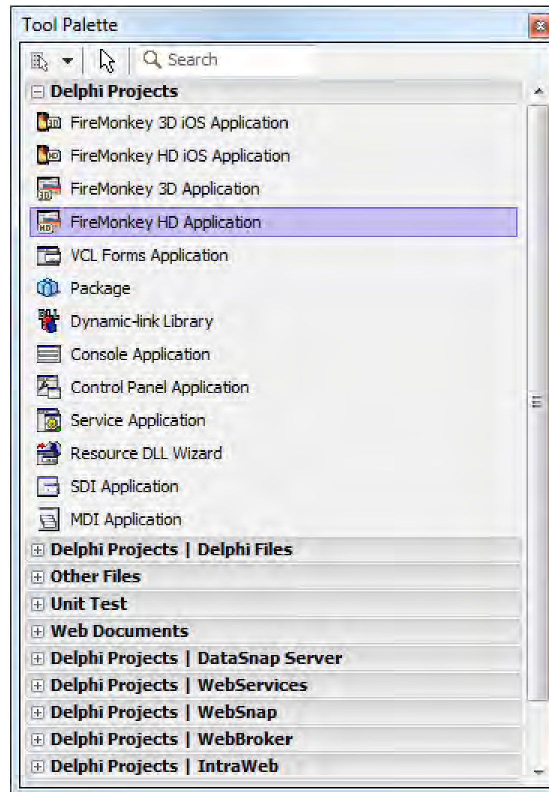
Obviamente habría que comenzar por el proceso de análisis de requisitos y diseño previo del sistema, pero dado que éstos no son temas que se aborden en este libro asumiremos que ya se han llevado a cabo y que disponemos de la información precisa para comenzar a desarrollar la aplicación.

Inicio del proyecto

Cada vez que iniciamos Delphi nos encontramos con la página de bienvenida y una serie de ventanas adicionales adosadas a los márgenes izquierdo y derecho del entorno. Cada una de ellas es una herramienta y puede mostrarse, ocultarse y moverse a la posición que más cómoda nos resulte sin más que usar la técnica de arrastrar y soltar.

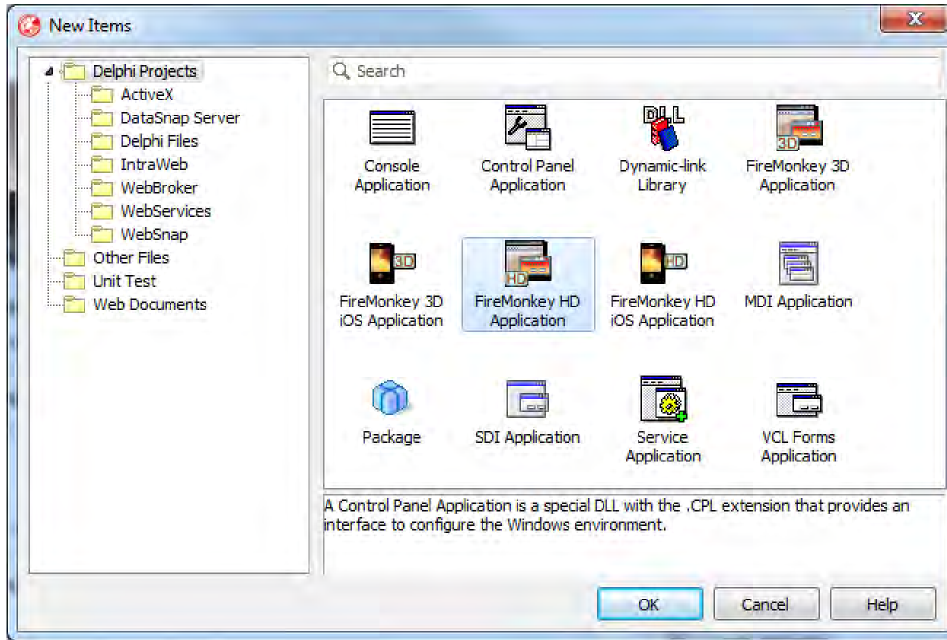
No es extraño que una misma función esté disponible en más de un punto del entorno: menú principal (parte superior), menús contextuales, botones, etc. Para iniciar un nuevo proyecto tenemos al menos tres posibilidades: las opciones del menú File>New, la opción New Project que aparece en la página de bienvenida (siempre que no la hayamos cerrado) y los elementos de la Paleta de herramientas (ventana Tool Palette inicialmente adosada en la parte inferior derecha del entorno).

Mientras no tengamos ningún proyecto abierto, en la Paleta de herramientas encontraremos (véase figura inferior) los distintos tipos de proyectos que podemos iniciar agrupados en múltiples categorías. No hay más que hacer doble clic sobre el elemento que interese para crear un proyecto nuevo.



En el menú File>New citado antes se ofrecen opciones para los tipos de proyectos más comunes. Siempre puede recurrirse a la opción Other de dicho menú para abrir el cuadro de diálogo New Items (figura de la página siguiente), en el que están todos los tipos de elementos existentes en nuestra edición de Delphi.

Como puede apreciarse en las dos figuras, ambas reflejan las plantillas disponibles para crear proyectos y agregar a éstos diferentes tipos de elementos, tendremos que elegir la categoría del proyecto: biblioteca de clases/componentes, servicio de Windows, aplicación web, aplicación con GUI (interfaz gráfica de usuario), programa de consola, etc. En algunas categorías, como es el caso de las aplicaciones con GUI, se ofrecen varias opciones: aplicación VCL, FMX para iOS, FMX HD y FMX 3D, entre otras.



Selección de la plantilla adecuada

Tenemos claro que nuestro cliente necesita una aplicación con GUI desde la que gestionar su colección, pero cabe preguntarse cuál de las plantillas existentes en dicha categoría deberíamos elegir. Tenemos tres plantillas que usan la biblioteca de componentes VCL: SDI Application, MDI Application y VCL Forms Application, y cuatro más que usan FMX: FireMonkey 3D Application, FireMonkey 3D iOS Application, FireMonkey HD Application y FireMonkey HD iOS Application.

Para hacer una elección adecuada deberíamos responder a las preguntas siguientes:

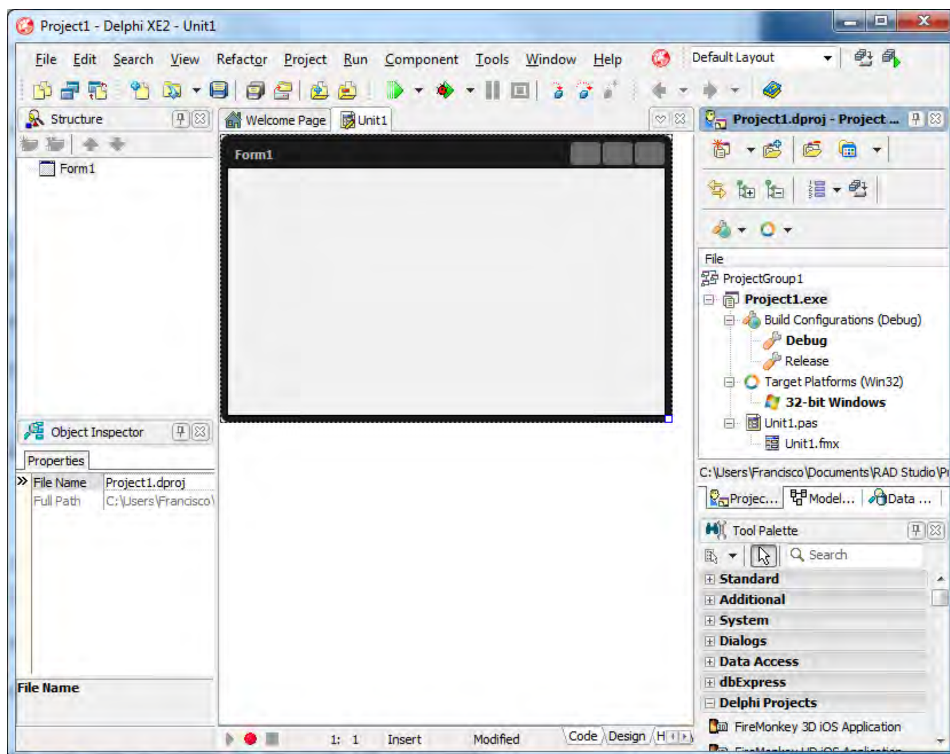
- ¿En qué plataforma ejecutará el cliente la aplicación?
- ¿Cambiará la plataforma en el futuro?
- ¿Precisa la aplicación efectos gráficos avanzados?

Si la plataforma de ejecución objetivo es Windows nos servirá cualquiera de las plantillas a excepción de las específicas para iOS. Éstas, como es fácil suponer, serían las apropiadas si la aplicación va a usarse en un iPad, iPhone o iPod. En caso de que la plataforma, actual o futura si se piensa en cambiar, sea MacOS X deberemos elegir necesariamente una plantilla FMX.

La única razón para elegir una plantilla VCL es que la aplicación vaya a ejecutarse únicamente en Windows, sin previsión de cambiar la plataforma en el futuro, y que conozcamos ya esta biblioteca o bien precisemos componentes que sean exclusivos de ella. En cualquier otro caso siempre es preferible optar por FMX, ya que se deja la puerta abierta a una posterior versión del proyecto para MacOS X y, además, esta biblioteca dispone de elementos visuales mucho más avanzados que la VCL.

Dado que el proyecto que abordamos tiene el objetivo básico de mantener información, facilitando su visualización y edición, no es previsible que necesitemos aplicar efectos 3D. Por ello elegiremos la plantilla FireMonkey HD Application.

Tras iniciar el proyecto, el entorno de Delphi debería mostrar un aspecto similar al de la imagen inferior. La ventana (formulario) que ocupa el área central es un contenedor, preparado para diseñar la interfaz de usuario de nuestra aplicación.



Configuración del proyecto

Todos los datos relativos a la configuración de nuestro proyecto están a nuestra disposición en el Gestor de proyectos (ventana Project Manager, inicialmente situada en la parte superior derecha). El nodo Project1[18] es el nombre asignado por defecto al único proyecto que tenemos abierto. En ese nodo existirán varios subnodos/ramas con más nodos hijo, en principio tres:

- Build Configurations: Cada nodo de esta rama almacena configuraciones de generación del proyecto, estableciendo parámetros que afectan al compilador de Delphi, el depurador, los paquetes incluidos en la aplicación, etc. Al menos existirán dos configuraciones distintas: una para la depuración y otra para la versión final del proyecto. Además cada configuración puede contar con más de un paquete de opciones, por ejemplo para cada plataforma para la que se genere el proyecto. En un momento dado sólo una de las configuraciones estará activa, apareciendo destacada en negrita.
- Target Platforms: Los nodos de esta rama definen las plataformas para las que se generará el proyecto. Al iniciar éste únicamente se configura la plataforma Win32 (32-bit Windows), pero pueden añadirse Win64 y MacOS X. Para cada plataforma puede establecerse un perfil remoto que haga posible la depuración y prueba del proyecto en una máquina distinta a la empleada para el desarrollo. Al igual que ocurre con las configuraciones, solamente una de las plataformas estará activa en cada momento.
- Unit1.pas: Este nodo representa el módulo de código asociado al formulario que contiene el proyecto. El propio archivo de definición del formulario aparece como nodo hijo. Como nodos hermanos (al mismo nivel que éste) irán apareciendo los elementos que se agreguen al proyecto: otros formularios, módulos de código, archivos de recursos, etc.

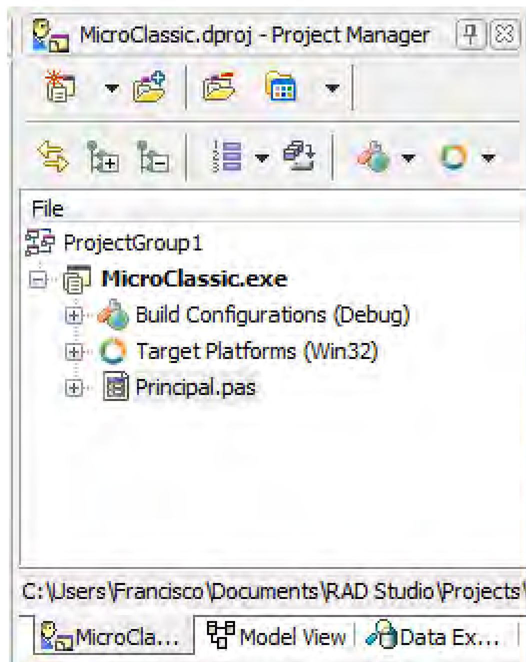
Tanto el nodo que representa al proyecto como cada uno de los nodos hijo cuenta con un menú contextual específico. En dichos menús están disponibles todas las operaciones que necesitaremos para configurar el proyecto.

---

18 El número final del nombre del proyecto, así como el asociado al formulario, podrían cambiar en caso de que en la carpeta de proyectos de Delphi ya hubiese guardados otros con ese nombre.

Guardar el proyecto y sus elementos

Los elementos que forman un proyecto recién iniciado se alojan en memoria, no se almacenan automáticamente en disco. La primera vez que se guarde cada componente en un archivo se establecerá su nombre, sustituyendo al que se asigna por defecto. Haciendo clic en el botón Save All, o bien usando la combinación de teclas Control-Mayús-S, Delphi abrirá un cuadro de diálogo en el que nos irá preguntando el nombre a dar a cada archivo, en nuestro caso el formulario podría llamarse `Principal` y el proyecto `MicroClassic`. Tras esta operación el Gestor de proyectos debería quedar como puede verse en la imagen inferior.



> NOTA
> Aunque todavía no hemos incluido ningún componente en el formulario, ni escrito código alguno, el proyecto puede ser compilado y ejecutado obteniendo una aplicación Windows simple. Pulsa F9 para comprobarlo.

Diseño de la interfaz de usuario

Conociendo lo básico sobre la administración de nuestro proyecto, vamos ahora a familiarizarnos con el diseñador de interfaces de usuario de Delphi. Es una necesidad, puesto que debemos componer la interfaz de la aplicación que estamos desarrollando para nuestro cliente.

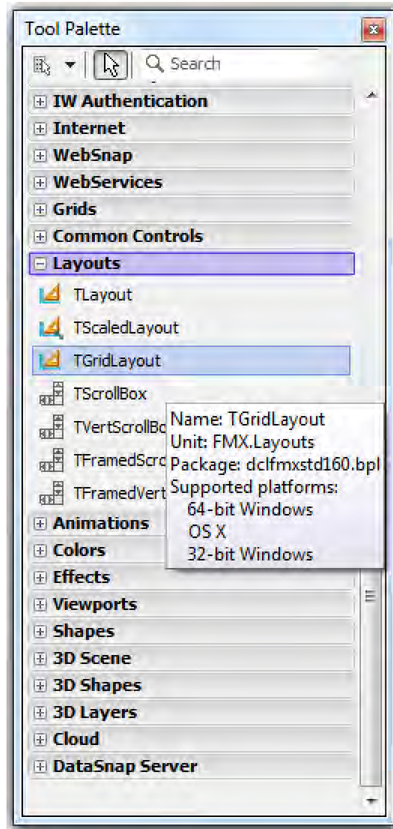
Como se indicaba antes, el formulario que aparece en el área central del entorno es un contenedor, un componente pensado para alojar a otros en su interior. En este caso dicho contenedor será la ventana principal del programa y, para actuar sobre ella, precisaremos básicamente dos herramientas de diseño: la Tool Palette, en la que se encuentran los componentes que podemos agregar, y el Object Inspector, que nos permitirá personalizar la apariencia y comportamiento de esos componentes.

En los siguientes apartados conoceremos algunos componentes y parte de sus propiedades. El objetivo es que nos familiaricemos con el método a seguir para diseñar la interfaz de una aplicación con Delphi, no describir de manera exhaustiva los componentes y sus características: propiedades, métodos y eventos. Ésta es información de referencia que puede encontrarse de manera inmediata en la documentación electrónica integrada del producto.

Búsqueda y selección de componentes

Todos los componentes de Delphi disponibles para la aplicación que estemos desarrollando los encontraremos en la Paleta de herramientas. Ésta, como puede apreciarse en la imagen de la página siguiente, cuenta con múltiples grupos desplegables. Cada uno de ellos contiene uno o más componentes. Basta con situar el puntero del ratón sobre cualquiera de ellos para conocer su nombre, el módulo y paquete que lo contienen y las plataformas para las que está preparado. Al hacer clic sobre él quedará seleccionado, de forma que pueda ser incluido en el formulario con otro clic.

El número de objetos que ofrece la Paleta de herramientas es considerable y, hasta que nos familiaricemos con la posición en que se encuentran, encontrarlos puede resultar un poco tedioso. Por ello en la parte superior de la ventana se ofrece un recuadro de búsqueda rápida: basta con escribir parte del nombre del componente para encontrarlo de inmediato.



Distribución de los componentes

En FMX prácticamente cualquier componente puede ser padre de cualquier otro, incluso es posible embeber un formulario en el interior de un panel de otro formulario. Existen, no obstante, componentes que han sido específicamente diseñados para actuar como contenedores, sin más funcionalidad que la de ayudar a la distribución de otros para facilitar la composición de la interfaz de usuario.

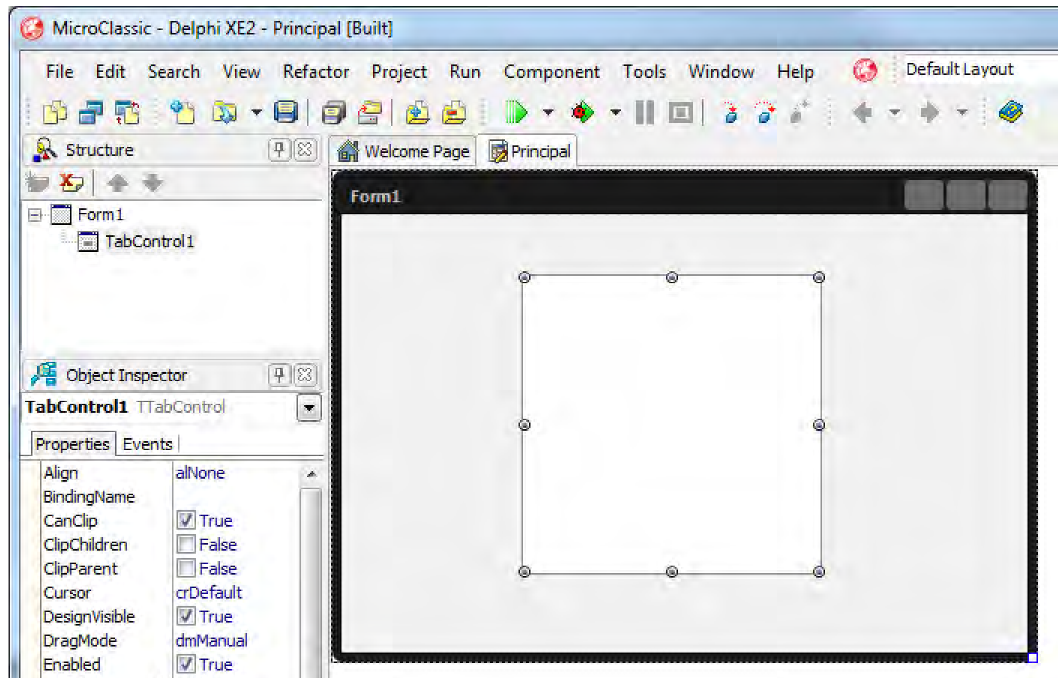
A estos contenedores les llamamos de segundo nivel por el simple hecho de que se introducen en un formulario que, lógicamente, se consideraría el primer nivel. Su finalidad puede ser la de agrupar elementos visualmente, mostrar contenidos más extensos que el espacio disponible en la interfaz, generar una interfaz dividida en múltiples páginas, etc.

Los componentes que nos interesan en este momento están alojados en varios de esos grupos, concretamente en Standard, Common Controls y Layouts. Pero vayamos paso a paso.

La información que ha de mantener la aplicación puede ser dividida en múltiples categorías claramente distintas, cada una de ellas con un conjunto de datos totalmente diferente. Ante esto tenemos básicamente dos opciones: diseñar una ventana específica para cada categoría o bien dividir una sola ventana en varias partes. Vamos a optar por la segunda.

> NOTA
>
> Aunque el diseñador con el que vamos a trabajar puede ser utilizado para la creación de prototipos, por regla general es más cómodo partir de una herramienta específica para tal fin y contar con un diseño previo de la interfaz antes de comenzar a trabajar con Delphi. Muchas de estas herramientas son accesibles desde el navegador como puede apreciarse en la imagen adjunta.

A fin de estructurar la interfaz en varias páginas independientes tendremos que agregar al formulario un componente `TTabControl` [19]. Una vez lo tengas localizado en la Paleta de herramientas sencillamente haz doble clic sobre él. Lo verás aparecer centrado en el formulario (como en la imagen inferior) y con un tamaño por defecto.



También se puede insertar un componente seleccionándolo en la Paleta de herramientas, con un único clic, y haciendo clic después en el punto del contenedor en el que quiera incluirse. En este caso la posición será la indicada y el tamaño el establecido por defecto. Mediante la técnica de arrastrar y soltar, antes de insertar el componente o a posteriori, podremos fijar también el tamaño. Esto no es algo que importe por el momento, ya que las dimensiones serán ajustadas automáticamente como veremos en un momento.

---

19 Dicho componente se encuentra en el grupo Common Controls. No obstante, lo más cómodo es usar la búsqueda rápida escribiendo el principio de su nombre. En lo sucesivo únicamente indicaré el nombre de los componentes, pero no la categoría en que se encuentre en la Paleta de herramientas ya que la búsqueda es inmediata.

Edición de propiedades

Para adaptar el TTabControl a nuestras necesidades deberemos modificar varias de sus propiedades, mediante un procedimiento que es siempre idéntico para todos los componentes de Delphi que vayamos a usar en el futuro. El punto clave se llama Inspector de objetos, la ventana con el título Object Inspector inicialmente adosada en la parte inferior izquierda del entorno.

El Inspector de objetos cuenta con dos páginas: Properties y Events. La primera contiene la lista de propiedades del objeto que se tenga seleccionado, cuyo nombre y tipo se muestran en la lista desplegable que hay en la parte superior, mientas que la segunda enumera los eventos que genera dicho objeto. Por el momento vamos a concentrarnos en las propiedades, dejando los eventos para después.

Una propiedad representa un atributo personalizable de un componente, pudiendo afectar fundamentalmente a su aspecto o a su comportamiento. En la mayoría de los casos el propio nombre de la propiedad, que aparece en la columna de la izquierda, deja clara su finalidad. La columna de la derecha muestra para cada propiedad el valor que tiene actualmente y ofrece un editor adecuado para cambiarlo[20]: un recuadro de texto, una lista desplegable con los valores permitidos, un botón de marcar/desmarcar, etc.

Teniendo seleccionado el componente TTabControl que incluimos antes, basta con hacer clic sobre él o bien desplegar la lista de la parte superior del Inspector de objetos y elegirlo, haremos los cambios siguientes:

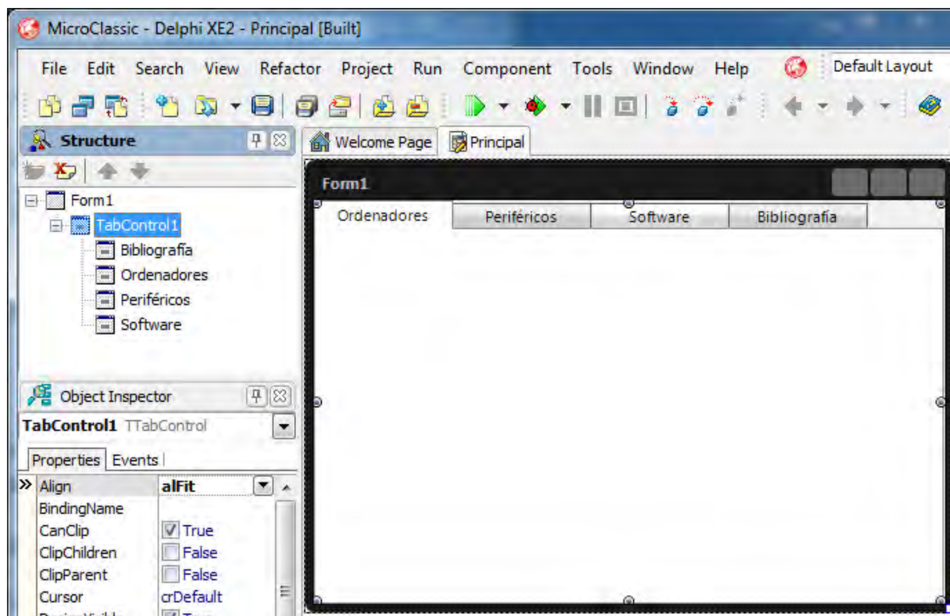
- Cambiar la propiedad Align, desplegando la lista asociada y asignándole el valor alContents. De esta forma el TTabControl ocupará todo el espacio disponible en su contenedor: el formulario.
- Abrir el menú contextual del TTabControl y, mediante la opción Add Item, agregar una página. También se puede usar la opción del mismo nombre que aparece en la parte inferior del Inspector de objetos. En cualquier caso aparecerá un nuevo componente: un TTabItem cuyo reflejo en la interfaz es una pestaña con un texto por defecto en su interior. Fíjate en que el Inspector de objetos muestra

20 Las propiedades de los objetos pueden establecerse durante la fase de diseño mediante el Inspector de objetos, pero también en el momento de la ejecución de la aplicación con el código adecuado como veremos después.

dicho componente como seleccionado, no el TTabControl [21] en el que está contenido.

- Modificar la propiedad Text del TTabItem recién agregado. Dicha propiedad contiene un texto cualquiera que será el que aparezca en la pestaña. Introduciremos Ordenadores como título.
- Cambiar la propiedad Width para ajustar el ancho de la pestaña. Esta propiedad almacena un número en coma flotante y la correspondencia con los píxeles en pantalla suele ser de uno a uno. Por tanto asignándole el valor 100 el ancho de la pestaña será de 100 píxeles.
- Elegir el valor taCenter para la propiedad TextAlign a fin de que el título de cada pestaña quede centrado en el ancho de ésta.

A excepción del primer paso, repetiríamos el resto para añadir las demás páginas a la interfaz consiguiendo que ésta quede como se muestra en la imagen inferior.

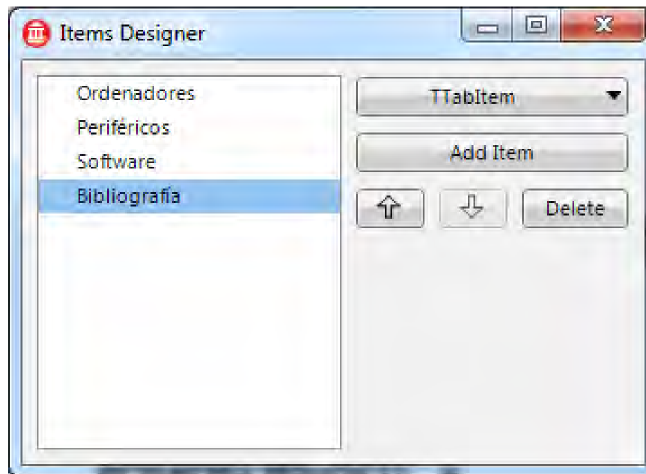


21 El compontente TTabControl es un contenedor especializado, diseñado para acoger un conjunto de componentes TTabItem cada uno de los cuales será una página. Los TTabItem son, a su vez, contenedores genéricos para otros componentes.

Propiedades como Text, Align o Width están presentes en la mayoría de los componentes[22] y siempre tienen la misma función, por lo que solamente tendremos que aprender a usarlas una vez.

Una vez agregados los TTabItem podemos hacer clic sobre ellos para cambiar la página activa. Dado que aún no hemos introducido nada en esas páginas, en todos los casos seguimos viendo un espacio vacío. En realidad al activar una de las pestañas lo que estamos haciendo es modificar la propiedad TabIndex del TTabControl, estableciendo la página que será inicialmente visible cuando se ejecute el programa. Puedes cambiar dicha propiedad directamente, en el Inspector de objetos, asignándole un número entre 0 y el número de páginas existentes menos 1.

Si queremos cambiar el orden de las pestañas o eliminar alguna de ellas lo más cómodo es abrir el Diseñador de elementos, sencillamente haciendo doble clic sobre el TTabControl. Se abrirá la ventana Items Designer (véase la imagen inferior) desde la que podremos seleccionar una página, desplazarla, eliminarla y añadir otras nuevas.



22 Los componentes, tantos los VCL como los FMX, están conforme a una jerarquía de herencia, de forma que unos están creados a partir de otros, heredando los primeros de los segundos características comunes. Tanto TTabControl como TTabItem, y muchos otros de los que conoceremos después, son componentes derivados de TControl y en consecuencia tienen en común los miembros definidos en dicha clase.

TRUCO
Una vez que se tiene seleccionado el `TTabControl` en el formulario, tras añadir las páginas, un clic sobre una pestaña no activa en el Inspector de objetos el `TTabItem` correspondiente. Sin embargo no es necesario abrir el Diseñador de elementos sólo para poder seleccionar una página, basta con hacer clic sobre ella en la ventana Structure o bien usando la lista desplegable del propio Inspector de objetos.

Estructura de la interfaz

Si bien en cada página/sección de la ventana habrá datos diferentes, específicos del tipo de objeto coleccionado, la interfaz tendrá una estructura homogénea, de forma que para el usuario resulte consistente y pueda aprender a usarla rápidamente. Cada página estará dividida en las siguientes zonas:

- Colocaremos una lista adosada al margen izquierdo con el nombre de cada objeto ya registrado, así como un botón en la parte inferior para añadir otros nuevos.
- En el área central colocaremos la ficha del objeto seleccionado en cada momento en la lista anterior, con los componentes que se precisen para recoger los datos correspondientes.
- En el margen derecho dispondremos un área vertical para contener las imágenes del objeto indicado en la ficha, facilitando el cierre/apertura de cada imagen y el desplazamiento vertical si el número de imágenes lo hiciese necesario.

Además de aportar coherencia de uso a la interfaz, el hecho de que todas las páginas compartan esta estructura también tiene un beneficio para nosotros como desarrolladores: una vez hayamos compuesto el diseño de la primera página podremos copiarlo a las restantes y después realizar los cambios que se necesiten, ahorrándonos así un tiempo importante.

Vamos, por tanto, a diseñar la interfaz para la primera de las secciones de la ventana, la correspondiente a los ordenadores, para lo cual insertaremos y personalizaremos un buen número de componentes FMX.

El primer paso será dividir la página en dos zonas independientes: la de la izquierda para la lista de ordenadores y la de la derecha para editar uno de ellos. Insertaremos en el TTabItem sendos componentes TLayout, sin que importe en principio la posición o dimensiones.

La única finalidad de los TLayout es facilitar el correcto ajuste de otros componentes insertados en él. Para ello se recurre fundamentalmente a tres propiedades: Align, Margins y Padding. La primera puede tomar uno entre una veintena de valores distintos, especificando cómo se establecerá la posición del componente en su contenedor y cómo se ajustará al espacio disponible. Por defecto Align toma el valor alNone y no se fijan automáticamente ni posición ni tamaño, aspectos que dependerán de las propiedades Position, Height y Width.

Selecciona el primer TLayout, despliega la lista de la propiedad Align (en el Inspector de objetos) y selecciona el valor alLeft. De esta forma ese panel se mantendrá siempre anclado al margen izquierdo, ocupando todo el alto disponible. Fija el ancho modificando la propiedad Width, a la que puedes dar el valor 120. A continuación cambia la propiedad Align del segundo TLayout dándole el valor alClient, consiguiendo así que ocupe el resto del espacio disponible y se ajusten automáticamente tanto la posición como el alto y el ancho del panel.

Aunque podríamos optar porque el ancho del panel izquierdo, el que contendrá la lista de ordenadores, sea fijo, no podemos anticipar cuál será la longitud del nombre más largo y por tanto cuál sería el ancho más adecuado. Lo más apropiado en estos casos es dejar que sea el usuario el que, en caso de necesidad, lo ajuste manualmente. Para ello agregaremos al TTabItem un componente TSplitter, un divisor vertical que el usuario pueda desplazar a izquierda y derecha. Puesto que el panel a controlar está anclado al margen izquierdo, daremos el valor alLeft a la propiedad Align del TSplitter. Además daremos a su propiedad MinSize el mismo valor de la propiedad Width del TLayout, 120, de forma que puede aumentarse y reducirse el ancho pero nunca quede tan estrecho que la lista resulte inservible.

> ADVERTENCIA
>
> Es de vital importancia que los TLayout y el TSplitter estén alojados dentro del TTabItem, teniendo a dicho componente como padre, ya que de lo contrario no desaparecerían al hacer clic en otra pestaña. La manera

más fácil de comprobarlo es usando la ventana Structure. Dentro de ésta puede utilizarse la técnica de arrastrar y soltar para mover los objetos de unos nodos a otros según interese.

El panel de la derecha estará dividido en un espacio central, con la información específica de un ordenador, y un área en el margen derecho que mostrará una lista de imágenes del ordenador elegido. Lo haremos básicamente siguiendo los mismos pasos de antes, si bien en este caso insertaremos en el TLayout de la derecha un TVertScrollBox, un TLayout y un TSplitter, dando los valores alRight, alClient y alRight, respectivamente, a la propiedad Align de cada uno. La propiedad Width del TVertScrollBox y MinSize del TSplitter tomarán ambas el valor 120.

El componente TVertScrollBox es un panel al que agregar contenido y que mostrará una barra de desplazamiento vertical cuando sea necesario. Se puede dar el valor True a la propiedad UseSmallScrollBars para sustituir las barras de desplazamiento habituales por otras más estrechas.

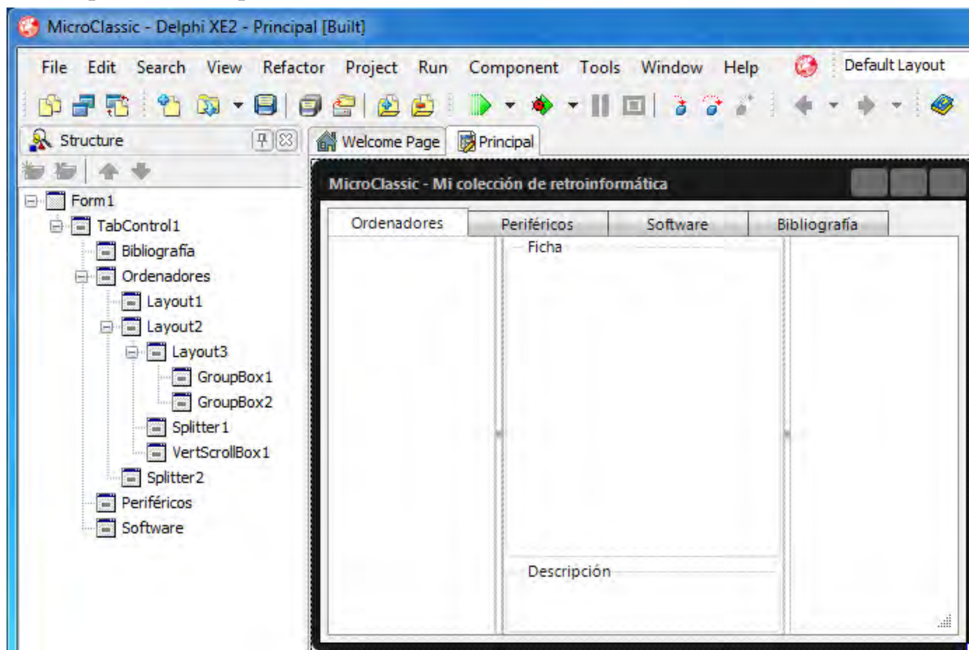
Para terminar con la estructura básica de la interfaz dividiremos nuevamente el TLayout central, reservando la parte superior para una ficha de características y la inferior para alojar una descripción textual. En este caso usaremos dos componentes TGroupBox, similares a TLayout pero que muestran un borde y un título. La propiedad Align del primero tendrá el valor alTop, ocupando la parte superior, y la del otro el valor alClient para que ocupe el resto del espacio[23]. Cambiando la propiedad Text de ambos estableceremos los títulos: Ficha y Descripción.

A fin de que los diferentes paneles no queden tan ajustados, tanto a los márgenes de la ventana como entre ellos, puedes editar las propiedades Margins y Padding. La primera establece la separación entre cada margen exterior del componente: superior, izquierdo, derecho e inferior, y los componentes situados alrededor. La segunda establece la separación interior, entre el margen del componente y su propio contenido.

---

23 Podrías preguntarte la razón de que demos el valor alClient al grupo inferior, en lugar de alBottom. Si lo hiciésemos el panel se ajustaría a la parte inferior, pero al cambiar el tamaño de la ventana se mantendría en esa posición en lugar de estirar su altura para ocupar todo el espacio existente. Si usásemos alBottom para el grupo inferior y alClient para superior, sería este último el que ajustaría su tamaño al cambiar las dimensiones de la ventana.

Tras modificar también la propiedad `Text` del formulario éste debería mostrar una apariencia similar a la de la figura inferior. Fíjate especialmente en el panel Estructura, donde se aprecia la relación entre todos los componentes. Puedes pulsar F9 para ejecutar el programa y comprobar cómo los paneles, aunque vacíos, se ajustan al cambiar el tamaño de la ventana. Al hacer clic en otra pestaña debería aparecer vacía, si no es así se debe a que alguno de los componentes no está insertado en el contenedor que le corresponde.



La lista de ordenadores

El panel izquierdo tendrá un diseño muy sencillo, componiéndose únicamente de un botón y una lista. El primero será un componente `TButton` y el segundo un `TListBox`, dos de los elementos más usados al crear interfaces de usuario.

Queremos que el botón permanezca en la parte inferior, por lo que una vez insertado en el `TLayout` editaremos su propiedad `Align` y le daremos el valor `alBottom`. Además cambiaremos también la propiedad `Text` escribiendo como título el texto `Añadir nuevo`.

Dado que queremos que la lista ocupe el resto del espacio del panel daremos el valor alClient a su propiedad Align. En principio la lista está vacía, los elementos se añadirían a medida que se usara la aplicación una vez terminada. No obstante vamos a agregar un par de elementos para conocer algunos aspectos sobre el componente TListBox.

Mediante la opción Add Item (se encuentra tanto en el menú contextual del TListBox como en la parte inferior del Inspector de objetos) agregaremos un elemento. Éste será un objeto TListBoxItem, es decir, otro componente que aparece como hijo del TListBox y, como componente, dispone de su propio conjunto de propiedades y eventos. Una de las propiedades es Text que cambiaremos del texto por defecto al valor Commodore 64. Repetimos la operación y agregamos otros elemento a la lista, cuyo nombre será Atari 400.

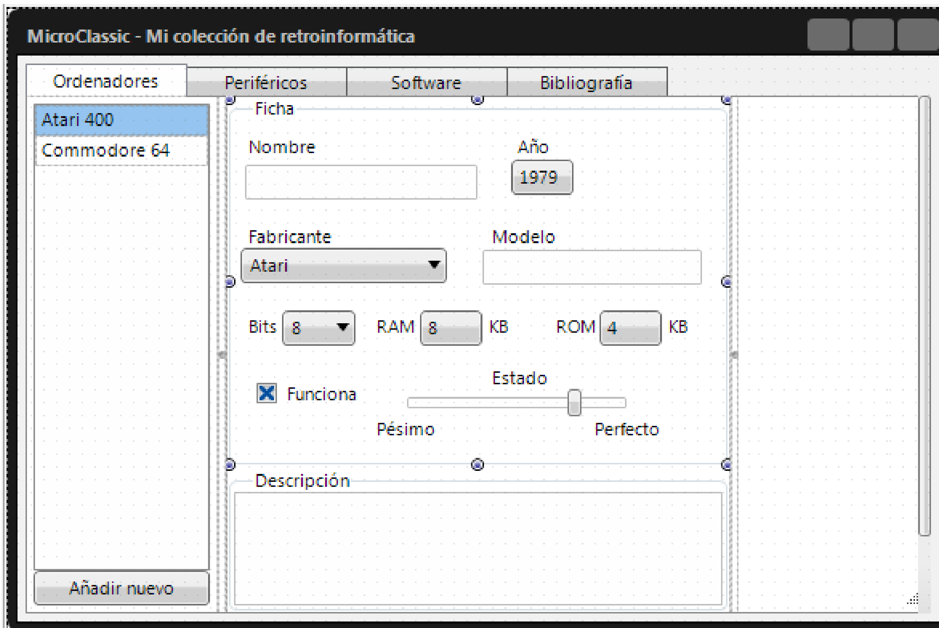
Estos objetos TListBoxItem, que ahora introducimos mediante las herramientas del diseñador de Delphi, la aplicación tendrá que crearlos desde código y facilitarlos al TListBox como veremos más adelante. Con solo estos dos elementos, sin embargo, ya podemos apreciar que el orden en que aparecen es el mismo en que se añaden a la lista. Cuando ésta tenga un número importante de objetos registrados localizar uno no será especialmente fácil. Al no existir un orden tendremos que recorrerla de principio a fin hasta encontrar lo que buscamos. Por suerte no hay más que dar el valor True a la propiedad Sorted del TListBox y su contenido se ordenará alfabéticamente, facilitando la tarea de búsqueda.

Que el usuario pueda elegir de la lista uno o más elementos es algo que depende de la propiedad MultiSelect. Por defecto tiene el valor False, por lo que al seleccionar un elemento automáticamente se desmarcará el que estuviese elegido en ese momento. Es el comportamiento que nos interesa en este caso, pero en otros podríamos necesitar la selección de varios elementos y para ello bastaría con dar el valor True a dicha propiedad. Cada TListBoxItem cuenta con la propiedad IsSelected, mediante la cual puede saberse si está seleccionado o no así como cambiar ese estado. Cuando MultiSelect es False también podemos recurrir a la propiedad ItemIndex del TListBox para saber el índice del elemento marcado.

En cuanto al botón que hemos colocado en la parte inferior, por el momento no tendrá funcionalidad alguna hasta que escribamos el código que ha de ejecutarse en respuesta a su pulsación. Para ello tendremos que conocer los eventos que genera, tema del que nos ocuparemos más adelante en este mismo capítulo.

Diseño de la ficha y descripción

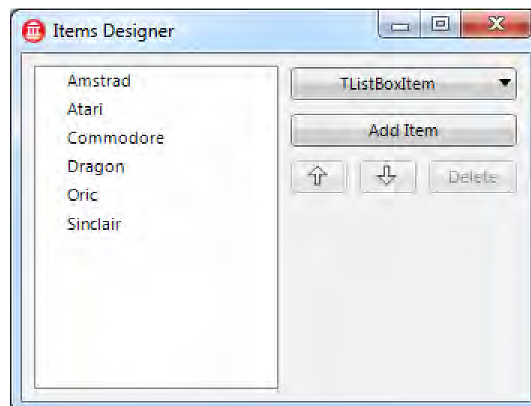
El área central de la página, concretamente el TGroupBox superior, será la que más elementos diferentes contenga tal y como puede apreciarse en la imagen inferior. Son una pequeña muestra de algunos de los componentes de uso habitual al diseñar interfaces de usuario con FMX. Veamos de qué componentes se trata y cuáles son sus propiedades fundamentales.



- TLabel : Permite colocar un texto allí donde interese, asignándolo a su propiedad Text y personalizando el estilo y tamaño con la propiedad Font si se desea. Suele utilizarse para colocar una etiqueta junto al resto de los elementos, salvo en el caso de aquellos que tienen la suya propia. En nuestro caso son TLabel todos los textos que se ven en la imagen anterior salvo la etiqueta Funciona del componente TCheckBox.
- TEdit: Su aspecto es el de un recuadro, al contar con un borde de forma rectangular. Inicialmente vacío, facilita la edición de un texto cualquiera en su interior. Dicho texto puede ser tanto recuperado como establecido mediante la propiedad Text. Dando el valor True

a la propiedad `Password` el texto introducido no será visible. En nuestra interfaz hay dos componentes de este tipo, uno para el nombre del ordenador y otro para el modelo.

- `TNumberBox`: Es un componente especializado para la introducción de números en un intervalo, establecido por las propiedades `Min` y `Max`. El valor actual se aloja en la propiedad `Value`. Además de poder editar el contenido, como si de un `TEdit` se tratase, también permite cambiar el valor simplemente arrastrando con el puntero del ratón a izquierda y derecha o usando las teclas arriba y abajo para incrementar/reducir. La propiedad `HorzIncrement` fija el factor de cambio en ese caso. Por defecto tiene el valor `1`. Usaremos tres `TNumberBox` para el año, la memoria RAM y ROM, con los límites `1965-1995` para el primero y `1-1024` para los otros dos.
- `TComboBox`: Al igual que un `TListBox`, este componente también contiene un conjunto de `TListBoxItem` de los cuales el usuario podrá elegir uno. La diferencia es que la lista se mantiene cerrada, mostrando el título del elemento seleccionado y una flecha en su extremo derecho. Al hacer clic en él se despliega la lista con un tamaño que permitirá mostrar tantos elementos como indique la propiedad `DropDownCount`. El índice del elemento elegido se almacena en la propiedad `ItemIndex`. La opción Add Item permite agregar elementos, como en el caso de `TListBox`, pudiendo también hacer doble clic sobre componente para abrir el Diseñador de elementos. En nuestra interfaz usamos dos de estos componentes: uno para mantener los fabricantes (véase imagen inferior) y otro con las posibles arquitecturas de los ordenadores (8, 16 ó 32 bits).



- TCheckBox: Este componente se usa cuando únicamente se necesitar saber si algo es cierto o falso, el usuario solo podrá marcar o desmarcar la opción. La propiedad Text contiene el título de ésta, no es necesario un TLabel adicional, y la propiedad IsChecked será True o False dependiendo del estado del componente. En nuestro diseño lo usamos para saber si el ordenador funciona o no.
- TTrackBar: Al igual que un TNumberBox, este componente facilita la introducción de un valor numérico entre un rango determinado por las propiedades Min y Max. La diferencia estriba en la representación visual como es obvio: un marcador que se desliza entre dos extremos. Cuando no importa tanto el valor exacto, porque la magnitud no es mensurable de manera directa sino una estimación, tiene más sentido usar un TTrackBar que un TNumberBox. En nuestro caso lo usamos para valorar el estado de conservación en que se encuentra el ordenador, desde 0 (pésimo) hasta 100 (perfecto). Los textos que hay en los extremos y sobre el TTrackBar son componentes TLabel.
- TMemo: La finalidad de este componente es facilitar la introducción de un texto relativamente extenso, compuesto de varias líneas, comportándose por lo demás como un TEdit. La propiedad Text contendrá todo el texto, pudiendo también accederse al mismo línea a línea mediante la propiedad Lines. Al seleccionar ésta en el Inspector de objetos verás un botón con puntos suspensivos en la parte derecha. Haz clic sobre él para acceder a un editor de texto simple. Este último componente no estará contenido en el primer TGroupBox sino en el segundo, el de la parte inferior, y puesto que queremos que ocupe todo el espacio en esa zona daremos el valor alClient a su propiedad Align. Modifica la propiedad Padding para evitar que el TMemo solape el título del TGroupBox en el que está contenido.

Insisto en lo importante que es que los componentes sean hijos del TGroupBox correspondiente, no del TLayout donde está el TGroupBox o incluso directamente el formulario, ya de lo contrario la interfaz no funcionará adecuadamente. Examina la ventana Structure para asegurarte de que todo es correcto.

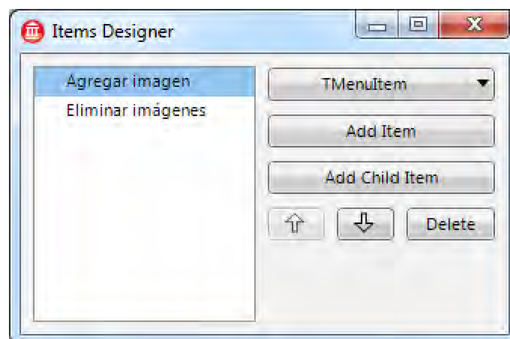
Siempre que tengas ya en el formulario un componente similar a otro que necesitas añadir, porque te interesen valores iguales para ciertas propiedades, puedes copiar al portapapeles y pegar donde convenga.

> TRUCO
> Si pegas un componente sobre el formulario es probable que no se inserte en el contenedor adecuado. Puedes utilizar los atajos para copiar y pegar directamente en la ventana Structure, seleccionando primero el elemento a copiar, pulsando Control-C y a continuación marcando aquél en el que desees pegar una copia del primero y pulsando Control-V. De esta manera no tendrás que mover los componentes al padre correcto después de haberlos pegado.

La lista de imágenes

La última sección de la interfaz, anclada al margen derecho de la ventana, estaría inicialmente vacía y su contenido lo agregaría el usuario a demanda. Para ello facilitaríamos un menú contextual, un elemento que no tiene un reflejo directo en el formulario. Lo que necesitamos es un componente TPopupMenu. Al insertarlo aparecerá como un pequeño icono. No importa el contenedor donde lo incluyamos, la asociación entre el menú y el elemento de la interfaz que lo mostrará se establece con la propiedad PopupMenu con la que cuentan todos los componentes visuales, incluido el TVertScrollBox que es el que nos interesa en este momento.

Una vez añadido el TPopupMenu tendremos que abrir el Diseñador de elementos, haciendo doble clic sobre el icono que representa al componente o bien con la opción Items Editor de su menú contextual. Agregaremos dos opciones, como puede verse en la imagen inferior.



Cada opción de un menú es un objeto `TMenuItem` y, entre otras, dispone de una propiedad `Text` que servirá para establecer el texto de la opción. Observa que en el Diseñador de elementos además del botón Add Item hay otro con el título Add Child Item. Éste sirve para convertir la opción elegida en ese momento en un submenú, con un conjunto de opciones de segundo nivel.

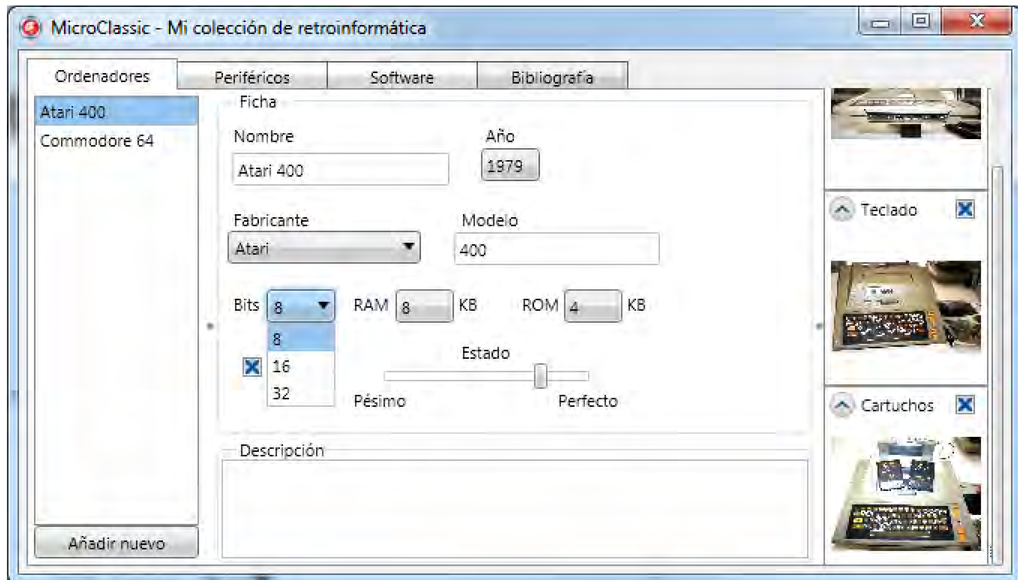
En cuanto al contenido del `TVertScrollBox`, cada vez que se quiera agregar una imagen se insertará en él un componente `TExpander` con un `TImage` en su interior. El primero es un contenedor que permite mostrar/ocultar su contenido con un pequeño botón, de forma que cuando está oculto solamente se ve un título que, como es habitual, asignaremos a la propiedad `Text`. La propiedad `IsExpanded` controla el estado, expandido o cerrado, del componente. A fin de que las imágenes se vayan disponiendo de arriba a abajo, daríamos el valor `alTop` a la propiedad `Align` del `TExpander`.

Cada imagen sería mostrada por un componente `TImage` que la alojaría en su propiedad `Bitmap`. Ésta cuenta con una opción Edit[24] que nos permite recuperar una imagen de archivo y ajustar su tamaño.

24 Inserta un `TImage`, selecciona su propiedad `Bitmap`, haz clic sobre el botón con puntos suspensivos y selecciona la opción Edit.

Con el objetivo de comprobar cómo quedaría la interfaz y cómo se comportaría el componente `TVertScrollBar` podemos agregar varios `TExpander` con sus `TImage` dentro. Éstos tendrían la propiedad `Align` establecida a `alClient`, ocupando así todo el espacio disponible en el `TExpander`.

El aspecto final de la interfaz será similar al mostrado en la imagen inferior. Tras pulsar F9, para compilar y ejecutar, prueba a cambiar el tamaño de la ventana y comprueba que los componentes se ajustan cada uno según lo establecido en sus respectivas propiedades `Align`. Además puedes abrir las listas desplegables, modificar los valores de las cajas de texto y campos numéricos, etc. Verifica asimismo que al cambiar a otras páginas de la ventana aparecen vacías, de no ser así habría que revisar en la ventana Structure que cada componente se halle en el contenedor que debería.



Si estableciste la propiedad `PopupMenu` del `TVertScrollBar`, al hacer clic con el botón secundario del ratón sobre este componente debería aparecer el menú contextual con las dos opciones que habíamos configurado. Puede resultar algo difícil, ya que la mayor parte del `TVertScrollBar` se encuentra ocupado por los componentes `TExpander` y estos, a su vez, por los `TImage`. La solución es sencilla: usar la propiedad `PopupMenu` de dichos componentes para asociarlo con el mismo menú contextual.

Gestión de eventos

Si bien la interfaz que hemos compuesto tiene una cierta funcionalidad básica, aportada por el comportamiento intrínseco de cada componente como es la edición de los TEdit y TMemo o la selección en los TComboBox y TListBox, lo cierto es que al cambiar el elemento elegido en la lista de la izquierda no cambia nada, la ficha no se actualiza. Tampoco hay respuesta al hacer clic en el botón que presumiblemente añade un ordenador nuevo, ni al elegir las opciones del menú contextual de la lista de imágenes.

Todas esas acciones: la activación de un elemento en una lista, el clic sobre un botón o la selección de una opción en un menú generan eventos, señales a las que, por el momento, no estamos respondiendo. La lista de eventos disponibles para cada componente es extensa pero, al igual que ocurre con las propiedades, muchos de ellos son comunes y en cualquier caso solamente precisaremos responder a un pequeño subconjunto de ellos.

¿Qué significa “responder a un evento”? Sencillamente establecer una conexión de forma que en cuanto se detecte la señal que lo genera se lance la ejecución de una porción de código (un método) propia. Se trata, por tanto, de elegir el evento que corresponde a la acción que nos interesa y escribir el código que llevará a cabo la tarea esperada por el usuario.

Método general

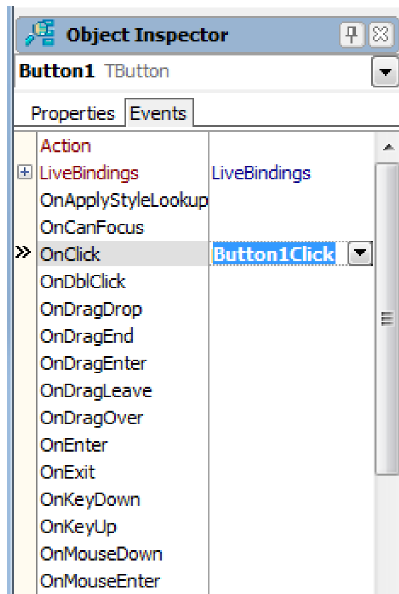
La mayoría de los componentes exponen múltiples eventos distintos y, por regla general, uno de ellos es al que se responde con mayor frecuencia, pudiendo hablarse de evento por defecto. En el caso del TButton, por ejemplo, ese evento se llama OnClick y se genera al pulsar el botón, ya sea con el ratón, la barra espaciadora si es el elemento activo, un atajo de teclado, etc. Si queremos responder a ese evento por defecto lo único que tenemos que hacer es un doble clic sobre el componente[25]. El diseñador responderá abriendo el Editor de código de Delphi, generando el esqueleto

25 No todos los componentes responden al doble clic asociando un método al evento por defecto y abriendo el editor de código. TListBox o TComboBox, por ejemplo, abren en su lugar el Diseñador de elementos. El resto de los que hemos usado hasta ahora sí lo hacen: TEdit, TNumberBox, TCheckBox, TTrackBar, etc.

del método correspondiente y colocando el cursor en la posición adecuada para que comencemos a escribir.

En cualquier caso, el método general para acceder a los eventos de un componente es abrir la página Events del Inspector de objetos. En principio la lista mostrará únicamente los nombres de los eventos disponibles, sin ningún valor asociado. A diferencia de las propiedades, no se establece un contenido por defecto para cada evento.

Para asociar un método a ejecutar cuando se genere un evento no hay más que seleccionar éste y hacer doble clic sobre él. Delphi abrirá el Editor de código y escribirá la cabecera del método por nosotros. Además en el Inspector de objetos se introducirá como valor del evento el nombre del nuevo método (véase la figura inferior).



El código insertado por Delphi se compondrá de una cabecera con el nombre del método, su lista de parámetros y las palabras clave begin y end delimitando el bloque de sentencias a ejecutar, inicialmente vacío:

```
procedure TForm1.Button1Click(Sender: TObject);
begin

end;
```

NOTA

Observa que en la parte inferior del entorno de Delphi hay unas pestañas con los nombres Code y Design. Con ellas podrás alternar entre el Editor de código y el Diseñador de interfaces. También puedes usar el atajo de teclado F12 con el mismo fin.

Si queremos asociar un evento con un método ya existente, en lugar de hacer doble clic desplegaremos la lista que hay a la derecha del nombre del evento y elegiremos el método que interese.

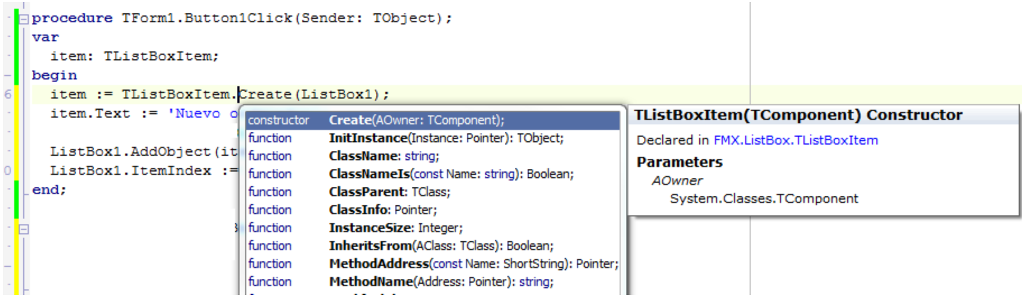
Al igual que ocurre con las propiedades, iremos familiarizándonos con los eventos de los componentes a medida que necesitemos utilizarlos. Sus nombres son indicativos del momento en que se producen y, como es lógico, en la ayuda electrónica del producto se facilita una referencia de todos ellos. En los puntos siguientes conoceremos algunos de los que precisamos para nuestro actual proyecto.

Agregar ordenadores nuevos

Lo primero que haremos será escribir algo de código en el interior del método que se ha abierto al hacer doble clic sobre el TButton, unas sentencias que crearán un nuevo TListBoxItem, lo agregarán a la lista y lo seleccionarán como elemento activo. El método quedaría así:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  item: TListBoxItem;
begin
  item := TListBoxItem.Create(nil);
  item.Text := 'Nuevo ordenador';
  item.Parent := ListBox1;

  ListBox1.AddObject(item);
  ListBox1.ItemIndex := item.Index;
end;
```

Las tres primeras sentencias crean el objeto y lo asignan a la variable item, usándola a continuación para modificar la propiedad Text y establecer como padre del elemento a la propia lista. Las otras dos añaden el objeto a la lista y a continuación recupera el índice que le corresponde para cambiar la propiedad ItemIndex del TListBox.



A medida que escribamos el código comprobaremos cómo el Editor nos asiste en la tarea, abriendo listas desplegables con las propiedades y métodos de los objetos, mostrando ayuda sobre los parámetros que toma un método, etc. Podemos usar la combinación Control-Barra espaciadora para abrir a demanda este tipo de ayuda cuando no aparezca automaticamente, por ejemplo tras escribir el principio del nombre de un componente para obtener la lista de los que se han definido y coinciden con ese prefijo.

Sincronizar la lista con la ficha (y viceversa)

Cada vez que cambiamos el elemento elegido en la lista de ordenadores el componente TListBox genera un evento OnChange. Sería el momento en el que deberíamos actualizar el contenido de la ficha del ordenador, recuperando de algún lugar los datos que habría que mostrar en los componentes TNumberBox, TComboBox, TEdit, etc.

Ya sabemos que el índice del elemento activo en la lista se encuentra en su propiedad ItemIndex. También podemos recuperar directamente el TListBoxItem que corresponda a dicho índice a través de la propiedad SelectedItem. Lo lógico sería almacenar la información de cada ordenador en el objeto TListBoxItem que lo represente en la lista. Por el momento conservaremos únicamente el nombre del ordenador y su año, sirviéndonos para ello de las propiedades Text y Tag[26] respectivamente.

---

26 La propiedad Tag está presente en casi todos los componentes y permite almacenar un valor NativeInt, un entero cuyo tamaño dependerá de la plataforma: 32 ó 64 bits.

Siguiendo el método general antes indicado, abrimos el método asociado al evento OnChange del TListBox e introducimos el código mostrado a continuación:

```
procedure TForm1.ListBox1Change(Sender: TObject);
begin
  Edit1.Text := ListBox1.Selected.Text;
  NumberBox1.Value := ListBox1.Selected.Tag;
end;
```

Con esto tendríamos la mitad del proceso de sincronización entre lista y ficha, ya que los cambios que se produzcan en ésta última no se reflejan en la primera. Los datos introducidos en el TEdit que contiene el nombre y en el TNumberBox que aloja el año deberían guardarse en el TListBoxItem activo en ese momento, efectuando las asignaciones opuestas a las hechas en el evento OnChange de la lista.

Ambos componentes también generan el evento OnChange, en este caso cada vez que cambia su contenido. En el caso del TEdit esto significa que mientras escribimos, cada pulsación de tecla produce dicho evento. Realmente no necesitamos guardar el nuevo valor continuamente, en tiempo real, basta con hacerlo cuando haya finalizado la edición. Por ello usaremos el evento OnExit que señala el momento en que se abandona un componente para pasar a otro, con independencia del mecanismo usado para ello: tecla Tab, clic de ratón, atajo de teclado, etc.

El método asociado al evento OnExit de cada componente tendrá una sola línea de código ejecutable: la asignación del nuevo valor introducido a la propiedad asociada del TListBoxItem como puede verse a continuación:

```
procedure TForm1.Edit1Exit(Sender: TObject);
begin
  ListBox1.Selected.Text := Edit1.Text;
end;

procedure TForm1.NumberBox1Exit(Sender: TObject);
begin
  ListBox1.Selected.Tag := Trunc(NumberBox1.Value);
end;
```

NOTA

La propiedad Value del TNumberBox almacena un número en punto flotante, mientras que la propiedad Tag contiene un entero. Por ello la asignación de la primera a la segunda requiere una conversión que

efectuamos con la función `Trunc()`. Ésta se limita a truncar la posible parte decimal del valor y devuelve únicamente la parte entera. La asignación en sentido contrario no representa ningún problema y por ello no se precisa conversión explícita.

Acciones de las opciones de menú

Ocupémonos ahora de las opciones del menú contextual que permitirán agregar y eliminar imágenes. Dichas opciones no están en la ventana del diseñador, pero sí en la ventana Structure. Elegimos el primer `TMenuItem`, asociado a la opción encargada de agregar una nueva imagen, y hacemos doble clic sobre ella. Veremos que el evento por defecto es `OnClick` como en el caso de los botones.

Tendremos que facilitar al usuario un medio para que pueda elegir el archivo en el que se encuentra la imagen a añadir a la ficha. Para ello agregaremos al formulario un componente `TOpenDialog`. Se trata de un elemento que no tiene parte visual y su objetivo es abrir un cuadro de diálogo de selección de archivo. Sirviéndonos una vez más del Inspector de objetos cambiaremos tres propiedades de este componente:

- `Filter`: Sirve para filtrar los archivos que aparecerán en el cuadro de diálogo, estableciendo uno o más patrones separados por punto y coma. En nuestro caso le asignaremos el valor siguiente para contemplar los formatos gráficos más habituales.

  ```
  *.jpg;*.gif;*.png;*.tif
  ```

- `InitialDir`: Fija el directorio que se abrirá inicialmente en el cuadro de diálogo. Usaremos el valor `%USERPROFILE%` para situarnos en la carpeta del usuario que ejecuta el programa.
- `Options`: Es una propiedad compuesta de un conjunto de valores booleanos, de lo cuales activaremos `ofFileMustExist` para forzar la elección de un archivo que exista.

Una vez tenemos el componente `TOpenDialog` preparado, volveremos al Editor de código de Delphi para escribir las sentencias que han de ejecutarse al elegir la opción. Serían las siguientes:

```
procedure TForm1.MenuItem1Click(Sender: TObject);
```

```
var
  expander: TExpander;
  image: TImage;
begin
  if OpenDialog1.Execute then
  begin
    expander := TExpander.Create(nil);
    image := TImage.Create(nil);

    expander.Align := TAlignLayout.alTop;
    expander.Text :=
ExtractFileName(OpenDialog1.FileName);
    expander.Parent := VertScrollBox1;
    image.Bitmap.LoadFromFile(OpenDialog1.FileName);
    image.Align := TAlignLayout.alClient;
    image.Parent := expander;

    expander.AddObject(image);
    VertScrollBox1.AddObject(expander);
  end;
end;
```

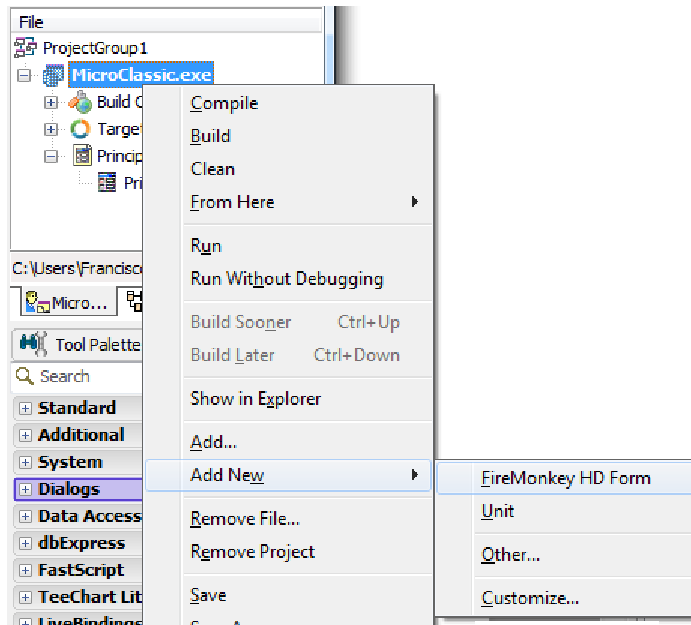
El método `Execute` de `TOpenDialog` abre el cuadro de diálogo y, en caso de que el usuario seleccione un archivo, devuelve el valor `True`. En ese caso creamos un `TExpander` y un `TImage` y, básicamente, los configuramos como habríamos hecho con el diseñador, pero estableciendo las propiedades y añadiendo los componentes mediante código en lugar de hacerlo con el Inspector de objetos.

Dado que las imágenes están asociadas a la ficha de cada ordenador, además de añadirlas al `TVertScrollBox` también habría que conservarlas junto con el resto de los datos, algo que dejaremos para más adelante. La opción de eliminación de las imágenes actuaría sobre esos datos y, de forma secundaria, también los eliminaría de la ventana liberando los `TImage` y `TExpander`.

Un visor para las imágenes

La mayoría de las aplicaciones además de una ventana principal y cuadros de diálogo auxiliares, que son los elementos con que cuenta actualmente nuestro proyecto, también recurren a ventanas secundarias para ciertas tareas. Nosotros usaremos una ventana adicional como visor de imágenes.

El objetivo es que nos familiaricemos con los pasos que hay que seguir a la hora de agregar formularios y cómo usar unos desde otros, aplicándolo de manera práctica sobre nuestro proyecto. El primer paso será abrir el menú contextual asociado a ese proyecto, como puede verse en la imagen inferior, y elegir la opción Add New>FireMonkey HD Form.

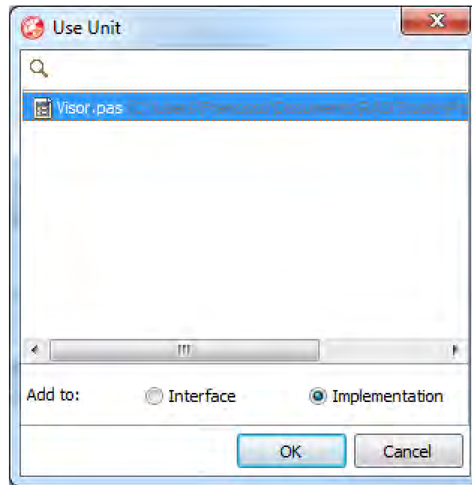


En el diseñador se abrirá el nuevo formulario, con el nombre `Form2`, vacío como lo estaba el formulario con el que iniciábamos el proyecto al principio de este capítulo. El único componente que vamos a insertar en él es un `TImageViewer`, a cuya propiedad `Align` daremos el valor `alClient` para que ocupe toda la ventana.

También modificaremos la propiedad `Caption` del nuevo formulario, eliminando su contenido de forma que sea una ventana sin título, y cambiaremos la propiedad `BorderStyle` dándole el valor `bsSizeToolWin`, consiguiendo así que el borde de este formulario sea algo más estrecho de lo normal, pareciéndose a las típicas ventanas de herramientas.

Hechos estos cambios guardaremos el nuevo formulario, al que podemos llamar `Visor`. Hay que tener en cuenta que ése sería el nombre del archivo en que se almacena, no el nombre del formulario en sí que continúa siendo `Form2` puesto que no lo hemos modificado.

Volviendo al formulario principal del proyecto, tendremos que agregar una referencia al segundo para poder utilizarlo. Lo hacemos mediante la opción File>Use Unit que da paso al cuadro de diálogo inferior. No hay más que elegir el único elemento disponible y hacer clic en el botón OK.



Queremos que la ventana secundaria, que nos permitirá ver mejor las fotografías, se abra al hacer doble clic sobre cualquiera de las miniaturas que hay en la lista derecha del formulario principal. Selecciona en la ventana Structure los tres componentes TImage, a continuación abre la página Events del Inspector de objetos y haz doble clic sobre el evento OnDblClick. De esta forma los tres componentes compartirán un mismo método de respuesta a ese evento. Dicho método sería el siguiente:

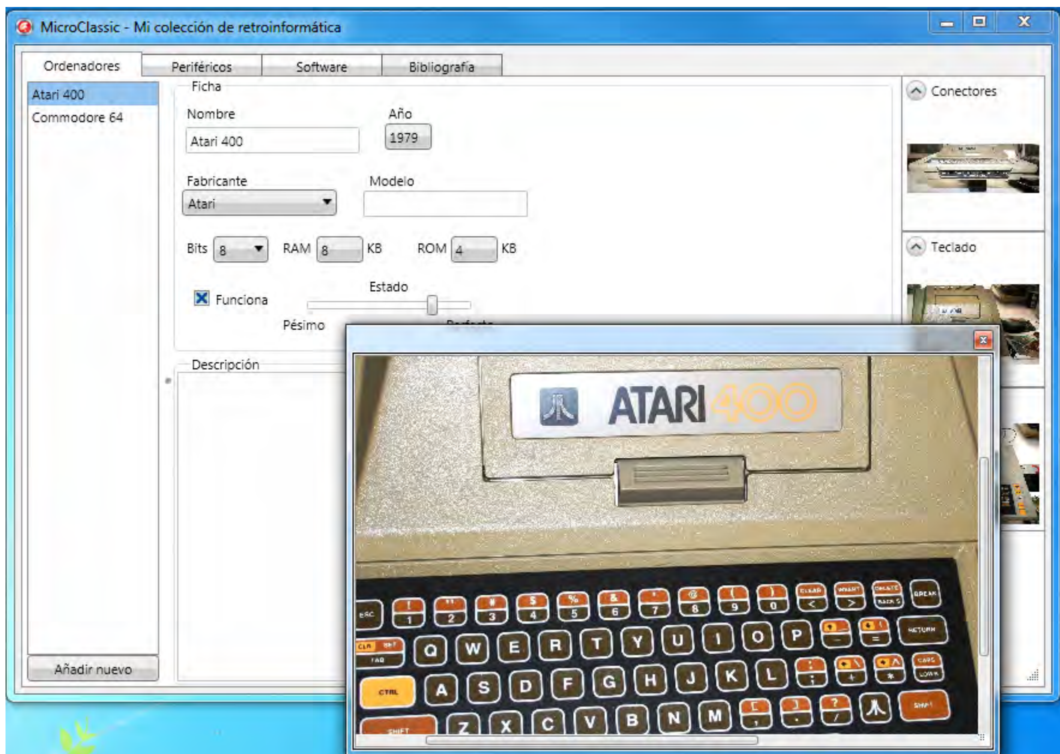
```
procedure TForm1.Image4DblClick(Sender: TObject);
begin
  Form2.ImageViewer1.Bitmap := (Sender As TImage).Bitmap;
  Form2.ShowModal;
end;
```

Con la primera sentencia asignamos la imagen sobre la que se ha hecho doble clic, almacenada en la propiedad Bitmap del TImage, a la propiedad homónima del TImageViewer. Puesto que todos los TImage comparten el mismo gestor de evento, usamos el parámetro Sender que recibe el método para saber qué TImage lo generó. La segunda muestra la ventana secundaria de forma modal, lo que significa que no podremos volver a la principal hasta que no cerremos ésta.

A fin de que las imágenes que el usuario agregue respondan también al doble clic abriendo el visor, tendremos que añadir la siguiente línea de código al final del método asociado al evento `OnClick` de la opción del menú contextual:

```
image.OnDblClick := Image4DblClick;
```

Ahora ya estamos en disposición de probar la aplicación. Pulsamos F9 y hacemos doble clic sobre una de las imágenes. Veremos abrirse el visor en la ventana secundaria. Usando el ratón podremos desplazar la imagen en las cuatro direcciones y con la rueda cambiar el factor de zoom.



Como puede comprobarse, gracias a los componentes y diseñadores de Delphi componer una aplicación con múltiples ventanas y una interfaz relativamente compleja, dividida en varias páginas cada una con paneles adosados y redimensionables, es una tarea realmente sencilla. Si tuviésemos que implementar toda la funcionalidad escribiendo código precisaríamos varias miles de líneas y mucho tiempo de trabajo.

Herramientas de depuración

El proyecto en el que trabajamos apenas tiene código y, por ello, es improbable que contenga fallos más allá de los errores de sintaxis y similares que detecta el compilador y son corregidos de manera inmediata. En aplicaciones reales, sin embargo, la probabilidad de que el comportamiento del programa en ejecución no se ajuste a lo esperado crece de manera proporcional a la cantidad de código y su complejidad.

Hay que partir de la detección de los potenciales problemas, una tarea que debe realizarse en la fase de desarrollo sin esperar a que sea el usuario el que los encuentre y los notifique. Por ello como parte del ciclo de diseño y desarrollo se recurre a pruebas de caja negra/blanca y la aplicación de pruebas unitarias a todas y cada una de las funciones implementadas, verificando que la respuesta es correcta en todos los casos.

Cuando se detecta un problema, y el fallo del que proviene no es obvio, la herramienta más útil es el depurador que, en el caso de Delphi, está integrado en el entorno[27]. La configuración por defecto de un proyecto es Debug y, mientras no se cambie en el Gestor de proyectos, el compilador incluirá en la aplicación compilada la información necesaria para facilitar la depuración. Esto nos permitirá emplear las opciones que se describen a continuación.

Puntos de parada

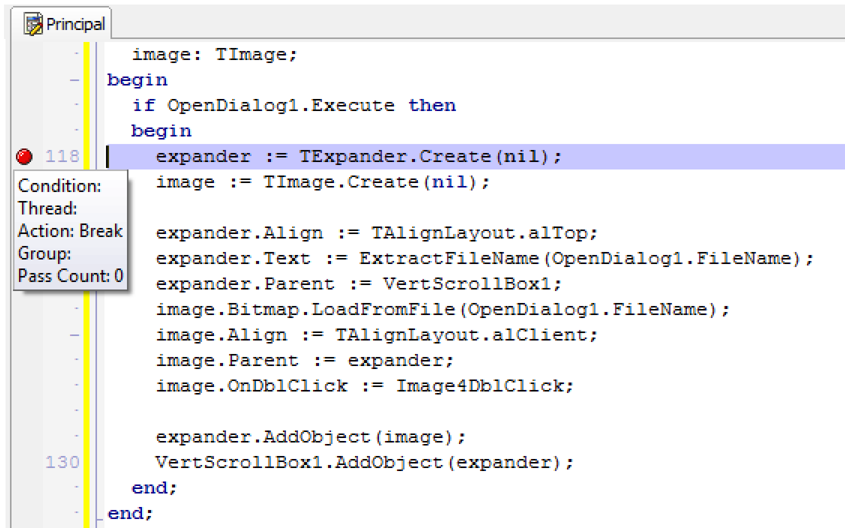
Al lanzar la ejecución del proyecto con la tecla F9, asociada a la opción Run del menú del mismo nombre, en principio la ejecución no se detendrá salvo que se encuentre un fallo. En ese caso aparecerá una ventana indicando el problema y dándonos opción a parar (botón Break) o continuar como si nada hubiera ocurrido (botón Continue).

Si el programa no hace lo que se espera de él, pero no genera un fallo de ejecución, la mejor forma de detenerlo para su depuración es situar un punto de parada allí donde prevemos que está el problema. No hay más que

---

27 Este depurador integrado está preparado para comunicarse a través de red y facilitar así la depuración remota de aplicaciones que se ejecutan en otras plataformas. Aquí, no obstante, nos centraremos por el momento en la depuración local del proyecto.

hacer clic con el puntero del ratón situado en el margen izquierdo de la línea que interese. Como se aprecia en la imagen inferior, aparecerá un círculo rojo en ese punto y la línea se marcará con un color distinto.



Éste sería un punto de parada simple, lo que significa que detendrá la ejecución del programa siempre que se llegue a la sentencia marcada. Mediante la opción Run>Add Breakpoint se pueden configurar puntos de parada más específicos de distintos tipos, por ejemplo para detener la ejecución cuando se haya pasado un cierto número de veces por la sentencia o se cumpla una condición determinada. Por el momento nos servirá uno simple colocado en la sentencia mostrada en la imagen anterior.

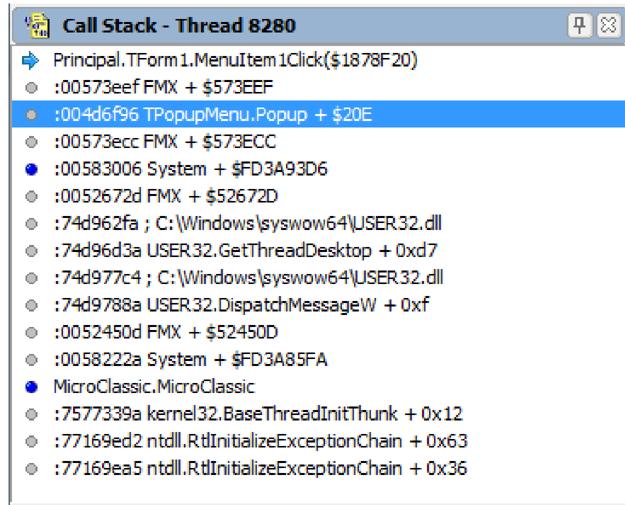
Estado del proceso

Tras colocar el punto de parada lanzamos la ejecución, pulsando F9, y elegimos la opción de agregar una imagen del menú contextual del panel derecho de la aplicación para provocar así que se llegue a la sentencia en la que está el punto de parada. En cuanto seleccionemos una imagen se activará el depurador, mostrando la sentencia activa.

Aunque la ejecución del programa se ha detenido la imagen del proceso asociado se encuentra almacenada en memoria, junto con todos los datos de estado que hacen posible continuar cuando interese. Además veremos que

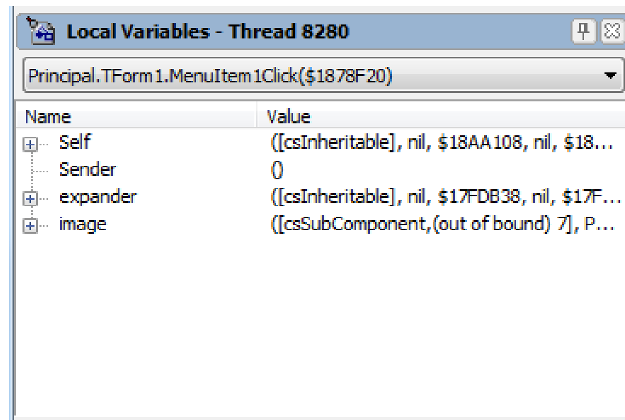
en el margen izquierdo se han abierto una serie de paneles específicos para la tarea de depuración, con información útil para determinar cómo se ha llegado hasta el punto actual, cuál es el contenido de las variables locales y otros objetos, etc.

La ventana Call Stack muestra la pila de invocaciones que ha seguido la aplicación desde su inicio (la línea inferior) hasta llegar al punto en que se encuentra (primera línea, marcada con una flecha en el margen izquierdo). En casos en los que un método puede ser alcanzado por distintas vías, esta información nos permitirá saber exactamente cómo se ha llegado hasta él. Como se aprecia en la imagen inferior, se incluyen las conexiones entre el sistema operativo, que detecta el clic de ratón que abre el menú contextual y lo despacha al hilo de ejecución de la ventana correspondiente, y la aplicación, encargada de invocar al método Popup para abrir el menú.



El menú contextual de cada línea de esta ventana facilita opciones para acceder al código fuente correspondiente o, en su ausencia, al código ensamblador ejecutado.

Bajo la anterior están los paneles Watches y Local Variables. El primero estará vacío y el segundo mostrará, como en la imagen siguiente, el contenido de las variables locales al método en que está la ejecución. Dado que no tenemos variables simples (números, cadenas, etc.) sino solamente objetos, el contenido es de interpretación algo más compleja.



Ejecución paso a paso

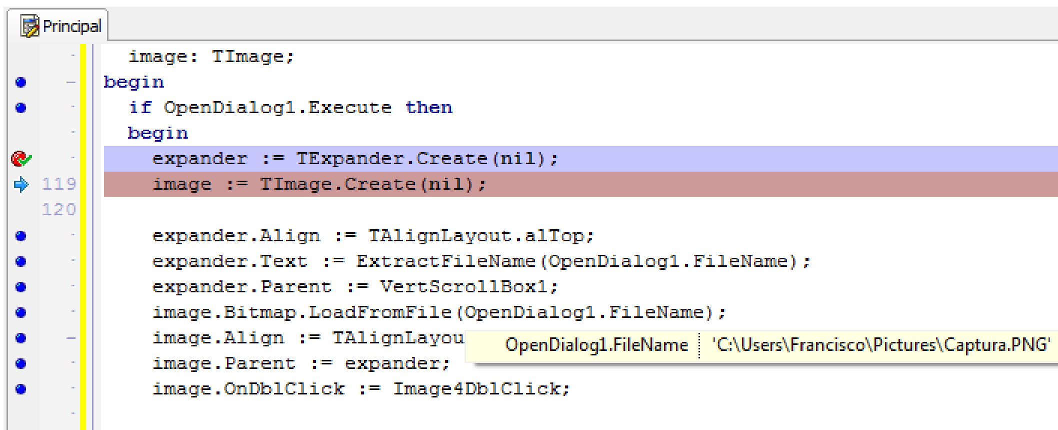
El estado de la aplicación, y por tanto el contenido de las anteriores ventanas, irá cambiando a medida que avancemos en la ejecución del código. Tenemos distintas opciones a nuestra disposición para ejecutar una o más sentencias. Las más usuales, todas disponibles en el menú Run, son las cuatro siguientes:

- Step Over: Ejecuta la sentencia resaltada en el editor como una unidad atómica, es decir, en un único paso.
- Trace Into: Como la anterior, ejecuta la sentencia resaltada pero no en un único paso en caso de que contenga llamadas a métodos. En ese caso se pasará a la primera línea del método invocado a fin de facilitar su depuración.
- Run to Cursor: Ejecutará todas las sentencias desde el punto en que esté detenido el proceso hasta la línea en que tengamos colocado el cursor. Es útil cuando se quiere saltar un bloque de código que se sabe no tiene problemas.
- Run Until Return: Normalmente se usa tras haber entrado en un método con Trace Into y su finalidad es ejecutar hasta finalizarlo y volver al punto en que se le invocó.

Las tres primeras opciones tienen asociados atajos de teclado muy sencillos: F8, F7 y F4 respectivamente. En cuanto tengamos que llevar a cabo una sesión de depuración real nos acostumbraremos rápidamente a utilizarlos en lugar de recurrir a los botones o a las opciones de menú. La cuarta también puede ejecutarse mediante el atajo Mayúsculas-F8.

Examinar el contenido de variables

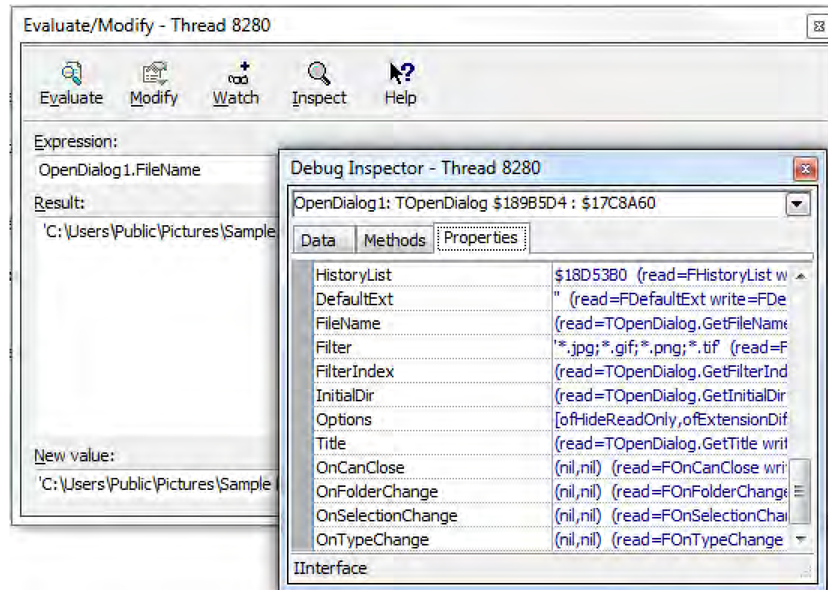
Antes se ha citado la ventana Local Variables como el lugar donde puede consultarse el valor de cualquier variable local o parámetro recibido por el método. En realidad, basta con colocar el puntero del ratón sobre el nombre de una variable, una propiedad o incluso un objeto para obtener en una pequeña ventana emergente su contenido. En la ventana inferior, por ejemplo, se ha colocado el puntero sobre la propiedad FileName del TOpenDialog a fin de saber cuál es la ruta del archivo elegido.



Si el dato que queremos conocer no aparece en el código, por ejemplo una propiedad de un objeto o incluso una expresión de acceso a elementos de una colección, podemos recurrir a la opción Run>Evaluate/Modify (Control-F7). Ésta da paso a una ventana como la que se ve en segundo plano en la imagen de la página siguiente, con un recuadro en el que podemos escribir cualquier expresión obteniendo su resultado debajo. Además podemos modificar el valor de variables y propiedades, facilitando así la realización de comprobaciones sin necesidad de detener la depuración, cambiar el código y volver a depurar.

Cuando la variable o propiedad a examinar no contiene un valor simple sino una referencia a un objeto, la información mostrada por la opción anterior puede resultar insuficiente. En ese caso el botón Inspect que hay en su parte superior abrirá el Debug Inspector, una ventana que permite inspeccionar todo el estado interno del objeto. Con el botón Watch agregaríamos la expresión utilizada a la ventana Watch List mencionada anteriormente, lo

cual resulta útil si queremos vigilar continuamente el contenido de una variable o propiedad sin necesidad de estar recurriendo una y otra vez a las opciones que acaban de describirse.



> NOTA
>
> Durante la sesión de depuración se puede usar en cualquier momento la opción Run (F9) para seguir ejecutando el programa normalmente.

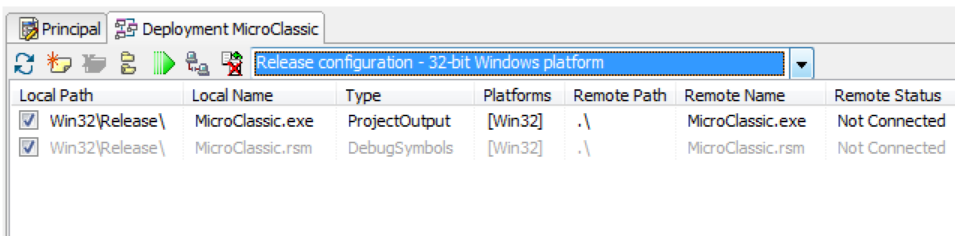
Despliegue de la aplicación

El proyecto en el que estamos trabajando es un ejemplo sencillo y no tiene requerimientos especiales para su despliegue e instalación en el equipo del hipotético usuario. Basta con hacer doble clic en la configuración Release (en el Gestor de proyectos), compilar y llevar el ejecutable a donde se quiera.

No obstante Delphi incorpora las opciones necesarias para configurar la tarea de despliegue y llevarla a cabo automáticamente. Ése es el objetivo de

la opción Run>Deployment que, por el momento, vamos a conocer muy superficialmente.

La citada opción abrirá una nueva página en el área central del entorno (como la de la imagen inferior). La lista permite seleccionar cualquiera de las configuraciones definidas, mientras que los botones dispuestos a su izquierda sirven para agregar archivos y paquetes requeridos por la aplicación, por ejemplo controladores de bases de datos; conectar con la máquina remota en la que se desplegará la aplicación y llevar a cabo el despliegue.



El despliegue e instalación de la aplicación cerraría el ciclo que comenzábamos al inicio de este capítulo, con la creación de un nuevo proyecto. Habría que decir que en realidad la vida de la aplicación comienza en este momento, una vez que se empieza a usar en su entorno de explotación real, y lo habitual es que precise un trabajo de mantenimiento y actualización mucho más extenso que el de diseño y desarrollo iniciales.

Conclusión

Todo el recorrido hecho en este capítulo, desde el inicio de un nuevo proyecto, el diseño de su interfaz, edición de código y depuración hasta llegar al despliegue final, nos ha servido para tomar contacto con un gran número de elementos del entorno de Delphi que, a la postre, era el objetivo del capítulo.

Ya sabemos usar las herramientas fundamentales de este entorno, si bien quedan muchas funciones aún por explorar. Encontrándote en el diseñador de interfaces de usuario abre el menú Edit, en él encontrarás opciones para para alinear los componentes, establecer el orden en que irán tomando el

foco de entrada al usar el tabulador, etc. El menú contextual del Editor de código está también repleto de opciones mediante las que podrás establecer marcas, dar formato automáticamente al código, buscar la declaración de cualquier método o variable, llevar a cabo tareas de refactorización, etc.

Si bien optamos desde un principio por usar la biblioteca de componentes FMX, en lugar de la clásica VCL, el proceso que se ha seguido sería prácticamente idéntico y encontraríamos diferencias básicamente en los componentes: sus nombres, propiedades y eventos con que cuentan. No obstante casi siempre es posible encontrar equivalencias entre ambas bibliotecas, lo cual abre las puertas a la conversión de aplicaciones VCL a FMX o viceversa.

Un ejemplo concreto: en la VCL no hay un componente `TLayout` que facilite la distribución de los elementos de la interfaz, pero podríamos usar el componente `TPanel` con la misma finalidad. En la VCL los componentes no disponen de tantos valores en la propiedad `Align`, ya que su objetivo es alinear los componentes pero no modificar sus dimensiones. Sin embargo los valores para `Align` de la FMX pueden suplirse con la propiedad `Anchors` que sí tienen los componentes de la VCL.

A continuación

Una vez que conocemos el entorno de Delphi XE2, con el que nos familiarizaremos paulatinamente a medida que trabajemos con él, nos falta otro pilar fundamental para poder crear aplicaciones con esta herramienta: aprender a usar su lenguaje de programación.

Como se explicó en la introducción, el lenguaje Delphi es heredero de Object Pascal y por tanto está basado en Pascal, pero con una serie de características avanzadas que le diferencian claramente y que le hacen merecedor de tener un nombre propio.

El objetivo del siguiente capítulo es introducir las características más importantes del lenguaje Delphi, asumiendo que el lector ya tiene conocimientos previos de cualquier otro lenguaje y también fundamentos sobre programación orientada a objetos.

Capítulo 3: Introducción al lenguaje

A lo largo del desarrollo del proyecto usado como ejemplo en el capítulo previo hemos escrito algo de código, unas pocas sentencias introducidas en métodos cuyos nombres y parámetros ya están preestablecidos. La mayoría de esas sentencias han sido simples asignaciones de unas propiedades a otras, siendo más importante conocer la finalidad de las propiedades de origen y destino que cualquier otra cosa.

En este capítulo se introducen las construcciones más importantes de Delphi, el lenguaje de programación empleado por Delphi XE2, pero sin entrar a definir conceptos básicos de programación: qué es una variable, una expresión o un condicional; ni conceptos del paradigma de orientación a objetos, concentrándonos en su uso con el lenguaje Delphi.

Sintaxis básica

Delphi se diferencia de la mayoría de los lenguajes de programación actualmente más usados, como Java, C++, C# o JavaScript, en que su sintaxis no está derivada de C sino de Pascal. Esto implica, entre otras cosas, que:

- No se diferencie entre mayúsculas y minúsculas en los identificadores (nombres de variables/métodos), palabras reservadas y, en general, ningún elemento del lenguaje.
- Que los bloques de código no se delimitan entre llaves sino mediante palabras clave del lenguaje: `begin` y `end`.
- Que en la declaración de variables el tipo no se antepone a los identificadores sino que se coloca al final separado por dos puntos.
- Que no es necesario introducir entre paréntesis las expresiones asociadas a muchas instrucciones.
- Que los operadores lógicos se representan por su nombre: `or`, `and`, `not`, en lugar de por símbolos como `||`, `&&` y `!`.

El estilo sintáctico puede agradarnos o no, es cuestión de gustos, pero en términos generales el código estilo Pascal resulta más fácil de leer que el código estilo C. Este último, sin embargo, es más compacto al sustituir muchas palabras clave por símbolos.

Comenzaremos analizando cuál es la estructura general de los archivos de código Delphi y después, una vez que tengamos una visión general de sus diferentes partes, iremos estudiando más detalladamente aspectos como los tipos de datos, estructuras de control, etc.

Módulos de código en Delphi

Al crear un nuevo archivo de código Delphi nos encontraremos fundamentalmente con una de dos estructuras posibles, según se trate de un programa o de un módulo (*unit*). El programa es el módulo de código por el que se iniciaría la ejecución de la aplicación y, por ello, solamente puede existir uno por proyecto. Los módulos Delphi, por el contrario, son contenedores de código: clases, estructuras de datos, funciones, etc., pudiendo utilizarse tantos como se necesiten.

En un proyecto Delphi típico, como el desarrollado en el capítulo previo, cada uno de los formularios está asociado un módulo de código independiente, cuyo nombre será el del formulario y la extensión .pas. El proyecto en sí tiene vinculado un archivo, con extensión .dpr, en el que se aloja el código de puesta en marcha del programa. Puedes acceder a dicho código mediante la opción View Source del menú contextual asociado al proyecto (en el Gestor de proyectos).

Estructura de un programa

Al abrir el archivo de código asociado al proyecto MicroClassic encontraremos lo siguiente:

```
program MicroClassic;

uses
  FMX.Forms,
  Principal in 'Principal.pas' {Form1} ,
  Visor in 'Visor.pas' {Form2};

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

Un módulo de código que va a actuar como punto de entrada a la aplicación siempre comienza con una cabecera conteniendo la palabra clave program, tras la cual se indica el nombre del programa (proyecto). El bloque de sentencias por las que se iniciará la ejecución se delimita con las palabras clave begin y end.

Entre la cabecera y el bloque de sentencias a ejecutar pueden existir varias secciones. La más habitual es la que se inicia con la palabra clave uses y que recoge las referencias a los módulos externos que van a utilizarse en el programa. Cuando en uno de los puntos del capítulo previo usábamos la opción File>Use Unit para poder usar un segundo formulario desde el primero, lo que hacíamos era pedirle a Delphi que agregase el nombre del módulo correspondiente a la cláusula uses. No habría ningún problema en añadirlo manualmente, escribiendo el nombre del módulo antes del punto y coma y separado del anterior por una coma.

Otra sección optativa que podría aparecer en un programa, antes del bloque de sentencia, es la de declaración de variables. Ésta se inicia con la palabra clave var, tras la cual se declaran las variables indicando su nombre y tipo. Cada declaración finaliza con un punto y coma.

Un aspecto importante: observa que en general las sentencias se finalizan con un punto y coma, pero que tras la palabra clave end, que señala el final del programa, hay que disponer un punto. Cuando se está acostumbrado a la sintaxis de C, C++ o Java es muy habitual colocar el punto y coma sistemáticamente, lo cual generaría un error de compilación en este caso.

Estructura de un módulo (*unit*)

Los módulos en Pascal (y por tanto en Delphi) están pensados para alojar código que será utilizado desde el módulo principal (programa) o bien desde otros módulos de código. Por ello su estructura es distinta a la descrita en el punto previo, encontrándose dividida en dos secciones bien diferenciadas: la interfaz y la implementación[28]. La cabecera, con el nombre del módulo, se inicia con la palabra clave unit en lugar de program, según puede apreciarse en el siguiente esqueleto de código.

```
unit Principal;

interface

uses módulos …

type
 Definición de tipos

var

 Declaración de variables

implementation

métodos y tipos internos

end.
```

28 A éstas pueden añadirse opcionalmente dos secciones más que contendrían código de inicialización y finalización del módulo, marcadas por las palabras clave initialization y finalization respectivamente.

En la sección `interface` se exponen los elementos del módulo que son accesibles desde el exterior, básicamente tipos de datos y variables. Si examinamos el módulo correspondiente al formulario principal del proyecto `MicroClassic`, por ejemplo, veremos que la definición del `TForm1` está en la sección de interfaz, así como la declaración de una variable llamada `Form1` de tipo `TForm1`. Esto permite usar el formulario desde otros módulos, accediendo a las propiedades y métodos con que cuenta.

La sección `implementation` puede contener definiciones de tipos, declaraciones de variables y, sobre todo, el código de implementación de los elementos definidos en la sección de interfaz. Nada de esto será visible desde el exterior una vez que el módulo se compile, a diferencia del contenido de la sección de interfaz del que podrá recuperarse información mediante un mecanismo conocido como RTTI (*Run-Time Type Information*).

Referencias a módulos

La cláusula `uses` suele aparecer siempre al principio de un módulo de código, seguida de una enumeración de referencias a módulos de los que se pretende utilizar tipos, variables o métodos definidos en su sección de interfaz.

En el proyecto `MicroClassic` hay una referencia al módulo `Visor` en la cláusula `uses` de la sección de implementación. Esto nos permite usar desde la ventana principal la variable `Form2` declarada en el segundo formulario, así como invocar al método `ShowModal` de la ventana.

Los componentes de la VCL y la FMX, así como todos los servicios básicos de la RTL, se encuentran distribuidos en un conjunto de módulos bastante grande. Por regla general no tendremos que agregarlos manualmente a la cláusula `uses` de nuestro módulo de código, ya que de ello se encarga el diseñador de Delphi.

No es extraño, dado el gran número de módulos de RTL, VCL y FMX, que haya algunos con una finalidad similar y, por ello, coincidan en nombre. Tanto en la RTL como en la FMX existe un módulo `Types` con definiciones de tipos comunes. El módulo `Controls`, con componentes básicos, está presente tanto en la FMX como en la VCL.

Para evitar la ambigüedad que genera esta duplicidad se usan referencias *cualificadas*, de forma que el nombre del módulo incluye como prefijo de ámbito el nombre de la biblioteca a que pertenece[29]: `System.Types`, `FMX.Types`, `Vcl.Controls`, `FMX.Controls`, etc.

Esta denominación compuesta se establece al crear un nuevo módulo de código, momento en que se fija su nombre al disponerlo tras la palabra clave `unit`. El nombre del archivo será el mismo pero con la extensión `.pas`.

Comentarios

Como muchos otros lenguajes de programación, Delphi contempla tanto comentarios hasta el final de línea como bloques de comentarios. La inclusión de comentarios en el código, siempre de manera adecuada, es una costumbre buena y recomendable.

Un comentario hasta final de línea, entendiendo como tal la aparición de un retorno de carro, se indica mediante los caracteres `//`. Éstos y cualquier contenido que les siga serán ignorados por el compilador. Un ejemplo:

```
...
begin
  if OpenDialog1.Execute then
  begin // Si se ha elegido un archivo
    expander := TExpander.Create(nil);
...
```

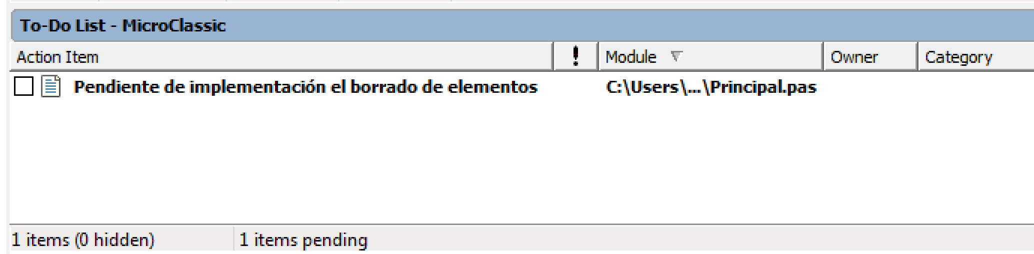
Los bloques de comentarios se delimitan con el carácter `{` al inicio y `}` al final. También es posible usar las parejas de caracteres `(*` y `*)` en lugar de los dos anteriores. Son el recurso habitual para, por ejemplo, documentar un método indicando parámetros, valor de retorno, precondiciones, etc.

Una clase de comentario a la que Delphi da un trato especial es aquél que comienza con la palabra `TODO` seguida de dos puntos y un texto. Por ejemplo:

```
procedure TForm1.MenuItem2Click(Sender: TObject);
begin

  { TODO: Pendiente de implementación el borrado de
elementos }
```

29 Para la RTL el prefijo es `System`, para la FMX es `FMX` y para la VCL es `Vcl`.

El texto del comentario aparecerá en una ventana, inicialmente oculta, que podemos abrir con la opción View>To-Do List. En la imagen inferior puede verse que hay dos columnas que hacen referencia al responsable de la tarea y la categoría, datos que pueden establecerse con las opciones `-o` y `-c` tras la palabra `TODO`.



Tipos de datos fundamentales

Una de las subsecciones que puede aparecer en un programa, en cualquiera de las secciones de un módulo (interfaz o implementación) y también en el interior de los métodos, es el apartado de declaración de variables. Éste se inicia con la palabra clave `var` e irá seguida de tantas declaraciones como se precise, ajustándose cada una a la siguientes sintaxis:

```
var
  variable1, ..., variableN: tipo;
```

Todas las variables que aparecen en la lista, separadas por comas, serán del tipo que se indique al final de la declaración. Como la mayoría de los lenguajes de programación Delphi cuenta con un conjunto de tipos intrínsecos para operar con números, caracteres, enumeraciones y cadenas de caracteres, así como tipos compuestos como las matrices (*arrays*), los conjuntos y los registros o estructuras.

Algunos de los tipos numéricos de Delphi cambian su tamaño, y por tanto el rango de valores que pueden albergar, según que la aplicación se compile para una plataforma de 32 ó 64 bits. Los caracteres y cadenas de caracteres pueden ser de tipo ASCII o UNICODE, siendo este último el formato usado por defecto. En los siguientes apartados se enumeran los tipos fundamentales, pero sin explicaciones detalladas sobre aspectos básicos como las diferencias entre tipos numéricos o de caracteres.

Números enteros y en coma flotante

Los tipos que permiten operar con números forman parte del conjunto de tipos simples de Delphi y se divide en dos grupos: tipos enteros y tipos reales. Una variable de tipo entero puede almacenar cualquier valor de un subconjunto de *Z*, mientras que una real contiene, usando almacenamiento en coma flotante, un valor de un subconjunto de *R*[30]. El tipo concreto que elijamos definirá la amplitud de ese subconjunto.

Al elegir un tipo entero debemos partir de si necesitamos o no almacenar el signo (si vamos a operar con números negativos o no) y cuál es el rango de valores que usaremos. Dependiendo de ello escogeremos entre la decena de tipos de la siguiente tabla.

Tamaño	Tipo sin signo	Tipo con signo
8 bits	`Byte`	`ShortInt`
16 bits	`Word`	`SmallInt`
32 bits	`LongWord`	`LongInt`
64 bits	`UInt64`	`Int64`
32 bits	`Cardinal`	`Integer`

Además de los anteriores, que mantienen su tamaño sin que importe la plataforma para la que se compile el proyecto, también podemos usar los tipos `NativeInt` y `NativeUInt` si queremos que el tamaño sea de 32 ó 64 bits dependiendo de la plataforma.

En cuanto a los tipos reales tenemos básicamente tres opciones: `Single`, `Double` y `Extended`, según precisemos precisión simple, doble o extendida. La ocupación es de 4, 8 ó 10 bytes, respectivamente, siempre con signo. Debe tenerse en cuenta que el tipo `Extended` tiene menos precisión cuando se compila para Win64 frente a Win32, concretamente el tamaño de 10 bytes en Win32 se reduce a 8 en Win64, por lo que si el proyecto a desarrollar es multiplataforma es recomendable no depender de este tipo de dato.

---

30 Hay que interpretar *Z* y *R* en un contexto matemático: *Z* representa el conjunto de los números enteros y *R* el de los números reales.

Por último tenemos el tipo `Currency` de 64 bits y coma fija, por lo que podría decirse que está a medio camino entre los números enteros y los de coma flotante. Es el tipo a usar cuando se necesita trabajar con números con parte decimal pero sin los problemas de redondeo de `Single`, `Double` o `Extended`.

Caracteres y cadenas de caracteres

Delphi cuenta con tres tipos diferentes para trabajar con caracteres y otros tantos para las cadenas de caracteres. Dos de ellos son, en realidad, un alias, por lo que tendríamos que hablar realmente de dos tipos. La diferencia entre éstos estriba en el método de almacenamiento de los caracteres: ASCII o UNICODE.

Los tipos `AnsiChar` y `WideChar` permiten almacenar un carácter ASCII (8 bits) o UNICODE (16 bits[31]), respectivamente. El tipo `Char` es sinónimo de `WideChar`, haciendo de UNICODE la opción por defecto en Delphi. De esta forma no estamos limitados al reducido conjunto de caracteres occidental y es posible usar caracteres de muchos otros alfabetos.

Para el trabajo con cadenas de caracteres[32] disponemos de los tipos `AnsiString`, `UnicodeString` y `String`, siendo este último un alias de `UnicodeString`. Como es fácil imaginar el primero trabaja con cadenas de caracteres ASCII, mientras que el segundo lo hace con cadenas de caracteres UNICODE. También se mantiene por compatibilidad con versiones previas el tipo `WideString`, empleado para cadenas de caracteres UNICODE en un formato específico para el modelo de componentes COM de Windows.

Delphi XE2 ofrece en el módulo `System.Character` un conjunto de funciones, así como una clase llamada `TCharacter`, que permiten comprobar la categoría de un carácter: puntuación, dígito numérico, letra minúscula/mayúscula, etc.; obtener el código numérico que corresponde a un carácter, convertir a mayúscula o minúscula, etc.

---

31 En realidad la longitud de carácter UNICODE es variable, pudiendo llegar a los 32 bits, pero en Delphi XE2 se usan 16 bits por carácter.

32 Pascal, y por tanto el lenguaje Delphi, almacena las cadenas de caracteres con un prefijo en el que se indique la longitud real, lo cual le diferencia de los lenguajes al estilo de C que recurren a un carácter nulo al final y no hay indicación explícita de la longitud.

Otros tipos de datos básicos

Aparte de los valores numéricos, los caracteres y las cadenas de caracteres, también pueden considerarse datos de tipos básicos los booleanos y los punteros.

El tipo Boolean puede contener uno de dos valores predefinidos: True y False, pudiendo ser por tanto el destino de cualquier expresión de tipo booleano y participar en expresiones lógicas. Aunque también existen los tipos ByteBool, WordBool y LongBool raramente tendremos que usarlos.

En cuanto a los punteros, su tamaño está en consonancia con la plataforma para la que se compila el proyecto. En Win64 las direcciones son de 64 bits y por lo tanto los punteros también. Para Win32 y MacOS X el tamaño es de 32 bits. Los punteros son direcciones a datos que pueden ser virtualmente de cualquier tipo: números, caracteres, registros, etc. La declaración de un puntero se efectúa anteponiendo al tipo base el carácter ^. Por ejemplo:

```
  añoBisiesto: Boolean;
  pBisiesto: ^Boolean;
begin
  pBisiesto := @añoBisiesto;
  pBisiesto^ := True;
```

Aquí pBisiesto es un puntero a un dato Boolean. Con el operador @ se obtiene la dirección de la variable añoBisiesto y se almacena en el puntero. Mediante el operador ^ se accede al contenido de la dirección indicada por el puntero, por ejemplo para modificarlo como se ha hecho en la última sentencia.

Notación para identificadores y literales

Para declarar variables además de conocer los tipos de datos existentes también tendremos que saber cuáles son las reglas a seguir a la hora de crear identificadores. Asimismo la introducción de valores literales conlleva el uso de una cierta notación propia de cada lenguaje de programación.

En Delphi los identificadores definidos por el programador, para dar nombre a variables, métodos, tipos propios, etc., pueden tener cualquier longitud, si bien el compilador solamente utiliza los primeros 255. En la práctica esto es más que suficiente para que no haya problemas con la aparición de identificadores duplicados.

El primer carácter de un identificador ha de ser un carácter alfabético o el carácter _. A continuación pueden emplearse letras (mayúsculas o minúsculas, ya que el lenguaje no diferencia), números y el propio carácter _. Aunque no están permitidos los símbolos ni espacios en blanco, sí se contempla el uso de cualquier carácter alfabético UNICODE, lo cual implica que pueden crearse identificadores en los que aparezca la eñe o letras acentuadas sin ningún problema.

Una regla básica, aplicable prácticamente a todos los lenguajes de programación, es que no deben utilizarse como identificadores palabras reservadas del lenguaje como son las instrucciones, operadores lógicos, etc.

En cuanto a la notación a emplear para la introducción de valores literales, las reglas generales son las siguientes:

- Los valores numéricos se introducen como tales: sucesiones de dígitos entre los que, opcionalmente, puede aparecer un punto para separar la parte entera de la decimal.
- Para números muy grandes o muy pequeños puede recurrirse a la notación exponencial, colocando tras los dígitos numéricos que forman el multiplicador el carácter `E` seguido del exponente al que se elevará la base `10`. Por ejemplo: `45E6` equivaldría a `45000000`.
- Es posible facilitar valores numéricos en base hexadecimal usando el prefijo `$`. Por ejemplo: `$B800`.
- Los caracteres hay que delimitarlos entre comillas simples. Por ejemplo: `'b'`. También se pueden introducirse a través del código asociado, si es que se conoce, mediante el prefijo `#`. Por ejemplo: `#98` sería equivalente a `'b'`.
- Las cadenas de caracteres se delimitan entre comillas simples. Aunque no es usual, también pueden escribirse como secuencias de códigos de carácter[33]. Por ejemplo: `#66#97#114#99#111` equivaldría a `'Barco'`.

Los valores literales se usan para dar un contenido a las variables, pero también pueden emplearse como parámetros a facilitar a los métodos o como constantes en la evaluación de expresiones.

---

33 Esta técnica es útil especialmente cuando es preciso introducir en una cadena caracteres no imprimibles, como pueden ser tabuladores, avances de líneas y similares.

Enumeraciones y subrangos

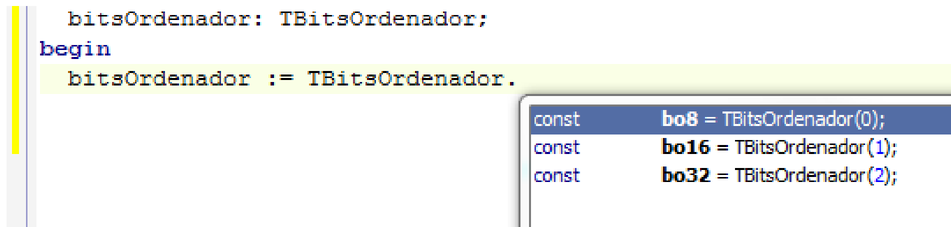
Durante el desarrollo del proyecto `MicroClassic` hemos tratado con propiedades que, como `Align` o `BorderStyle`, se caracterizan por tomar uno entre una enumeración de posible valores. En el caso de `Align` su tipo es `TAlignLayout`, definido en `FMX.Types` como se muestra a continuación:

```
type
  TAlignLayout = (alNone, alTop, alLeft, alRight,
      alBottom, alMostTop, alMostBottom, alMostLeft,
      alMostRight, alClient, alContents, alCenter,
      alVertCenter, alHorzCenter, alHorizontal,
      alVertical, alScale, alFit, alFitLeft, alFitRight);
```

Nosotros podemos definir nuestros propios tipos enumerados de manera similar, usándolos para limitar los valores que pueden tomar las variables declaradas como de ese tipo. La definición se realiza en la sección `type`, previamente al uso del nuevo tipo en la sección `var`. En nuestro caso podríamos definir un tipo como el siguiente para elegir los bits de cada ordenador:

```
type
  TBitsOrdenador = (bo8, bo16, bo32);
```

Al asignar valor a una variable de este tipo veríamos, como en la figura inferior, la lista de los valores válidos.



Internamente a cada identificador de la enumeración se le asigna un valor numérico, como puede apreciarse en la lista de la imagen anterior. Por defecto se inicia con el valor `0` y para los elementos posteriores se va incrementando en una unidad. En caso de que nos interese asociar un valor concreto a cada identificador podemos hacerlo con la siguiente sintaxis:

```
type
  TBitsOrdenador = (bo8 = 8, bo16 = 16, bo32 = 32);
```

De esta forma cada identificador de la enumeración se podría evaluar numéricamente como la cantidad de bits que tiene la máquina, en caso de que haya necesidad de realizar un tratamiento numérico de dicho valor.

Asimismo contamos con propiedades y variables en nuestro proyecto que solamente permiten un subrango de valores de un tipo básico. Uno de ellos es el año de fabricación del ordenador, limitado mediante las propiedades `Min` y `Max` del componente `TNumberBox` al intervalo `1965-1995`. Para almacenar este dato en lugar de recurrir a un tipo como `Integer`, con un rango de valores posibles amplísimo, podríamos definir un tipo subrango y usarlo como puede verse en el siguiente fragmento de código:

```
type
  TAñoFabricacion = 1965..1995;

var
  fabricado: TAñoFabricacion;

begin
  fabricado := 1950; // Esta asignación daría un error
```

El subrango se define colocando los valores extremos, mínimo y máximo, separados por dos puntos. Los extremos en este caso son valores numéricos, pero también podrían ser caracteres o incluso identificadores de un enumeración. Como se apunta en la última sentencia, la asignación de un valor fuera del rango indicado en el tipo provocaría un error de compilación

Matrices

Los tipos descritos en los puntos previos son tipos simples, en el sentido de que al declarar una variable de cualquiera de ellos ésta puede contener un único valor. En contraposición los tipos compuestos o estructurados se forman a partir de los anteriores y tienen la capacidad de almacenar múltiples valores, ya sea del mismo tipo fundamental o de tipos diferentes según los casos.

El tipo compuesto más habitual es el *array* o matriz[34]: una colección de valores que pertenecen al mismo tipo fundamental y a los que se accede individualmente mediante uno o más índices.

---

34 Los términos *array* y *matriz* son denominaciones genéricas, existiendo otras más específicas como *vector* para referirse a las matrices de una sola dimensión (estructura lineal), tabla cuando se hace referencia a matrices bidimensionales, etc.

El número de dimensiones y de elementos por dimensión de una matriz pueden establecerse en el momento de la declaración de la variable, hablándose en este caso de matrices estáticas, o bien con posterioridad a la declaración, caso en el que se denominan matrices dinámicas. Por tanto estas últimas son las que usaremos siempre que no conozcamos de antemano el número exacto de datos que precisamos almacenar.

La declaración de una matriz estática se realiza disponiendo delante del tipo base la palabra `array` y a continuación, entre corchetes, tantos subrangos como dimensiones se precisen estableciendo los límites de cada dimensión. Supongamos que queremos almacenar para cada ordenador las distintas resoluciones gráficas que ofrece, asumiendo que como máximo serán diez. Podríamos hacerlo al menos de dos formas diferentes: usando una matriz de cadenas de caracteres, de forma que cada cadena contuviese los datos de una resolución, o bien recurrir a una matriz bidimensional en la que por cada resolución se contemplase el número de píxeles en horizontal, en vertical y el número de colores. La declaración de las variables y el acceso a su contenido se realizarían así:

```
var
  resolucion1: array[1..10] of String;
  resolucion2: array[1..10,1..3] of Integer;

begin
  resolucion1[1] := '256x192x16';

  resolucion2[1][1] := 256;
  resolucion2[1,2] := 192;
  resolucion2[1,3] := 16;
```

Como puede apreciarse en las últimas tres sentencias, cuando se accede a un elemento de una matriz con varias dimensiones los índices pueden colocarse entre parejas de corchetes independientes o bien separados por comas en un solo par de corchetes. La primera matriz tendría diez elementos, con índices del 1 al 10, y cada uno de ellos sería una cadena. La segunda formaría una tabla con 30 enteros, estructurados en 10 filas de 3 columnas o viceversa según interpretemos los índices.

Una pecualiaridad del lenguaje es que los subrangos que determinan el tamaño no tienen necesariamente que ser de números enteros, pudiendo también usarse caracteres o valores de una enumeración. Esos mismos tipos serían empleados como índices para acceder a los elementos de la matriz. Cada dimensión puede usar índices de un tipo distinto en caso de que nos interese. Un ejemplo podría ser el siguiente:

```
type
  TMes = (Ene, Feb, Mar, Abr, May, Jun,
          Jul, Ago, Sep, Oct, Nov, Dic);

  TContenido = (Análisis, Listado, Práctico);
  TAñoPub = 1970..1995;
...

var
  revista: array[TAñoPub, TMes, 1..500] of TContenido;
...

begin
  revista[1982, Ene, 10] := Listado;
```

La matriz `revista` tiene tres dimensiones. La primera usa el subrango `TAñoPub` y por tanto tendrá cómo índice inferior `1979` y como superior `1995`, lo que hace un total de 26 elementos. La segunda dimensión usa como índice la enumeración `TMes`, contando con 12 elementos por cada año. Por último la tercera dimensión, que hace referencia a la página de la revista, tiene 500 elementos.

El tipo `TContenido` se ha definido para clasificar los contenidos de las revistas en tres hipotéticas categorías. En la última sentencia se muestra cómo se indicaría que el número de enero de 1982 de una revista, en su página 10, contiene un listado.

Cuando el número de elementos necesarios sea un dato desconocido a priori, determinándose durante la ejecución, recurriremos a las matrices dinámicas. Éstas tienen algunas limitaciones en comparación con las estáticas, a saber: los índices siempre serán números enteros y el límite inferior es el cero. El primer paso es declarar la matriz, empleando la palabra clave `array` pero sin disponer detrás los corchetes. Posteriormente se indican los elementos mediante el procedimiento `SetLength`, tal y como se muestra en el código siguiente:

```
var
  resolución: array of String;
...
begin
  ...
  SetLength(resolución, 7);
  resolución[0] := '256x192x16';
```

En este caso la variable `resolución` tendrá 7 elementos, con índices comprendidos entre 0 y 6. También es posible trabajar con matrices dinámicas de más de una dimensión, por ejemplo:

```
var
  numFascículos, numPáginas: Integer;
  coleccionable: array of array of TContenido;
begin
  numFascículos := 48;
  numPáginas := 64;

  SetLength(coleccionable, numFascículos, numPáginas);

  coleccionable[0, 0] := Análisis;
```

En la declaración se recurre a la recurrencia `array of array` repitiéndola tantas veces como dimensiones se precisen. Al invocar al procedimiento `SetLength` se han de facilitar tantos enteros como dimensiones tenga la matriz, indicando cada uno de ellos los elementos que tendrán.

Indistintamente de que la matriz sea estática o dinámica, siempre podemos recurrir a la función `Length` para saber su tamaño, la función `Low` para conocer el índice inferior y la función `High` para el índice superior.

Registros

En una matriz todos los elementos tienen el mismo tipo, siendo básicamente iguales y diferenciándose únicamente por su contenido y el índice que tienen asociado. En contraposición, un registro es un tipo de dato complejo que se compone de elementos diferenciados por su nombre (no un simple índice) y por su tipo que no ha de ser necesariamente el mismo.

Antes de declarar una variable de tipo registro es necesario definir la estructura de éste, estableciendo el nombre y tipo de cada uno de sus miembros. La sintaxis es simple y en el ejemplo siguiente se pone en práctica definiendo un registro para almacenar los datos básicos de un ordenador para el proyecto `MicroClassic`.

```
type
  TOrdenador = record
    nombre     : String;
    año        : TAñoPub;
    fabricante : String;
    modelo     : String;
    bits       : TBitsOrdenador;
    RAM        : SmallInt;
    ROM        : SmallInt;
    funciona   : Boolean;
  end;
```

La definición del registro es similar a la declaración de una lista de variables, pero no conlleva reserva de espacio alguna para almacenamiento. Esa reserva se lleva a cabo cada vez que declaremos una variable de tipo `TOrdenador` usando la sintaxis habitual.

El acceso a los elementos de un registro no se hace mediante un índice, como en el caso de las matrices, sino usando el nombre que se asignó a cada miembro. Para ello se recurre a la sintaxis `variable.miembro`, tal y como se aprecia en la imagen inferior.

```
  unOrdenador: TOrdenador;

begin
  unOrdenador.
      var    nombre: string;
      var    año: TAñoPub;
      var    fabricante: string;
      var    modelo: string;
      var    bits: TBitsOrdenador;
      var    RAM: SmallInt;
      var    ROM: SmallInt;
      var    funciona: Boolean;
```

La variable `unOrdenador` es de tipo `TOrdenador` y, por tanto, cuenta con los miembros definidos en dicho tipo, lo cual queda patente en la lista de ayuda que ofrece el editor de Delphi. Esto nos permitiría guardar los datos de un ordenador, pero el supuesto cliente es coleccionista y en consecuencia tendrá un buen número de ellos. Para almacenar todos sus datos podríamos emplear una matriz de elementos `TOrdenador`:

```
var
  colección: array of TOrdenador;
  numOrdenadores: Integer;
begin
  // Determinar el número de ordenadores en la colección
  numOrdenadores := 75;
  SetLength(colección, numOrdenadores);

  colección[0].nombre := 'Atari 400';
  colección[0].año := 1979;
...
```

El valor de `numOrdenadores` habría que establecerlo a priori y podría incrementarse posteriormente en caso de que fuese preciso.

Al trabajar con variables que son registros es muy habitual encontrarse con el tedio de tener que introducir reiteradamente el nombre de la variable para poder acceder a sus miembros, especialmente si éstos son muchos. La sentencia `with` sirve para abreviar dichas referencias como puede apreciarse en el siguiente fragmento de código:

```
with colección[0] do begin
  nombre := 'Atari 400';
  año := 1979;
  ...
end;
```

NOTA

Los miembros de un registro pueden ser de cualquier tipo, incluyendo matrices y otros registros. Cuanto más complejos sean los tipos de los miembros tanto más lo serán las referencias a usar para acceder a su contenido. En Delphi incluso se contempla la posibilidad de que un registro tenga propiedades y métodos, de manera análoga a una clase.

Conjuntos

Aparte de matrices y registros/estructuras, tipos presentes en la mayoría de los lenguajes de programación, en Delphi también contamos con conjuntos. Una variable de este tipo representa un conjunto en su acepción matemática: contendrá elementos de un cierto tipo base, pudiendo estar vacío y contemplándose operaciones como la comprobación de pertenencia de un elemento, unión, diferencia e intersección de conjuntos, etc.

El tipo base de un conjunto normalmente es una enumeración o un subrango, no pudiendo en cualquier caso superar los 256 elementos. En el proyecto `MicroClassic` un conjunto sería útil para determinar las conexiones de entrada/salida de cada ordenador: ¿tiene conexión para casete? ¿cuenta con salida de TV? ¿dispone de conector para joystick?, etc.

Para declarar una variable de tipo conjunto se usa la sintaxis `set of tipo`, pudiendo ser el tipo un subrango, una enumeración o bien el nombre de un tipo definido previamente. En el siguiente fragmento de código se muestran varias de las alternativas:

```
type
  TConexiones = (Casete, Disco, TV, RF, Joystick);
var
  conectores: set of TConexiones;
  gráficos: set of (sprites, scroll, fuentes);
  salidaTV: Boolean;

begin
  conectores := [Casete, Joystick];
  conectores := conectores + [TV];

  salidaTV := TV in conectores;
```

La variable `conectores` puede estar vacía, contener uno o más valores de la enumeración `TConexiones`. Lo mismo ocurre con la variable `gráficos`, si bien en este caso la enumeración se ha indicado directamente en la declaración sin recurrir a la definición de un tipo.

En las sentencias de asignación puede apreciarse cómo se almacena inicialmente un conjunto[35] con dos elementos en `conectores`, añadiéndose un elemento adicional después mediante el operador `+`. Con el operador `in` se comprueba la pertenencia de un elemento al conjunto, por lo que `salidaTV` tomará el valor `True` ya que `conectores` contiene el elemento `TV`.

Expresiones

Conociendo los tipos de datos básicos de Delphi ya podemos declarar las variables que precisemos para almacenar la información de nuestras aplicaciones, datos que, por regla general, serán usados en algún momento en expresiones que los evaluarán o transformarán. En los sencillos ejemplos de los puntos previos, así como en el código que escribimos en el capítulo anterior asociado a los eventos de algunos componentes, la mayoría de las expresiones empleadas han sido de asignación, disponiendo a la izquierda del operador `:=` la variable o propiedad receptora y a la derecha la expresión a asignar: una constante, variable, expresión aritmética, relacional o lógica, resultado de invocar a un método, etc.

---

35 Las literales de tipo conjunto van delimitadas entre corchetes, pudiendo éstos estar vacíos (conjunto vacío) o bien contener uno o más elementos separados por comas.

No todos los operadores son aplicables a todos los tipos de datos y, en consecuencia, el tipo de una variable determinará la posibilidad de usarla o no en una cierta expresión. También debe tenerse en cuenta que un mismo operador puede implicar una operación diferente según el tipo de dato al que se aplique

Delphi dispone de los operadores aritméticos, relacionales y lógicos habituales en todos los lenguajes de programación, así como otros para tipos de datos específicos como son los conjuntos, las cadenas de caracteres o los punteros. Todos ellos se encuentran descritos en la documentación electrónica que acompaña al producto, por lo que en los siguientes apartados se hará solamente una enumeración somera de los mismos.

Aritméticas, relacionales y lógicas

Además de la expresión de asignación, que es probablemente la más usada, la mayor parte del resto de expresiones se engloban en una de las categorías siguientes:

- **Aritméticas**: Los operandos son de alguno de los tipos numéricos y además de los cuatro operadores básicos también existen los operadores `div` y `mod` para obtener el cociente y resto, respectivamente, de una división entera.
- **Relacionales**: Se construyen con operadores que evalúan la relación entre dos operandos, dispuestos a izquierda y derecha, para determinar si son iguales, distintos, el primero mayor que el segundo, etc. Para comprobar la desigualdad el operador es <>. El resto de operadores son los habituales: =, <, >, <= y >=.
- **Lógicas**: Estas expresiones también se denominan booleanas ya que sus operandos son siempre de ese tipo. Aparte de los tres operadores clásicos: `not`, `and` y `or`, en Delphi también existe el operador `xor` (`or` exclusivo).

Sobre operadores numéricos enteros pueden aplicarse también los operadores de manipulación de bits, cuyo nombre coincide con los cuatro operadores lógicos anteriores pero evaluando el resultado bit a bit. A ellos hay que agregar los operadores `shl` y `shr` que desplazan los bits del valor entero hacia la izquierda o derecha, respectivamente, tantas posiciones como se indique con el operando dispuesto a la derecha.

Conjuntos

Las expresiones en las que se usan conjuntos como operandos emplean operadores que ya conocemos, como los aritméticos y relacionales, pero la operación realizada es específica para este tipo de dato. En un ejemplo previo vimos, por ejemplo, cómo realizar la unión entre dos conjuntos mediante el operador +. De manera análoga el operador – sirve para hallar la diferencia entre dos conjuntos y el operador * para realizar la intersección.

Con los operadores relacionales podemos comprobar la igualdad y desigualdad, como haríamos con cualquier otro tipo de dato, y también el hecho de que un conjunto sea subconjunto de otro, operador <=, o superconjunto, operador >=. Todos éstos actúan siempre sobre dos conjuntos. El único operador que actúa sobre un elemento y un conjunto es in, con el objetivo de comprobar si el primero pertenece al segundo.

Punteros

Al tratar este tipo de dato conocimos dos operadores asociados: @ y ^. El primero sirve para obtener la dirección de memoria donde está almacenado el contenido de una variable, pudiendo así guardarla en un puntero. El segundo se emplea cuando al usar un puntero lo que se quiere es acceder al contenido al que hace referencia.

Los punteros pueden compararse entre sí para saber si contienen o no la misma dirección, mediante los operadores de igualdad (=) y desigualdad (<>). Asimismo el operador + permite aplicar un desplazamiento sobre la dirección almacenada en el puntero, lo cual es útil generalmente para acceder a elementos concretos de una estructura más o menos compleja.

Otros tipos de expresiones

Además de las ya citadas también podemos encontrarnos con expresiones en las que participan cadenas. En estos casos el operador + sirve para concatenar dos cadenas, dando una como resultado, y los operadores relacionales facilitan la comparación entre cadenas: igualdad, desigualdad, menor que o mayor qué desde un punto de vista lexicográfico.

Son expresiones las invocaciones a métodos, en las que cada parámetro facilitado puede a su vez ser otra expresión siempre y cuando el resultado genere un dato del tipo adecuado; también lo es el acceso a un elemento de una matriz mediante el operador `[]` y un índice que puede obtenerse como resultado de la evaluación de otra expresión.

Cuando el tipo de un dato no se ajusta a nuestras necesidades podemos usar una expresión de moldeado (conversión) de tipo o *type-casting*. La sintaxis de esta expresión es la siguiente:

```
NuevoTipo(dato)
```

El tipo del dato y el nuevo tipo han de ser compatibles[36] para que la conversión pueda realizarse. Lo son los punteros entre sí, los distintos tipos numéricos enteros entre sí o los tipos enteros y los caracteres.

Sentencias

Los tipos de datos y los operadores usados para formar expresiones con los datos son dos de los pilares de Delphi. El tercero lo forman las sentencias con que cuenta el lenguaje, mediante las que podremos controlar el flujo de ejecución del programa, realizar llamadas a métodos, etc.

Las sentencias con que cuenta Delphi son muy similares a las que podemos encontrar en cualquier lenguaje de programación imperativo, si bien su sintaxis puede diferir en aspectos puntuales. El clásico condicional `if`, por ejemplo, usa la palabra clave `then` para dar paso a las sentencias a ejecutar en caso de que la evaluación sea cierta, cuando en la mayoría de los lenguajes dicha palabra se da por asumida y por tanto se omite.

El tipo de sentencia más simple posiblemente sea la asignación, una sentencia que usábamos repetidamente en el capítulo previo para guardar ciertos datos y recuperarlos de las propiedades de los componentes. Una expresión por sí sola no es una sentencia, ya que el programa ha de saber qué quiere hacerse con el resultado de su evaluación. La asignación de ese resultado a una variable o propiedad sí es una sentencia ejecutable.

---

36 Para realizar conversiones entre tipos no compatibles, como puede ser un número en punto flotante y un tipo entero, hay que recurrir a funciones de Delphi que realizan el trabajo de adaptación, como pueden ser `Trunc` o `Round`.

Condicionales

Las sentencias que ejecutan o no una sentencia dependiendo de un valor booleano son dos: `if-then-else` y `case`. El valor puede ser un literal, una variable, una expresión relacional o lógica, el resultado obtenido por la invocación a una función, etc. Lo que importa al final es el tipo del valor, algo que se aplica a todas las sentencias.

La sintaxis general de la sentencia `if-then-else` es la siguiente:

```
if valor-booleano then
  sentencia-si-true
else
  sentencia-si-false;
```

El apartado `else` y la sentencia asociada son opcionales. En caso de que se necesite ejecutar más de una sentencia habrá que crear una sentencia compuesta: un bloque de sentencias delimitados con las palabras clave `begin` y `end`. Esto también es aplicable a otras sentencias de Delphi que veremos después como `case`, `while` o `repeat`.

> ADVERTENCIA
>
> En Delphi solamente se marca con punto y coma el final de las sentencias. En el caso de `if-then-else` la sentencia completa concluye con la sentencia que hay tras `else`, no con la que hay tras `then`, y por ello no se usa punto y coma detrás de ésta. Es un error habitual si se está acostumbrado a C/C++, Java o un lenguaje similar.

En cuanto a la sentencia `case`, su sintaxis es la mostrada a continuación:

```
case valReferencia of
  val1: sentencia1;
  val2: sentencia2;
  ...
end
```

El valor de referencia dispuesto tras `case` se compara con la enumeración de valores[37] que hay a continuación, ejecutándose la sentencia que corresponda. Opcionalmente puede agregarse un último apartado `else` con la sentencia a ejecutar si no hay coincidencia con ninguno de los valores indicados.

---

37 Estos valores no han de ser necesariamente un entero o un carácter, también pueden utilizarse subrangos y listas de valores.

Iterativas

Este tipo de sentencias sirven para repetir la ejecución de una o más sentencias de acuerdo a unas condiciones que pueden ser evaluadas al inicio o al final de cada ciclo. Dependiendo de nuestras necesidades podemos emplear las instrucciones `while`, `repeat` o `for`.

Una sentencia `while` evalúa la expresión facilitada al inicio de cada ciclo, procediendo a ejecutar la sentencia escrita tras la palabra clave `do` en caso de que el resultado sea `true`. Tras esto se vuelve al principio y se repite la evaluación, interrumpiéndose la iteración cuando se obtenga el valor `false`. La sintaxis es la siguiente:

```
while valor-booleano do
  sentencia;
```

En contraposición la sentencia `repeat` primero ejecuta las sentencias y después evalúa la expresión booleana dispuesta al final, tras la palabra clave `until`. Esto implica que esas sentencias se ejecutarán al menos una vez. Las palabras `repeat` y `until` forman por sí mismas una sentencia compuesta, por lo que no es necesario crear un bloque con `begin-end`. La sintaxis:

```
repeat
  sentencia;
  ...
until valor-booleano;
```

A diferencia de las dos anteriores, la sentencia `for` rige el número de iteraciones no evaluando una condición arbitraria sino comprobando si una variable de tipo ordinal (entero, carácter, enumeración) ha alcanzado un cierto límite. Éste puede venir indicado explícitamente, a través de un valor literal, variable o una expresión, o bien ser inferido a partir del número de elementos contenidos en una colección.

La sintaxis de la primera modalidad de esta sentencia es la mostrada a continuación:

```
for variable := inicial {to|downto} límite do
  sentencia;
```

Si el valor inicial es menor que el final ambos se separarán con la palabra clave `to`, en caso contrario se usará `downto`. En cualquier caso la sentencia que sigue a `do` se ejecutará una vez para cada valor de ese intervalo, incluyendo los valores límite. Como en los demás casos la sentencia a ejecutar puede ser compuesta, es decir, un bloque de sentencias delimitado por las palabras `begin` y `end`.

Cuando se usa la sentencia `for` para enumerar los elementos de una colección, como puede ser una matriz, un conjunto, una cadena de caracteres o alguna colección especializada, la sintaxis es más simple:

```
for variable in colección do
  sentencia;
```

En este caso la sentencia se ejecutará tantas veces como elementos contenga `colección`. En cada ciclo `variable`, que ha de ser del tipo base de `colección`, tomará uno de los elementos.

En el interior de cualquier tipo de construcción iterativa puede utilizarse la instrucción `break` para provocar su interrupción, desviando la ejecución a la primera sentencia tras el bucle, y la instrucción `continue` para iniciar un nuevo ciclo.

Procedimientos/Funciones

Las sentencias pueden disponerse en la sección principal de un programa, el bloque `begin` y `end` de un módulo `program`, pero en la mayoría de las ocasiones tomarán la forma de métodos: procedimientos y funciones alojados en la sección de implementación de un módulo Delphi.

Los procedimientos únicamente se diferencian de las funciones en que éstas devuelven un valor como resultado y los primeros no. En ambos casos se trata de bloques de código a los que se asigna un nombre[38] y que se componen de declaraciones y sentencias. La sintaxis general es la siguiente:

```
procedure nombre[(parámetros)];
[var  declaración de variables; ]
begin
  sentencia;
  ...
end;

function nombre[(parámetros)]: tipoRetorno;
[var   declaración de variables; ]
begin
  sentencia;
  ...
  Result := valor; // Devolución del resultado
end;
```

38 También existe la posibilidad de usar métodos anónimos, bloques de código que no tienen asociado un nombre.

El nombre de un procedimiento (o función) debe ajustarse a la notación que se describió anteriormente y, en cualquier caso, no entrar en conflicto con otros identificadores. No obstante un procedimiento puede coincidir en nombre con otro que estén en el mismo ámbito siempre y cuando sus listas de parámetros difieran. Es lo que se conoce como *sobrecarga* de métodos, una característica útil para mantener un mismo nombre para una cierta operación que puede aplicarse a distintos parámetros.

Las variables declaradas en el interior de un procedimiento son locales, lo que significa que se les asigna memoria y se inicializan automáticamente cuando la ejecución entra en el procedimiento y son destruidas al finalizar. Por tanto no es posible acceder a ellas desde el exterior del ámbito del procedimiento.

Cuando lo que se define es una función debe indicarse en la cabecera, tras el nombre y la lista de parámetros si existe, el tipo del valor que devolverá. Ese valor se asignará a la seudo-variable `Result` antes de salir de la función. Obviamente dicho valor puede ser resultado de cualquier expresión o haberse obtenido de recursos externos.

Tanto procedimientos como funciones pueden tomar opcionalmente una lista de parámetros, en cuyo caso habrá que indicar tras el nombre del procedimiento, y entre paréntesis, el nombre local con el que se conocerán y su tipo, separando unos de otros mediante punto y coma. Por defecto los parámetros se reciben por valor: cuando se invoca al procedimiento lo que se envía es una copia del valor, por lo que no es posible modificar el original. En caso de que nos interese poder cambiar el contenido original de un parámetro habrá que designar éste como variable, precediéndolo con la palabra clave `var` tal y como se ve en el siguiente ejemplo:

```
procedure hipo(C1, C2: Real; var H: Real);
begin
  H := H + C1*C1 + C2*C2;
end;
```

Al invocar a este procedimiento el tercer parámetro deberá ser una variable, los otros dos pueden ser constantes, variables o resultados de la evaluación de una expresión. Lo que se hace es acumular en dicha variable, sumando a su contenido actual, el resultado de un cálculo sobre los otros dos. Podríamos usar este procedimiento así:

```
var
  H: Real;

begin
```

```
  H := 0.0;
  hipo(3.2,4.1,H);
  hipo(1.8,5.7,H);
 // En H tenemos la suma de las hipotenusas
```

Los parámetros de entrada pueden tener asociado un valor por defecto que sería el que tomarían en caso de que en la invocación no se facilitase explícitamente. Dicho valor se indicará tras el tipo del parámetro y el símbolo = como se aprecia en el siguiente ejemplo:

```
procedure nuevoOrdenador(nombre: String; bits: Byte = 8);
...
```

Al invocar a este método si no se indica el número de bits del ordenador se asumirá que son 8, que es el valor por defecto facilitado en la cabecera del procedimiento.

En caso de que un parámetro de entrada sea una matriz, en la declaración se indicará el hecho mediante la palabra clave `array` pero sin especificar el número de elementos. En el interior del procedimiento puede recurrirse a las funciones `Length`, `High` y `Low` para conocer el número de elementos y los límites de los índices. También puede enumerarse la matriz sin necesidad de esta información mediante una sentencia `for`, tal y como se muestra a continuación:

```
procedure setResoluciones(resoluciones: array of String);
var
  unaR: String;
begin
  for unaR in resoluciones do
    // Tratamiento del elmento unaR
end;
...
  setResoluciones(['12x15','15x32']);
```

Observa cómo al invocar a `setResoluciones` se ha facilitado una matriz como valor literal. Igualmente se podría haber declarado una variable y usarla después, tanto si la matriz es estática como dinámica.

> NOTA
>
> En el interior de un procedimiento aparte de declarar variables también se permite la definición de tipos que serían locales. Incluso es posible definir otros procedimientos dentro, lo que en Delphi se llaman rutinas anidadas, a los que no podría invocarse desde el exterior del método contenedor.

Control de excepciones

La ejecución de determinadas sentencias, válidas sintácticamente y que por tanto no producen errores de compilación, puede provocar un fallo que, de no ser controlado, interrumpa el funcionamiento de una aplicación. Estos fallos de ejecución se denomina genéricamente excepciones. Ejemplos de sentencias que pueden producirlas son los cálculos aritméticos, por ejemplo al intentar dividir por cero; el acceso a elementos de una matriz, cuando el índice está fuera de rango, o las conversiones inválidas de tipo.

Delphi cuenta con sentencias que posibilitan el tratamiento controlado de las excepciones, partiendo de que sabemos qué sentencias pueden generarlas y cómo proceder en caso de que ocurran. Se trata de la instrucción `try/except`. En el siguiente ejemplo, que controla un posible error de acceso a un elemento inexistente de una matriz, puede apreciarse su sintaxis:

```
var
  indice: Integer;
begin
  try
    for indice := Low(matriz) to High(matriz) do
      sentencia;
  except
    on EAccessViolation do
      // Tratamiento de la excepción
  end;
```

Las palabras clave `try` y `except` forman un bloque, colocándose entre ellas las sentencias que podrían generar una excepción. Tras `except` aparecerán una o más cláusulas `on`, asociada cada una de ellas a un tipo de excepción. El tipo más genérico es `Exception`, de la cual derivan varias decenas de excepciones especializadas. Una de ellas es `EAccessViolation` que se produce al intentar acceder a memoria no asignada, justamente lo que ocurre cuando el índice de una matriz está fuera de rango.

Nuestro propio código puede producir excepciones mediante la sentencia `raise`. Esto es útil cuando al verificar una condición de error se quiere delegar su tratamiento en el código que ha invocado a un procedimiento. La sintaxis de dicha instrucción es la siguiente:

```
raise excepción;
```

La excepción puede ser cualquiera de las predefinidas o bien un tipo de excepción definido a medida de las necesidades de la aplicación.

En ocasiones interesará ejecutar una o más sentencias con independencia de que se produzca o no una excepción. Las sentencias colocadas tras `try` dejarán de ejecutarse en cuando se detecte una excepción, mientras que las que siguen a `except` solamente se ejecutarían en caso de que tenga lugar. Aquellas sentencias que deban ejecutarse siempre se dispondrán en un bloque `finally`, tal y como se muestra en el siguiente ejemplo:

```
  pixeles: array of array of Byte;
begin
  SetLength(pixeles, 1280, 1024);
  try
     // Sentencias que pueden producir la excepción
  finally
    SetLength(pixeles, 1, 1);
  end;
```

Otras sentencias

Además de las citadas en los puntos previos Delphi cuenta con algunas sentencias más, algunas ya en desuso como es el caso de `goto`. Heredada de las primeras versiones de Pascal, esta sentencia permite desviar la ejecución a un punto concreto del código que se ha marcado con una etiqueta. La sintaxis de uso es la mostrada a continuación:

```
var
 label Salida;
begin
...
if hayQueSalir then goto Salida;
  // Sentencias a ejecutar si no hay que salir
Salida:
  // Sentencias a ejecutar en la salida
```

La etiqueta se declara de forma similar a una variable, pero con el prefijo `label` y prescindiendo de tipo. Después se dispone en el punto del código que interese seguida de dos puntos. En la mayoría de los casos este tipo de construcciones pueden componerse con sentencias condicionales, pero en ocasiones el uso de `goto` puede resultar mucho más simple y claro.

Otra sentencia de Delphi, descrita en un punto previo, es `with`. Su objetivo es asumir en un bloque de código una o más referencias a fin de no tener que repetirlas. La sintaxis general es la siguiente:

```
with ref1, ref2, ... do
  sentencia;
```

Clases y sus miembros

Al desarrollar una aplicación con Delphi las definiciones de tipos, declaraciones de variables, implementación de procedimientos y funciones y, en general, todas las sentencias formarán parte de una o más clases. Raramente introduciremos el código en un módulo de programa, de hecho dicho módulo lo mantiene el Gestor de proyectos como ya sabemos y en él se introducen únicamente las sentencias necesarias para poner la aplicación en marcha.

Delphi es un lenguaje orientado a objetos si bien no todos los elementos que lo forman pueden tratarse como tales, por lo que podría decirse que no es un lenguaje orientado a objetos puro. Existe una serie de tipos básicos, entre ellos los tipos numéricos, el tipo booleano o los caracteres, que son intrínsecos del lenguaje y no producto de la instanciación de una clase.

En los puntos siguientes se expone la sintaxis a emplear para definir clases en Delphi, aplicándolo a un ejemplo práctico que nos servirá para completar el proyecto `MicroClassic` creado en el capítulo previo. No se ofrecen explicaciones sobre los conceptos de orientación a objetos, tales como la encapsulación, al herencia, construcción de objetos o polimorfismo. Si esta terminología te resulta ajena deberás recurrir a un texto de introducción genérico sobre programación orientada a objetos.

Definición de una clase

Al igual que ocurre con los registros y, en general, con cualquier tipo de dato no propio del lenguaje, una clase ha de ser definida antes de poder utilizarla como tipo para declarar variables (objetos). Si pretendemos que la clase sea accesible desde otros módulos, desde el exterior del módulo en que está definiéndose, deberemos colocarla en la sección de interfaz.

Hay que diferenciar entre la definición de la clase, que es básicamente una enumeración de los miembros que forman parte de ella junto con sus tipos y otros atributos, y la implementación de los métodos con que cuente. Ésta se realiza por separado, método a método, y habitualmente se coloca en la sección de implementación del módulo.

Comenzaremos por tanto definiendo la clase, momento en el que fijaremos su nombre y, opcionalmente, la clase de la que es descendiente. Dado que

una definición de clase crea un nuevo tipo de dato la colocaremos en el apartado `type`. La sintaxis básica es la mostrada a continuación.

```
type

TMiClase = class[(ClaseBase)]
  miembros
  ...
end;
```

La clase ascendiente, en caso de que nuestra clase derive de otra, se indicará entre paréntesis tras la palabra clave `class`. El mecanismo de herencia dará a la nueva clase todas las características de aquella de la que deriva, punto de partida para diseñar una ampliación o especialización de dicha clase.

Los miembros de una clase pueden ser:

- **Variables**: En este contexto se denominan atributos o campos de la clase. Se usan las reglas ya conocidas para la declaración de variables, teniendo en cuenta que no existen por sí mismas sino en el interior de objetos de la clase en que se declaran.
- **Procedimientos/Funciones**: En una clase se llaman genéricamente métodos. Lo explicado anteriormente en relación a los parámetros y valores de retorno se aplica igualmente aquí. Habitualmente los métodos están especializados en tratar los atributos del objeto sobre el que trabajan.
- **Constructores**: Son métodos especializados en la inicialización de objetos de la clase que está definiéndose. Se designan con la palabra clave `constructor` y su nombre siempre es `Create`. En un apartado posterior profundizaremos algo más sobre ellos.
- **Destructores**: Su función es complementaria a la del constructor, encargándose de eliminar objetos de la clase. Se designan con la palabra clave `destructor` y su nombre siembre es `Destroy`. Nunca toman parámetros.
- **Propiedades** y **eventos**: Los componentes Delphi se crean definiendo clases que descienden de clases concretas definidas en la VCL o la FMX. Las clases de componentes además de atributos y métodos siempre ofrecen propiedades, que son la vía para fijar los atributos desde los diseñadores integrados en el entorno, y también eventos. Por el momento no nos preocuparemos de estos tipos de miembros.

Visibilidad de los miembros de una clase

Definir una clase en la sección de interfaz de un módulo no implica que se permita acceder libremente a todos sus miembros. Dentro de la propia clase se pueden establecer distintos niveles de visibilidad, agrupando los miembros en los apartados `private`, `protected`, `public` y `published`.

Los miembros privados son visibles solo para los demás miembros de la misma clase y también para el resto del código del módulo en que está definida[39]. Establece el tipo de acceso más restrictivo y suele aplicarse a los atributos a fin de que no puedan ser manipulados más que por la propia clase.

Cuando interesa que un miembro esté accesible para clases que pudieran derivar de la actual, pero no para el resto, lo que se hace es definirlo como protegido. Un atributo protegido sería accesible, por tanto, desde el código de la clase y el módulo en que se define, así como por cualquier clase descendiente con independencia de dónde se defina.

Todos aquellos miembros diseñados para un uso general se dispondrán en el apartado público, estando a disposición de cualquier módulo. Los constructores, salvo excepciones, siempre son públicos, al igual que todos aquellos métodos que no son de uso interno a la propia clase. También pueden serlo tipos de datos definidos en la clase: registros, enumeraciones, etc. Raramente se hacen públicos los atributos ya que se perdería totalmente el control sobre su contenido.

Finalmente, en el apartado `published` se *publican* normalmente las propiedades de clases de componentes a fin de que los diseñadores de Delphi puedan obtener la información que precisan para facilitar su manipulación. No es un requerimiento, pueden existir propiedades no publicadas que no aparecerían en el Inspector de objetos y, por tanto, no podrían usarse durante la fase de diseño pero sí serían accesibles desde el código del programa, durante su ejecución.

> ADVERTENCIA
>
> Los identificadores de la sección `published` han de ser ASCII, no UNICODE, por lo que no pueden usarse letras acentuadas ni la eñe.

---

39 Normalmente cada clase se define en un módulo propio, por lo que los miembros privados serían inaccesibles desde fuera.

Construcción de objetos

Al declarar una variable de un tipo básico, y posteriormente asignarle un valor, el propio compilador de Delphi se encarga de asignar la memoria necesaria y llevar a cabo la inicialización. Esto es así porque dichos tipos forman parte del lenguaje y se conoce su estructura, algo que no puede aplicarse a las clases que nosotros mismos definimos.

La declaración de una variable usando como tipo el nombre de una clase no conlleva la creación de un objeto de ésta (la *instanciación* de la clase) sino solamente la reserva de un pequeño espacio donde se guardará una referencia a dicho objeto. Por ello el primer paso a dar en el código será la construcción del objeto y asignación de la referencia a la variable, por ejemplo:

```
type
  class TMiClase = class
   ...
  end;

var
  miObjeto: TMiClase;
begin
  miObjeto := TMiClase.Create;
  ...
```

El constructor se encargará de asignar la memoria que se precise para almacenar todos los datos del objeto y, en caso necesario, realizar otras tareas como la asignación de valores iniciales, apertura de archivos, preparación de conexiones, etc. Si la propia clase no facilita un constructor Delphi incluirá uno por defecto que se ocupará de lo imprescindible.

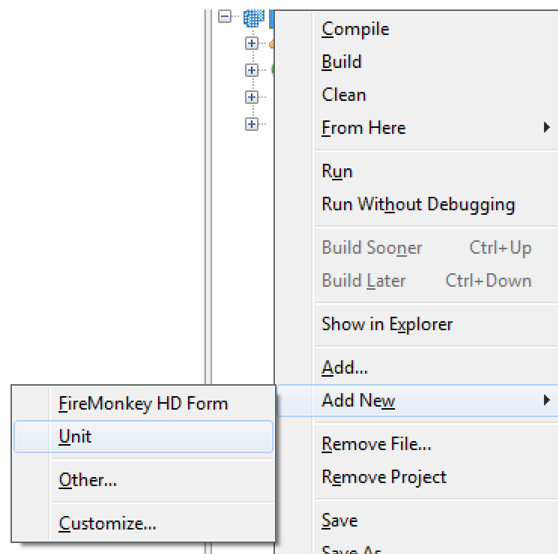
Una misma clase puede contar con varios constructores, siempre que éstos acepten listas de parámetros distintas. Con ello se consigue facilitar diferentes vías de inicialización de los objetos. En cuanto se define un constructor en una clase Delphi ya no incluye el constructor por defecto, por lo que si se quiere facilitar la creación de objetos sin necesidad de parámetros habrá que agregar explícitamente un constructor para ello.

Cuando una clase deriva de otra la primera se ocupará de la inicialización de los miembros que haya agregado, dejando a la segunda que realice su trabajo, invocando al constructor base con los parámetros adecuados. Para ello hará uso de la palabra clave `inherited` seguida del nombre del constructor y los parámetros que correspondan.

La clase TColeccionable

Con lo que sabemos hasta ahora podemos comenzar a definir una clase propia cuyo objetivo será contener la información básica correspondiente a cada objeto coleccionable del proyecto MicroClassic y facilitar su tratamiento. Denominaremos a esta clase TColeccionable.

El primer paso será agregar al proyecto un nuevo módulo de código, usando para ello la opción Add New>Unit de su menú contextual o del menú principal de Delphi.



Tras añadir el módulo lo guardaremos con el nombre Coleccionable. Su contenido inicial será el que puede verse a continuación:

```
unit Coleccionable;

interface

implementation

end.
```

El nombre del módulo es el que hemos dado al archivo a la hora de guardarlo. Por lo demás solamente tenemos las secciones de interfaz e implementación vacías.

Consideraremos como datos básicos de cualquier objeto coleccionable los siguientes:

- **Nombre**: Una cadena con el nombre del objeto, un dato que permitirá diferenciarlo del resto.
- **Año de presentación**: Un entero con el año en que se presentó el ordenador, lanzó el software, publicó la revista, etc.
- **Estado de conservación**: Es aplicable a todos los objetos, incluso al software si hacemos referencia al estado del soporte en que se encuentra. Definiremos un subrango `0..100` para este dato.
- **Descripción**: Una cadena con una descripción más extensa del objeto que la que ofrece su nombre.
- **Imágenes**: Una matriz donde conservaremos imágenes del objeto: fotografías del ordenador, digitalización de la portada de libros o revistas, capturas de pantalla del software, etc. El tipo base de la matriz sería `TBitmap`, definido en el módulo `FMX.Types`.

Además agregaremos los miembros necesarios para acceder a estos datos y cualquier otro necesario para el funcionamiento de la clase, por ejemplo un método para añadir imágenes o un atributo entero que nos permita saber cuántas imágenes tiene asociadas en cada momento.

Comenzaremos agregando a la sección `interface` la cláusula `uses` haciendo referencia al citado módulo `FMX.Types`, ya que vamos a necesitar un tipo definido en él:

```
uses
  FMX.Types; { Módulo con el tipo TBitmap }
```

El paso siguiente, en el apartado `type`, será la definición de los tipos propios que vayamos a usar en la clase. Serán dos: un subrango para el año de presentación y otro para el estado de conservación:

```
type
  TAñoPresentación    = 1965 .. 1995;
  TEstadoConservación =    0 .. 100;
```

Con esto ya tenemos lo necesario para proceder a definir la clase `TColeccionable`. Lo haremos en el apartado `type` de la misma sección `interface`, ya que nos interesa que esta clase sea accesible desde el exterior. De hecho ésta será la función básica de la clase `TColeccionable`, ya que por sí misma no hace referencia a un tipo de objeto concreto: un ordenador, una revista, un periférico, etc.

Para cada uno de esos tipos de objeto habrá que definir una clase especializada derivada de `TColeccionable`, actuando ésta como una clase *abstracta*[40].

La definición de clase la iniciaremos declarando los atributos privados, las variables en las que se almacenará la información asociada a cada uno de los objetos:

```
TColeccionable = class abstract
private
  Fnombre      : String;
  Faño         : TAñoPresentación;
  Festado      : TEstadoConservación;
  Fdescripción : String;
  Fimágenes    : array of TBitmap;
  nImágenes    : Integer;
```

También tendremos algunos métodos de uso interno a la clase, por lo que aparecerán asimismo como privados. Su objetivo será facilitar el acceso a algunas propiedades y realizar tareas que el usuario de la clase no tiene porqué conocer:

```
  { Métodos de uso interno a la clase }
  procedure setNombre(nombre: String);
  function  getImagen(Index: Integer): TBitmap;
  procedure ampliaImágenes;
```

La implementación de estos métodos se llevará a cabo más adelante, en la sección `implementation`. En la definición de la clase únicamente se introduce su nombre, lista de parámetros y tipo de valor devuelto en el caso de las funciones.

Iniciaremos la lista de miembros públicos de la clase con el constructor, solamente habrá uno, que necesitará como parámetro el nombre del objeto a crear:

```
public
  { Único constructor. Precisa el nombre del objeto }
  constructor Create(nombre: String);
```

Dado que al construir el objeto solamente se establecerá su nombre, la clase tendrá que facilitar un mecanismo que permita introducir el resto de los datos y recuperarlos cuando se precise. Básicamente tenemos dos opciones:

---

40 Se denominan clases abstractas aquellas marcadas con la palabra clave `abstract` delante del nombre de la clase, en la definición. La finalidad de una clase abstracta es exclusivamente servir como base de otras, razón por la que Delphi no permite crear objetos a partir de ellas.

escribir métodos con ese fin o bien exponer los atributos a través de propiedades. En este caso vamos a usar una combinación de ambas alternativas.

Las propiedades ofrecen una vía natural para acceder a los datos de un objeto mediante la sintaxis `objeto.propiedad` y, de cara al usuario de la clase, parecen variables ya que puede tanto leerlas como escribirlas. Pueden, no obstante, definirse propiedades solamente de lectura y, más raramente, solo de escritura. A pesar de esa apariencia, internamente la clase puede controlar tanto el proceso de lectura como el de escritura.

Comencemos por el nombre del objeto, establecido durante su creación. Definiremos la siguiente propiedad pública que servirá tanto para recuperarlo como para modificarlo:

```
property Nombre: String read Fnombre write setNombre;
```

Tras el nombre de la propiedad, que será el que se utilice para acceder a ella con independencia de cuál sea el nombre real del atributo en que se almacena la información, se indica su tipo. La cláusula `read` indica qué hay que hacer cuando se quiera leer la propiedad, en este caso recuperarla del atributo `Fnombre`. Análogamente, la cláusula `write` apuntará a la variable o método que debe usarse cuando se cambie el valor de la propiedad. Al facilitar el nombre de un método éste se ejecutará cada vez que se asigne un valor a la propiedad, pudiendo realizar las comprobaciones y trabajo adicional que sea preciso.

El método `setNombre`, definido anteriormente como privado, recibirá como parámetro un valor de tipo `String` ya que éste es el tipo de la propiedad `Nombre`. Si la cláusula `read` hiciese referencia a un método `getNombre`, éste se definiría como una función sin parámetros y que devuelve un valor de tipo `String`.

Para el resto de los datos definiríamos igualmente propiedades de acceso facilitando la lectura y controlando la escritura. Para este caso nos limitaremos a leer y escribir de los atributos asociados, sin usar métodos:

```
{ Acceso al resto de datos del objeto }
property Año : TAñoPresentación read FAño write FAño;
property Estado : TEstadoConservación read Festado
write Festado;
property Descripción : String read Fdescripción write
Fdescripción;
property NumImágenes : Integer read nImágenes;
```

La propiedad NumImágenes es solamente de lectura ya que el número de imágenes cambia de manera indirecta al agregar otras nuevas con el método que definiremos a tal efecto y, por tanto, no tiene sentido su modificación directa. Dicho método lo definimos como público de la siguiente forma:

```
procedure añadeImagen(imagen: TBitmap);
function tieneImágenes: Boolean;
```

Mediante la función tieneImágenes el usuario de la clase podrá comprobar si un cierto objeto tiene o no imágenes asociadas. En caso afirmativo podría interesarle acceder a cualquiera de ellas, fin para el que definiremos una propiedad indexada, es decir, que se comportará como una matriz. La definición de esa propiedad la efectuaremos así:

```
property Imagen[I: Integer] : TBitmap read getImagen;
```

Tras el nombre de la propiedad y entre corchetes se indica el nombre y tipo del índice o índices. El método de lectura recibirá ese parámetro y deberá devolver un objeto TBitmap como resultado.

En este punto habríamos completado la definición de la clase. El paso siguiente es la implementación de los diferentes métodos con que cuenta, colocando su código y cualquier otra declaración que se precise en la sección implementation del módulo de código. Comenzaremos por el constructor:

```
implementation

const INI_NUM_IMAGENES = 1;

{ El constructor precisa el nombre del objeto y procede
  a inicializar los miembros fundamentales }
constructor TColeccionable.Create(nombre: String);
begin
  setNombre(nombre);
  SetLength(Fimágenes, INI_NUM_IMAGENES);
  nImágenes := 0;
end;
```

La constante INI_NUM_IMAGENES fijará la capacidad inicial de la matriz dinámica Fimágenes que habíamos definido como atributo. Esa capacidad, como veremos después, irá duplicándose a medida que sea necesario.

Dado que Create es un miembro de la clase TColeccionable, al implementarlo hay que preceder su nombre con el de la clase. Esta regla se aplica a todos los miembros, no solamente al constructor, ya que de lo contrario serían interpretados como procedimientos y funciones independientes y no como parte de la clase.

Además de guardar el nombre del objeto coleccionable que se recibe como parámetro, el constructor también asigna el tamaño inicial de la matriz `FImágenes` y el número de imágenes que contiene actualmente dicha matriz.

Lo siguiente será implementar los métodos encargados de facilita la lectura y escritura de las propiedades. La propiedad `Nombre` se leía directamente del atributo en que estaba almacenada, pero su modificación se realiza a través del siguiente método:

```
{ Método de escritura en la propiedad Nombre }
procedure TColeccionable.setNombre(nombre: String);
begin
  Fnombre := nombre;
end;
```

En este caso no se hace nada especial, sencillamente se guarda el nuevo valor sustituyendo al anterior. En el método de escritura de una propiedad sin embargo pueden realizarse comprobaciones sobre la validez del nuevo valor, notificar el cambio para que se actualicen otros elementos, etc.

El segundo método asociado a una propiedad es el de lectura de la propiedad `Imagen`, una propiedad indexada que devuelve un objeto `TBitmap` como resultado. Su implementación es la siguiente:

```
{ Facilita el acceso a las imágenes del objeto }
function TColeccionable.getImagen(Index: Integer):
TBitmap;
begin
  if Index < nImágenes then
    Result := FImágenes[Index]
  else
    Result := nil;
end;
```

Aquí la lectura sí que implica una comprobación previa, concretamente del índice facilitado ya que éste podría estar fuera del rango real de imágenes asociadas al objeto. Si el índice es inferior al número de imágenes (hay que recordar que es una matriz dinámica y por tanto el primer elemento es `0`) se devuelve la imagen correspondiente recuperándola del atributo `FImágenes`. En caso contrario se devuelve el valor `nil` que es el equivalente en Delphi al `NULL` de otros lenguajes de programación, representando la ausencia de valor. Si no hiciésemos esta comprobación, y usásemos directamente el índice facilitado para acceder a la matriz, podría producirse una excepción que habría que controlar. Nuestro método también podría optar por generar una excepción, con la sentencia `raise`, en lugar de devolver `nil`.

Lo único que nos queda por implementar de la clase es el método público encargado de agregar una nueva imagen al objeto y el método privado que se encargará de ampliar la matriz que las almacena. El código de dichos métodos es el mostrado a continuación:

```
{ Agrega una imagen a la lista de imágenes del objeto }
procedure TColeccionable.añadeImagen(imagen: TBitmap);
begin
  if nImágenes = Length(FImágenes) then
    ampliaImágenes;

  FImágenes[nImágenes] := imagen;
  inc(nImágenes);
end;

{ Amplía la matriz al doble de su tamaño actual }
procedure TColeccionable.ampliaImágenes;
begin
  SetLength(FImágenes, Length(FImágenes) * 2);
end;
```

En el método `añadeImagen` lo primero que hacemos es comprobar si el número actual de imágenes es igual al tamaño (la capacidad) de la matriz que las contiene. En ese caso es necesario ampliar el tamaño de la matriz, un trabajo que delegamos en el método `ampliaImágenes` y que, como puede apreciarse en el código de dicho método, es una tarea realmente sencilla: se invoca al procedimiento `SetLength` para ajustar el tamaño al doble del actual, multiplicando por 2 el valor devuelto por la función `Length`.

> NOTA
>
> El procedimiento `SetLength` conservará el contenido actual de la matriz cuyo tamaño está modificándose siempre y cuando éste sea superior al actual, es decir, si se reduce el tamaño lógicamente se perderá parte de la información almacenada.

Una vez nos hemos asegurado de que en la matriz hay espacio para la nueva imagen procedemos a almacenarla. Puesto que el número de imágenes se ha incrementado hacemos lo propio con el atributo `nImágenes`. Para ello usamos el procedimiento `inc`, útil para incrementar en una unidad cualquier variable de un tipo ordinal: entero, enumeración o subrango. Este procedimiento cuenta con un complementario llamado `dec` que, como es fácil imaginar, reduce en una unidad la variable entregada como parámetro.

Nuestra clase no cuenta con un destructor porque no tiene necesidad de liberar explícitamente recurso alguno: liberar memoria, cerrar archivos o conexiones. El único elemento dinámico es la matriz `FImágenes` pero su liberación será llevada a cabo automáticamente por Delphi cuando no exista referencia alguna a ella. Si en lugar de una matriz hubiésemos usado un puntero a `TBitmap` y asignado memoria dinámicamente, mediante los procedimientos `New` o `GetMem`, la responsabilidad de liberar esa memoria sería nuestra facilitando el puntero a `Dispose` o `FreeMem` y, por tanto, sería imprescindible implementar un destructor para ello.

La clase TOrdenador

La clase `TColeccionable` definida en el punto anterior es abstracta, lo que significa que no puede ser usada para crear objetos y, en principio, su utilidad podría parecer nula. El objetivo de esa clase es servir como base para el resto de las que existirían en el proyecto, aportando los atributos y métodos básicos con que ha de contar cada una de ellas y permitiendo agregar otros elementos para derivar clases especializadas. Además el hecho de que exista una base común permitirá tratar todos los objetos de la colección de forma genérica o polimórfica.

Tomando como base a `TColeccionable` la primera clase que derivaremos será `TOrdenador`, especializada en un tipo de objeto coleccionable concreto: los ordenadores. Habrá que agregar al proyecto un nuevo módulo de código, como hicimos al inicio del punto previo, que guardaremos con el nombre `Ordenador`.

Comenzaremos la sección de interfaz del nuevo módulo incluyendo en la cláusula uses el módulo `Coleccionable` y definiendo los tipos que sean necesarios para `TOrdenador`, en este caso únicamente la enumeración `TBitsOrdenador` que creábamos en un apartado previo de este mismo capítulo:

```
interface

uses
  Coleccionable;

type
  TBitsOrdenador = (bo8, bo16, bo32);
```

El inicio de la clase contendrá los atributos privados necesarios para almacenar la información de un ordenador. Dado que la clase TOrdenador va a ser derivada de TColeccionable, lo cual indicaremos disponiendo el nombre de ésta entre paréntesis tras la palabra clave class, los atributos de cualquier objeto coleccionable ya están presentes. Solo tendremos que agregar los específicos para un ordenador:

```
  TOrdenador = class(TColeccionable)
   private
    Ffabricante : String;
    Fmodelo     : String;
    Fbits       : TBitsOrdenador;
    FRAM        : SmallInt;
    FROM        : SmallInt;
    Ffunciona   : Boolean;
```

A continuación tendríamos los miembros públicos de la clase: el constructor y las propiedades que darán acceso a la información. En este caso todas ellas leerán y escribirán directamente en los atributos respectivos, aunque en la práctica lo habitual es que la escritura se delegue en un método que controle la validez de la información:

```
public
 constructor Create(nombre: String);

 property Fabricante: String           read Ffabricante
                                       write Ffabricante;
 property Modelo    : String           read Fmodelo
                                       write Fmodelo;
 property Bits      : TBitsOrdenador   read Fbits
                                       write Fbits;
 property RAM       : SmallInt         read FRAM
                                       write FRAM;
 property ROM       : SmallInt         read FROM
                                       write FROM;
 property Funciona  : Boolean          read Ffunciona
                                       write Ffunciona;

end;
```

El único elemento de la sección de implementación, por tanto, será el constructor al no existir métodos adicionales. Al igual que el constructor de TColeccionable, éste recibe como único parámetro el nombre del nuevo objeto, dato que ha de almacenarse en el atributo correspondiente. Ese atributo se declaró como privado en TColeccionable, por lo que no es accesible ni siquiera para los métodos de una clase derivada como es TOrdenador.

De hecho la clase derivada no tiene porqué saber cómo almacena los datos la clase ascendiente, ése es un trabajo que no le compete. Lo único que ha de hacer es delegar la inicialización de los atributos heredados en el constructor de la clase base, ocupándose después de inicializar los que le son propios. De esta forma el constructor quedaría así:

```
implementation

constructor TOrdenador.Create(nombre: String);
begin
  inherited Create(nombre);

  Fbits     := bo8;
  FRAM      := 64;
  FROM      := 32;
  Ffunciona := True;
end;
```

Usando la palabra clave `inherited` invocamos al constructor de la clase base facilitándole los datos que necesita. Una vez que él ha terminado su trabajo procedemos a asignar los valores que tendrán inicialmente el resto de atributos. En este caso hemos optado por los que presumiblemente son más comunes en la colección de nuestro cliente, ya que casi todos ellos son ordenadores de 8 bits con 64 KB de memoria RAM y que funcionan.

De manera análoga a ésta definiríamos el resto de las clases del proyecto: `TSoftware`, `TPeriférico`, `TPublicación`, etc. Todas ellas derivarían de `TColeccionable` y contarían con algunos atributos específicos y posiblemente también con métodos a medida.

Miembros de clase y el objeto self

Los atributos que hemos definido en las clases previas son miembros de instancia. Esto significa que cada uno de los objetos creados a partir de la clase tendrá su conjunto de atributos propio e independiente del resto de los objetos, algo imprescindible cuando se trata de mantener información diferenciada pero con una estructura común.

Por el contrario, y como es lógico por otra parte, no se crea una copia del código de los métodos, sino que éste es compartido por todos los objetos de la clase. Cabe preguntarse entonces cómo sabe un método durante su ejecución sobre qué objeto debe actuar. Aquí es donde entra en escena la variable `self` que siempre mantiene una referencia al objeto actual.

Siempre que en las sentencias de un método se haga referencia a miembros de instancia de la clase sin preceder su nombre con el de un objeto concreto se asume que dicho objeto es el apuntado por self. La referencia puede ser explícita, por ejemplo para evitar ambigüedades al coincidir dos identificadores, uno perteneciente a un miembros de la clase y otro que no, como podría ser el caso siguiente:

```
procedure TOrdenador.setRAM(FRAM: SmallInt);
begin
  self.FRAM := FRAM;
end;
```

En este caso el parámetro que recibe el método tiene el mismo nombre que el atributo asociado, por lo que para distinguir entre uno y otro se compone la referencia self.atributo.

Una clase también puede contar con miembros de clase, en contraposición a los miembros de instancia. Dichos elementos, también conocidos como miembros estáticos, se asocian con la clase en sí y no con los objetos que se crean a partir de ella, por lo que no existe una copia para cada objeto. Es útil contar con atributos de clase cuando se precisa compartir alguna información entre todos los objetos. La declaración de un atributo de clase se hace en el apartado class var de la clase. Para hacer referencia a un atributo de clase no puede emplearse la sintaxis self.atributo ya que no pertenece a un objeto concreto, en su lugar la referencia correcta sería clase.atributo.

Los métodos de clase, por su parte, pueden ser ejecutados sin disponer de una referencia a un objeto mediante la sintaxis clase.método. Este tipo de métodos no disponen de la variable self descrita anteriormente, razón por la que no les es posible acceder a atributos de instancia salvo que se les facilite una referencia a un objeto por otros medios. La definición de un método de clase va precedida de la palabra clave class.

Además de atributos y métodos, en una clase también pueden existir propiedades de clase, constructores de clase y destructores de clase. Las propiedades son similares a los métodos en el sentido de que pueden ser invocadas con la sintaxis clase.propiedad y no cuentan con la referencia self. Simplemente son un atajo para acceder más fácilmente a atributos de clase, sin más.

En cuanto a los constructores y destructores de clase su objetivo es facilitar la inicialización de los atributos de clase de manera automática, en cuanto se haga la primera referencia a la clase, y de manera análoga su destrucción.

Supongamos que para el proyecto MicroClassic queremos disponer de una clase que mantenga toda la colección de objetos y facilite su gestión. De dicha clase no tiene sentido que exista más de una instancia, aunque contenga muchos objetos la colección es solamente una. Podemos recurrir al conocido patrón de diseño *singleton*, usando un atributo de clase que almacene la única referencia existente a un objeto de la clase, el habitual método getInstance que facilita su recuperación y un constructor de clase que se encargue de la inicialización.

La nueva clase se llamaría TColeccion y se alojaría en el módulo Coleccion que agregaríamos al proyecto. La definición, en la sección de interfaz, sería la siguiente:

```
TColeccion = class
private
  class var refInstance: TColeccion;

  { Miembros de instancia de la clase para
    mantener los objetos de la colección }

  class constructor Create;
  class destructor  Destroy;

  constructor Create;

public
  class function getInstance: TColeccion;
  class property Instance: TColeccion read refInstance;
end;
```

refInstance es un atributo de clase cuya finalidad será almacenar la referencia al único objeto de la clase TColeccion que podrá existir. Podríamos agregar miembros adicionales, tanto de clase como de instancia, a fin de dotar a la clase de la funcionalidad esperada.

Además del constructor de clase también existe un constructor corriente, ambos privados, así como el destructor de clase. Los únicos miembros públicos son la función getInstance y la propiedad Instance, ambas con la misma finalidad: entregar la referencia almacenada en el atributo refInstance.

Pasamos a la sección de implementación con el código del constructor y el destructor de clase. El primero será invocado automáticamente por Delphi, no necesariamente al poner en marcha la aplicación pero siempre antes de que se acceda por primera vez a cualquiera de los miembros de la clase. No es un método accesible al usuario de la clase.

```
class constructor TColeccion.Create;
begin
  if refInstance = nil then
    refInstance := TColeccion.Create;
end;

class destructor TColeccion.Destroy;
begin
  refInstance.Free;
  refInstance := nil;
end;
```

El constructor verifica que `refInstance` no tiene aun una referencia y procede a crear el objeto `TColeccion`. El destructor[41] lleva a cabo el trabajo inverso, liberando el objeto y volviendo a asignar `nil` a la variable.

La llamada desde el constructor de clase al constructor `Create` no ha de confundirse con una invocación a sí mismo, lo cual provocaría una recursión infinita. Lo que se hace es delegar en el constructor estándar la creación del objeto `TColeccion`. En este caso nos limitamos a llamar al constructor base sin parámetros, mediante la palabra clave `inherited`, y añadimos una comprobación: si ya existe un objeto y se intenta crear otro se genera una excepción.

```
constructor TColeccion.Create;
begin
  inherited;
  if refInstance <> nil then
     raise Exception.Create(
      'No se pueden crear más objetos de esta clase');
end;
```

La excepción no llegará a producirse nunca ya que a este constructor únicamente se le invoca desde el constructor de clase y éste ya comprueba que no exista un objeto previo.

Resta solo la implementación del método `getInstance` que, como puede verse a continuación, es muy sencilla:

```
class function TColeccion.getInstance: TColeccion;
begin
  result := refInstance;
end;
```

---

41 Teóricamente el destructor nunca debería ser invocado si no se ha ejecutado previamente el constructor, por lo que es seguro invocar al método `Free` para proceder a la liberación del objeto. No obstante podría introducirse una comprobación previa del tipo `if Assigned(refInstance) ...`.

Una vez implementada la clase, desde cualquier punto de la aplicación podríamos acceder a la información sobre la colección usando el método `getInstance` o la propiedad `Instance` de `TColeccion`, ya sea almacenando la referencia en una variable o usándola directamente para acceder a los miembros de instancia:

```
var
  MiColeccion: TColeccion;
...
  MiColeccion := TColeccion.getInstance;
```

NOTA

Puesto que `TColeccion` es una clase de uso final, no pensada para derivar otras a partir de ella, podría marcarse con clase final añadiendo la palabra clave `sealed` tras `class`, en la cabecera. De esta forma se impediría explícitamente que `TColeccion` pudiera aparecer como clase base de otras.

Métodos anónimos

Los procedimientos y funciones definidos en las clases puestas como ejemplo en este capítulo se caracterizan por tener asociado un identificador, lo cual facilita su invocación desde otros puntos en los que se necesite su funcionalidad. Podríamos decir que los métodos, tal y como los hemos conocido en los puntos previos, son la vía habitual de agrupar sentencias bajo un nombre y así poder ejecutarlas a demanda.

Delphi permite obtener la dirección de un método, guardarla y facilitarla como parámetro. El tipo de dato asociado es un puntero a un método o referencia a un método. Éste, sin embargo, sigue tomando la forma habitual que ya conocemos y tiene asociado un nombre, una definición y una implementación.

En contraposición los métodos anónimos son porciones de código a los que no se asocia un nombre, siendo directamente asignados a una variable, entregados como parámetro a un método o devueltos como resultado desde una función. Sus aplicaciones son múltiples.

Un caso típico en el que resultan útiles los métodos anónimos se da al implementar un algoritmo de ordenación genérico, otorgando al que va a utilizarlo libertad para que aporte su función de comparación. Esto le permitirá usar el mismo algoritmo para, por ejemplo, ordenador ascendente o descendentemente o hacerlo por distintos atributos. Una función de este tipo podría ser la siguiente:

```
procedure Ordena(datos: TStringDynArray;
                 compara: TFuncComparador);
var
  indice, posicion: Integer;
  auxiliar: String;
begin
  for indice := Low(datos)+1 to High(datos) do
    begin
      auxiliar := datos[indice];
      posicion := indice - 1;
      while ((posicion >= Low(datos)) and
           compara(auxiliar, datos[posicion])) do
        begin
          datos[posicion+1] := datos[posicion];
          dec(posicion);
        end;
       datos[posicion+1] := auxiliar;
    end;
end;
```

El algoritmo implementado es el de inserción, sencillo y bastante eficiente para colecciones de datos pequeñas.

El primer parámetro que recibe la función es un `TStringDynArray`. Éste no es más que un nombre que se da en `System.Types` al tipo `array of String` a fin de facilitar su uso. Se trata, por tanto, de una matriz dinámica de cadenas.

Lo importante en este momento es el segundo parámetro, llamado `compara` y que, como vemos en el código del procedimiento, es usado a modo de función en el bucle `while`. Se le entregan dos elementos de la matriz y se espera que devuelve un valor booleano indicando si están en orden o, por el contrario, hay que intercambiarlos. El tipo `TFuncComparador` lo hemos de definir con anterioridad, como un tipo de dato más, teniendo en cuenta todos esos detalles: si es una función o un procedimiento, lista de parámetros y tipo de retorno. Para este ejemplo la definición sería la mostrada a continuación:

```
TFuncComparador = reference to function(I1,I2: String):
Boolean;
```

Para usar el anterior método de ordenación tendremos que entregar la matriz con las cadenas de caracteres y la función de comparación. Ésta se facilitará como un método anónimo según puede verse en el código siguiente:

```
var
  meses: TStringDynArray;

begin
  ...
  meses := SplitString('Ene,Feb,Mar,Abr,May,Jun,' +
                       'Jul,Ago,Sep,Oct,Nov,Dic', ',');

  Ordena(meses, function(l1, l2: String): Boolean
    begin
      if l1 < l2 then
        result := true
      else
        result := false;
    end);
...
```

Observa que el segundo parámetro al llamar a `Ordena` comienza con la palabra `function`, lo cual denota que es un método anónimo de tipo función. Se indican los parámetros y el tipo de retorno pero no se dispone un punto y coma a continuación. Las sentencias a ejecutar se colocan en un bloque `begin/end` que también termina sin punto y coma. Lo que indica el final del método es el cierre de paréntesis de la llamada al método de ordenación.

Tras la llamada, la matriz entregada como argumento ya se encuentra ordenada. Su contenido inicial se ha obtenido dividiendo una cadena mediante la función `SplitString`, cuyo segundo parámetro indica el separador empleado.

Dado que el método anónimo devuelve `true` siempre que el primer elemento sea menor que el segundo, indicando que ése es el orden correcto, el resultado será una ordenación ascendente. Lo interesante es que el algoritmo de ordenación sirve también para ordenar a la inversa, no habría más que cambiar en el método anónimo el operador < por >; para ordenar distinguiendo o no entre mayúsculas y minúsculas, teniendo en cuenta otros símbolos, etc. No habría más que introducir los cambios adecuados en el código que se entrega como segundo argumento.

Tipos genéricos

Al definir una clase por regla general siempre se hacen ciertas suposiciones sobre los tipos de los datos con que tratará. En el punto anterior, por ejemplo, el método de ordenación asume que la matriz será de cadenas de caracteres, pero podría interesarnos ordenar números o fechas, por poner dos ejemplos. El algoritmo en sí sería el mismo, lo único que cambiaría sería el tipo de la matriz de entrada y una variable interna, aparte del método anónimo que indica cuándo el orden es correcto.

Delphi permite parametrizar la definición de clases y métodos en función de uno o más tipos. Lo que se define es una clase genérica, ya que podría aplicarse a cualquier tipo de dato[42], que pasará a convertirse en concreta cuando se aplique para un tipo dado. En este sentido los tipos genéricos de Delphi serían análogos a las plantillas (*templates*) con que cuenta C++ o los tipos genéricos de Java.

El primer paso a dar es la definición del tipo de genérico. Una vez se tiene éste se procede a instanciar el tipo concreto, dando valor a los parámetros que se usen como tipos en la definición original. Finalmente las clases se usan para crear objetos como cualesquiera otras.

Definir un tipo genérico

La definición de un tipo genérico se diferencia, respecto a la de cualquier otro, en la inclusión de uno o más parámetros entre los símbolos < y > tras el nombre del tipo. Cada uno de esos parámetros es un identificador que posteriormente, durante la instanciación, será sustituido por un tipo de dato concreto. La sintaxis general es la siguiente:

```
NombreTipo<Tipo1, ..., TipoN> = class
  // En la definición de NombreTipo se usan los
  // parámetros TipoI como si fuesen tipos de datos
end;
```

Cada uno de los parámetros puede ser usado para indicar el tipo de atributos y parámetros, así como para definir otros tipos a partir de ellos.

---

42 Es posible establecer restricciones en la definición del tipo genérico, pero por el momento supondremos que puede ser cualquier tipo.

Supondremos que nos interesa contar con el algoritmo de ordenación usado anteriormente como ejemplo pero aplicable a varios tipos de datos. Tenemos básicamente dos opciones: definir un procedimiento de ordenación distinto para cada tipo de dato a ordenar, algo que podría hacerse sin problemas gracias a la sobrecarga, o bien optar por definir un tipo genérico que se encargase de ese trabajo. Eso es lo que hacemos con el código mostrado a continuación:

```
type

TOrdena<TipoDato> = class
type
  TipoArray = array of TipoDato;
  TFuncComparador = reference to
       function(I1,I2: TipoDato): Boolean;
  class procedure Ordena(datos: TipoArray;
                         compara: TFuncComparador);
end;
```

Únicamente hay un parámetro asociado a la clase `TOrdena`: `TipoDato`. Éste se usa para definir un tipo compuesto: `TipoArray`, así como para indicar el tipo de los parámetros que habrá de tomar la función de comparación. El método de ordenación en sí se ha declarado como un procedimiento de clase, de forma que podrá utilizarse directamente sin necesidad de crear antes un objeto.

A la hora de implementar el método `Ordena` hemos de tener en cuenta que el tipo de los elementos ya no es `String` sino `TipoDato`, por lo que habrá que cambiar la cabecera y también la declaración de la variable auxiliar. El resto del código permanecerá invariable.

```
class procedure TOrdena<TipoDato>.Ordena(datos: TipoArray;
                         compara: TFuncComparador);
var
  indice, posicion: Integer;
  auxiliar: TipoDato;
begin
  ...
end;
```

Observa que en la cabecera del método, al implementarlo, su nombre va precedido del nombre de la clase a la que pertenece y la misma lista de parámetros con que ésta se definió. De esta forma lo que se está facilitando es una plantilla de la implementación del método `Ordena`, plantilla a partir de la cual después se crearán tantas versiones diferentes del método como tipos concretos sea necesario tratar.

Instanciación y uso de un tipo genérico

Tras la definición del tipo genérico llega el momento de instanciarlo para los tipos concretos que se necesiten, operación que generará a partir de la plantilla una clase distinta para cada uno de esos tipos de datos.

En nuestro caso nos interesa poder ordenar tanto cadenas de caracteres como números enteros, por lo que generaremos dos instancias de la clase genérica `TOrdena` tal y como se muestra en el código siguiente:

```
type

TOrdenaString = TOrdena<String>;
TOrdenaInteger = TOrdena<Integer>;
```

A partir de este momento `TOrdenaString` y `TOrdenaInteger` serían dos clases diferentes pero que ofrecen lo mismo: el tipo `TipoArray`, el tipo `TFuncComparador` y el método `Ordena`. Los usaríamos como puede verse a continuación:

```
var
  meses: TOrdenaString.TipoArray;
  dias: TOrdenaInteger.TipoArray;
...
begin
  SetLength(dias, 10);
  SetLength(meses, 12);
..// Introducir datos en las matrices

  TOrdenaInteger.Ordena(dias,
   function(I1, I2: Integer): Boolean
    begin
      result := I1 < I2;
    end);

  TOrdenaString.Ordena(meses,
   function(I1, I2: String): Boolean
    begin
      result := I1 < I2;
    end);
```

En ambos casos se entrega como segundo argumento un método anónimo fundamentalmente idéntico, lo que cambia es el tipo de los argumentos que se reciben. Tras la llamada al método `Ordena` de cada clase las dos matrices quedarían ordenada. Análogamente podríamos ordenar matrices de cualquier otro tipo.

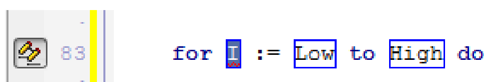
Asistencia a la edición de código

Si a medida que has leído este capítulo has ido poniendo en práctica los ejercicios propuestos, escribiendo personalmente el código de los diferentes ejemplos, seguramente ya habrás notado que Delphi ofrece múltiples funciones de asistencia a la edición de código.

Algunas de ellas se ponen en marcha automáticamente: al disponer un punto tras el nombre de un objeto aparece una lista con sus miembros, al abrir unos paréntesis para invocar a un método una ventana flotante informa sobre los parámetros que se esperan, etc. Hay otras menos inmediatas pero igualmente interesantes.

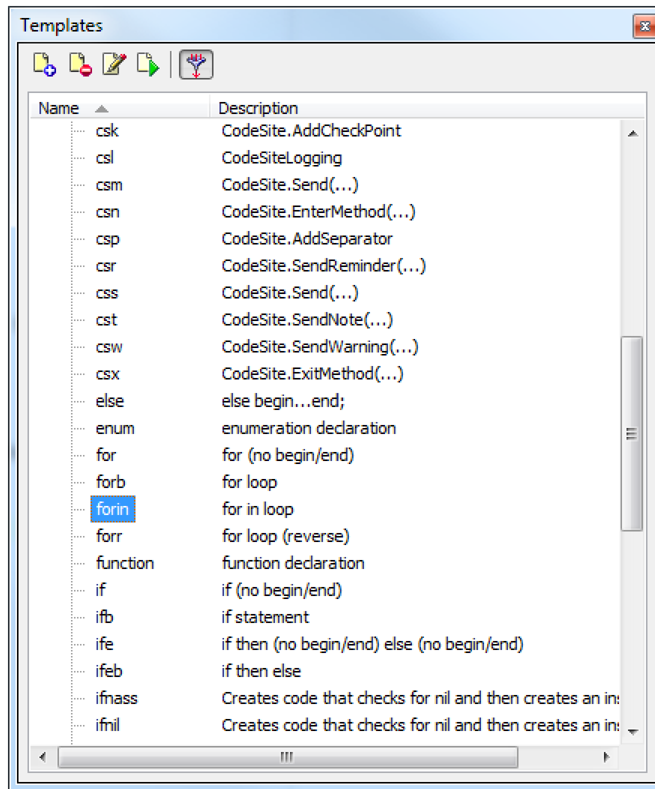
Plantillas de código

Esta función se desencadenan al introducir secuencias concretas de caracteres en el editor de texto. Si escribimos `for` y pulsamos la barra espaciadora o el tabulador en el editor aparece una plantilla de código, con el aspecto de la figura inferior:



El cursor se coloca en el primer recuadro, donde se espera que introduzcamos el nombre de la variable que actuará como contador del bucle, y la pulsación del tabulador avanzará a los recuadros que hay a la derecha, donde han de fijarse los límites inferior y superior del bucle. De esta forma se facilita la introducción en el código de estructuras usadas de forma habitual, ahorrándonos trabajo.

Si en lugar de un bucle `for` clásico necesitamos uno para recorrer los elementos de una colección no tenemos más que escribir `forin` en lugar de `for`, ejecutando otra de las plantillas predefinidas. Mediante la opción View>Templates podemos acceder a la ventana Templates, con la lista de todas las plantillas disponibles (véase la figura de la página siguiente). Los botones de la parte superior permiten agregar nuestras propias plantillas, así como eliminar y modificar las ya existentes.



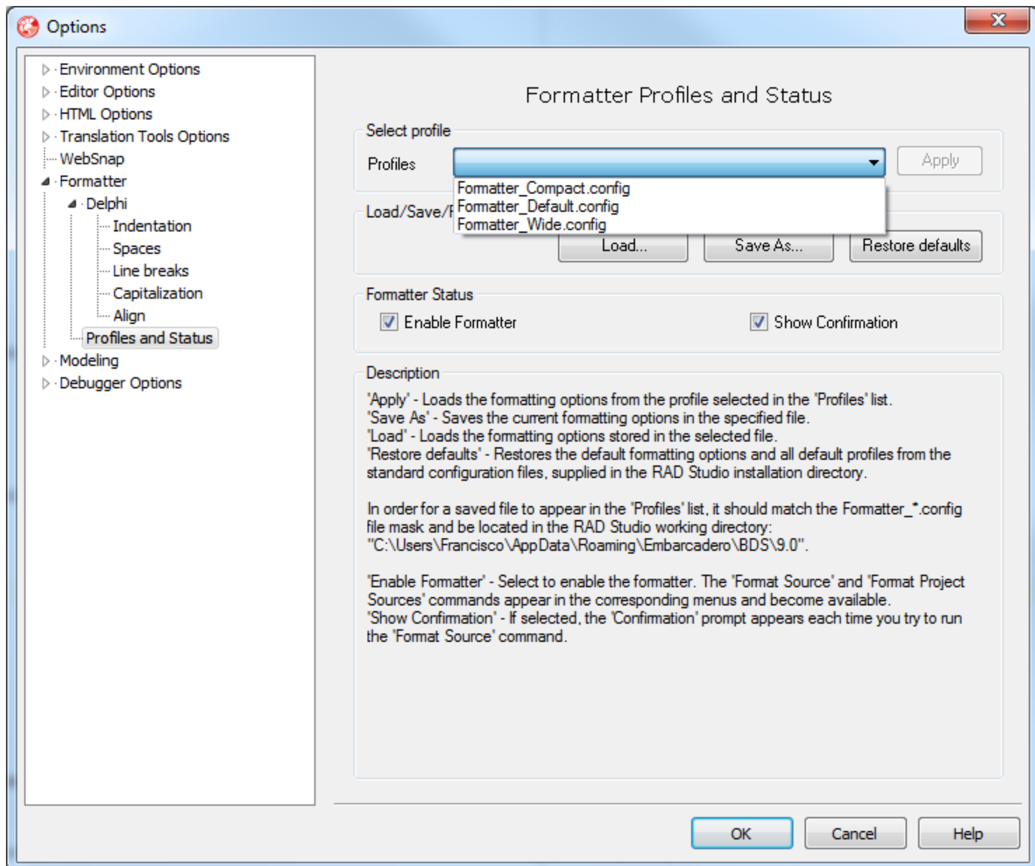
Además de las plantillas Delphi también introduce automáticamente código en ciertas situaciones, por ejemplo al iniciar la definición de un tipo, como puede ser una clase, la palabra clave `end` al final se inserta de manera inmediata. Lo mismo ocurre al iniciar un bloque de código con la palabra `begin` y pulsar Intro.

Otra función útil en este sentido es el autocompletado de clases, consistente en la generación automática de esqueletos vacíos de todos aquellos miembros de una clase que precisen implementación. No hay más que realizar la definición, colocar el cursor dentro del código de la clase y usar el atajo Control-Mayúsculas-C.

Formatear el código

A pesar de la ayuda que ofrece Delphi introduciendo código automáticamente en muchos casos, lo cierto es que la mayor parte de éste lo deberemos escribir personalmente y, al hacerlo, no siempre es fácil mantener un formato homogéneo y consistente que facilite la posterior revisión de ese código.

En cualquier momento podemos dar formato a todo el código del módulo en que estemos trabajando mediante la opción Edit>Format Source. También tenemos a nuestra disposición la opción Format Project Sources en el menú Project, mediante la que se daría formato a todos los módulos de código del proyecto. El formato automático se ajustará a los parámetros establecidos en la página Formatter de la opción Tools>Options de Delphi.



Navegar por el código

Cuando se trabaja en un proyecto de una cierta envergadura, con decenas o cientos de módulos conteniendo miles de líneas de código, localizar el punto que interesa en cada momento puede resultar entretenido. Por ello Delphi incorpora opciones que ayudan a esta tarea.

La herramienta más inmediata, siempre a la vista inicialmente, es el panel Structure que muestra la estructura del módulo de código actualmente en el editor. En dicha ventana se enumeran la clases y tipos de datos, así como cada uno de sus miembros. Basta un doble clic sobre cualquiera de ellos para ir directamente a su declaración o implementación, según los casos.

Cuando lo que buscamos no se encuentra en el módulo actual el panel anterior no resulta útil. Podemos recurrir a varias de las opciones del menú Search para localizar una clase concreta, realizar una búsqueda en todos los archivos del proyecto, localizar los puntos en los que se hace referencia a un cierto símbolo, etc.

Un atajo interesante es el que permite acceder a la declaración o implementación de un símbolo existente en el código. No hay más que mantener pulsada la tecla Control y hacer clic sobre la variable, método o clase, como si de un hiperenlace se tratase, para abrir el módulo en que se encuentre y colocar el cursor en la línea en que aparece.

Refactorización del código

En el menú Refactor de Delphi se encuentran todas las opciones que permiten *refactorizar* el código, una técnica que tiene por objetivo ir mejorando la calidad de éste a medida que se trabaja en él.

Un ejemplo típico en que es útil la refactorización: escribiendo la implementación de un método introducimos un conjunto de sentencias y más adelante, en otro método, nos damos cuenta de que necesitamos hacer lo mismo. Seleccionamos el conjunto de sentencias original y usamos la opción Refactor>Extract Method para convertirlas en un método, sustituyendo su aparición original por una llamada.

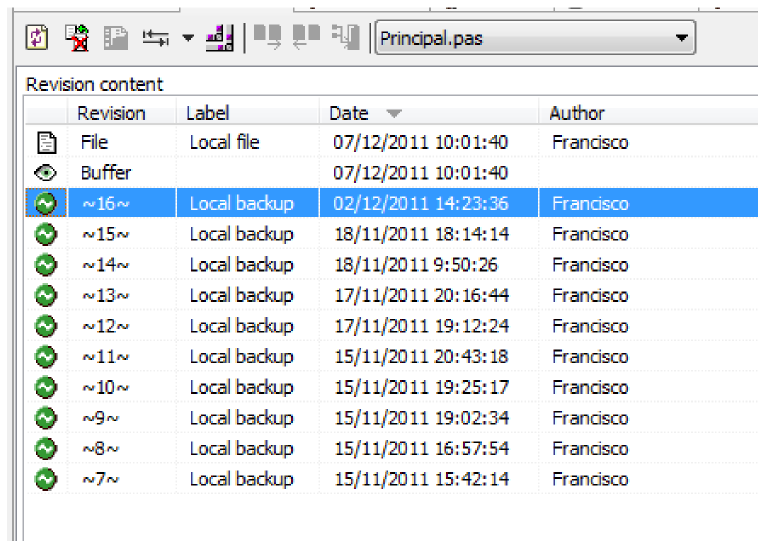
Algunas opciones de refactorización son sencillas, como puede ser renombrar un símbolo en todas sus apariciones, pero otras conllevan tareas más complejas que nos ahorrarán mucho tiempo de trabajo.

Acceso a versiones previas

Aunque al desarrollar software siempre es recomendable emplear un sistema de control de versiones de código fuente, lo cierto es que cuando el trabajo no es en grupo sino individual mucha veces se prescinde de esa útil herramienta.

Delphi conserva automáticamente las últimas versiones de cada uno de los módulos de código que componen un proyecto, ofreciendo opciones para la comparación y recuperación de versiones previas. Para acceder al historial de versiones de cualquier módulo no hay más que hacer clic en la pestaña History (en la parte inferior del editor).

Además de mostrar la lista de versiones existentes, inicialmente en orden cronológico inverso (desde el más actual al más antiguo), en la parte superior aparecerán una serie de botones que dan acceso a las opciones de comparación[43], anotación de comentarios sobre cada versión, recuperación, etc.



43 El motor que está detrás de las opciones de comparación de versiones es una versión reducida del software Beyond Compare incluida en algunas ediciones de Delphi. Contacta con Danysoft para obtener más información sobre la versión comercial.

Conclusión

El lenguaje Delphi es un lenguaje moderno, orientado a objetos, al uso y desarrollo de componentes y que cuenta con características avanzadas como los métodos anónimos o los tipos genéricos. Es un lenguaje que necesitamos conocer a fondo si queremos aprovechar toda su potencia durante el desarrollo de aplicaciones con Delphi, algo que requerirá un estudio detallado y extenso.

Este capítulo ha introducido los aspectos más importantes del lenguaje, pero resulta insuficiente para dominar por completo todas las características de Delphi. En realidad podríamos dedicar todo este libro únicamente a estudiar el lenguaje y nos faltaría espacio. Las explicaciones de los puntos previos han de servir como primera toma de contacto, por una parte, y también como referencias para poder profundizar en los aspectos que se vayan necesitando, por otra. Para ello no hay más que abrir la documentación electrónica de Delphi y realizar una búsqueda adecuada: tipos de datos, definición de clases, sentencias, etc.

A continuación

Con este tercer capítulo concluye la primera parte del libro, cuyo estudio nos habrá servido para sentirnos cómodos en el entorno de Delphi, saber cuáles son los pasos a seguir para desarrollar un proyecto y conocer los fundamentos del lenguaje.

Con estos pilares nos dirigimos a la segunda parte, dedicada al acceso a bases de datos desde Delphi. Son cuatro capítulos en los que analizaremos los diferentes mecanismos que nos ofrece para esta tarea, poniéndolos en práctica a través de ejercicios sencillos similares a los desarrollados en éste y el capítulo previo.

Apartado II: Bases de datos

Una gran parte de los proyectos software cuenta entre sus necesidades la del almacenamiento y recuperación de información, por regla general apoyándose en sistemas de bases de datos relacionales (RDBMS). Desde su primera versión Delphi ha puesto a disposición del desarrollador todo lo necesario para cubrir dicha necesidad, tanto en aplicaciones locales y cliente/servidor como en soluciones distribuidas. Los cuatro capítulos de esta segunda parte del libro están dedicados al estudio de las diferentes soluciones que ofrece Delphi XE2 en este campo.

- **Capítulo 4: Fundamentos del acceso a datos en Delphi XE2**
- **Capítulo 5: LiveBindings**
- **Capítulo 6: dbExpress**
- **Capítulo 7: DataSnap**

Capítulo 4: Fundamentos del acceso a datos en Delphi XE2

El objetivo de este capítulo es introducir los conceptos y elementos básicos del acceso a datos en Delphi: métodos disponibles, herramientas a nuestra disposición, componentes fundamentales, etc. No nos adentraremos aun, sin embargo, en los detalles de la conexión con bases de datos y otros aspectos que son específicos para cada caso concreto, tema al que se dedicarán los capítulos posteriores de esta sección del libro.

Aplicaciones y bases de datos

Las soluciones integradas en Delphi para el acceso a datos han ido evolucionando con el tiempo y adaptándose tanto a los cambios introducidos por otros fabricantes, a medida que han ido surgiendo nuevas versiones de servidores de datos y servicios, como a las necesidades de los desarrolladores que, a su vez, se adaptaban a los requerimientos de sus clientes.

Al diseñar un proyecto en el que se precisa almacenar/recuperar información hay que determinar qué uso se hará de ésta: cuántos usuarios precisan acceder a los datos, si lo harán de forma concurrente, si los datos serán modificados frecuentemente y se requiere respuesta en tiempo real (OLTP, *Online Transaction Processing*) o será principalmente estática y sobre ella se ejecutarán consultas complejas que pueden demorarse en el tiempo (*Data Warehousing/Data Mining*), etc.

En función de las respuestas a estas cuestiones se decidirá cómo ha de almacenarse esa información: localmente en el ordenador donde va a ser utilizada y sin mecanismos de seguridad (en un archivo con algún formato estándar como puede ser XML), en una base de datos local o en un servidor de bases de datos. La elección que se haga influirá, en un paso posterior, en la selección del mecanismo de acceso a datos que deba utilizarse en la aplicación.

Acceso a datos en Delphi XE2

Delphi XE2 cuenta con múltiples mecanismos de acceso a datos. Por una parte siguen ofreciéndose métodos obsoletos, como es el caso de BDE, a fin de mantener la compatibilidad con versiones previas del producto y facilitar la transición. Al mismo tiempo se incorporan otros métodos totalmente renovados como ocurre con dbExpress, un marco *(framework)* preparado para funcionar en múltiples plataformas y prácticamente con cualquier base de datos.

Al disponer de múltiples opciones es importante saber cuál debemos elegir en cada caso según las necesidades del proyecto que se pretenda abordar. La siguiente es una breve guía al respecto:

- **BDE** (*Borland Database Engine*): Este motor de bases de datos formaba parte de la primera versión de Delphi y fue evolucionando en versiones posteriores, contemplando tanto el acceso a bases de datos de escritorio, la más conocida de ellas dBase, como a servidores tipo RDBMS. No es recomendable su uso en nuevos proyectos ya que su desarrollo se abandonó hace tiempo, lo que significa que no habrá controladores actualizados para los RDBMS actuales ni tampoco soporte para características imprescindibles como es el uso de UNICODE.
- **IBX** (*InterBase Express*): Como su propio nombre indica se trata de un mecanismo específico para InterBase, un RDBMS a medio camino entre los grandes servidores de bases de datos de Oracle e IBM y las soluciones de escritorio. Solamente tiene sentido usar este método de acceso a datos si tenemos claro que la información siempre residirá en un servidor InterBase, un cambio a otro producto requeriría profundas modificaciones en nuestro proyecto.
- **dbGo:** Desde años Microsoft ha incorporado en las distintas versiones de Windows servicios de acceso a datos con el objetivo de que todas las aplicaciones los utilizasen en lugar de aportar soluciones propietarias. La encarnación del último intento en ese sentido fue ADO (*ActiveX Data Objects*) y OLE DB. La biblioteca dbGo de Delphi sirve para usar dichos servicios, lo cual aporta cierta independencia respecto a la base de datos usada pero, al tiempo, hace que las aplicaciones sean específicas para Windows.
- **MyBase:** Es una alternativa para el almacenamiento local de datos frente a otros métodos en desuso, como el mencionado BDE. Se fundamenta en el uso del componente `TClientDataSet` que estudiaremos después en este mismo capítulo.
- **dbExpress** (DBX): Introducido en Delphi 6, este marco de trabajo con bases de datos fue diseñado desde un principio como una alternativa ligera y multiplataforma frente a BDE. En Delphi XE2 la versión de dbExpress es la 4 y ha sido rediseñada para trabajar con un nuevo tipo de controlador, si bien se mantiene una buena compatibilidad hacia atrás que simplificará la transición de aplicaciones ya existentes. Hay controladores DBX para múltiples RDBMS tanto en Windows como en MacOS X, así como la posibilidad de acceder virtualmente a cualquier fuente de

información a través el nuevo controlador ODBC[44]. Éste es el método preferente de acceso a datos en nuevas aplicaciones desarrolladas con Delphi XE2.

- **DataSnap**: Más que un método de acceso a datos DataSnap es un *middleware*, un software que se interpone entre el servidor de datos y el cliente que ejecuta la interfaz de usuario, actuando a modo de servidor de aplicaciones. El servidor DataSnap recurre a dbExpress para acceder a la información, poniéndola a disposición del cliente mediante conectores que se comunican a través de TCP/IP y HTTP. Dichos conectores permiten que la interfaz se ejecute, además de en Windows o MacOS X, en dispositivos móviles con iOS, Android, Blackberry y Windows Phone.

Por las características mencionadas recomendaría usar dbExpress siempre que no haya requerimientos específicos que exijan una alternativa. Un caso podría ser, por ejemplo, que el RDBMS sea InterBase y se requiera un mayor control sobre el servidor de datos y aprovechar sus capacidades al máximo, lo cual justificaría el uso de IBX en lugar de dbExpress.

NOTA

Incluso cuando solamente se necesita un almacenamiento local de los datos es recomendable delegar la gestión de la información en un RDBMS, ya que ofrece mecanismos de seguridad e integridad de los datos y, además, facilitaría una posterior adaptación del proyecto a un entorno distinto.

Pueden utilizarse productos como Firebird Embedded[45] que integran el servidor de datos en una biblioteca que puede ser enlazada a la propia aplicación, no precisándose la instalación y configuración de un servidor independiente.

44 En versiones previas de Delphi no existía un controlador DBX nativo para ODBC, lo cual obligaba a usar otros métodos como interconexión, por ejemplo dbGo para acceder al controlador ODBC con que cuenta OLE DB.

45 Firebird es un RDBMS desarrollado como proyecto de código abierto (*open source*) a partir de InterBase. En http://www.firebirdsql.org/manual/ufb-cs-embedded.html se describe Firebird Embedded.

Estructura de una aplicación con acceso a datos

Las aplicaciones que van a trabajar con datos almacenados en algún tipo de almacén de información: un archivo simple, una base de datos local, un servidor de bases de datos o a través de un servidor de aplicaciones, tienen todas una estructura muy similar compuesta de los siguientes elementos:

- **Interfaz de usuario**: Los componentes que forman la interfaz de usuario deben tener la capacidad de mostrar los datos obteniéndolos de un almacén local y, por regla general, facilitar su edición. Si usamos la VCL tenemos a nuestra disposición un conjunto de componentes específicos para trabajar conectados a una fuente de datos. Con FMX los mismos componentes que usaríamos en cualquier interfaz, como los que utilizábamos en el segundo capítulo, pueden ser conectados mediante una técnica denominada LiveBindings[46].
- **Almacén local de datos**: Para que los componentes de interfaz puedan realizar su trabajo han de tener acceso a un almacén local (y normalmente temporal) que contenga los datos adecuados en cada momento. En la mayor parte de los casos dicho almacén será el componente `TClientDataSet` citado anteriormente y sobre el que volveremos después.
- **Conexión con la fuente de información**: Los datos se transferirán desde su origen al almacén local de la aplicación, y viceversa, a través de uno o más componentes que serán los encargados de leer los resultados de las consultas, convertir las modificaciones en las sentencias que correspondan y, en general, de toda la comunicación con el software sobre el que recaiga la responsabilidad de gestionar los datos de manera permanente.

Es en este tercer punto, la conexión con la fuente de información, donde residen la mayor parte de las diferencias entre aplicaciones. En los casos más simples puede prescindirse por completo de dicha conexión, mientras que en otros puede implicar la comunicación con varios servidores.

---

46 LiveBindings es una nueva característica de Delphi XE2 y puede ser usada tanto con componentes VCL como FMX. La estudiaremos detalladamente en un capítulo posterior.

Aunque no se trata de una taxonomía formal, cuando se tratan proyectos que tienen necesidades de acceso a datos se las suele clasificar en las siguientes arquitecturas:

- **Local**: El programa que actúa como interfaz de usuario y los datos residen en la misma máquina, por lo que no hay necesidad de comunicaciones a través de red. La gestión de los datos puede asumirla el propio programa, prescindiendo del elemento de conexión indicado antes, o bien dejarla en manos de un software especializado como puede ser un RDBMS embebido o local. En este caso será necesario recurrir a los elementos habituales de conexión: controlador dbExpress para el RDBMS y software cliente de éste, pero la comunicación entre ambas partes puede realizarse por medios más eficientes.
- **Cliente/Servidor**: Es posiblemente la arquitectura más común. Se caracteriza porque los datos se alojan en una máquina distinta a la que utiliza el usuario para ejecutar la aplicación, lo cual facilita que la información esté centralizada y accesible para múltiples usuarios (ya sea desde la misma aplicación u otro software). En el ordenador de cada usuario se precisarán como elementos de conexión el controlador dbExpress y el software cliente del RDBMS que se haya elegido, RDBMS que estará ejecutándose (la parte servidor) en el servidor de datos. Entre cliente y servidor se establecerá una conversación, normalmente a través de TCP/IP, que permitirá al primero enviar comandos al segundo y a éste devolver los resultados de su ejecución, principalmente datos solicitados.
- **Distribuida**: La casi omnipresencia de redes de comunicaciones y difusión de todo tipo de dispositivos con capacidad de computación, especialmente los móviles *inteligentes,* tabletas y portátiles, ha hecho que la arquitectura distribuida, antes casi restringida a entornos de grandes empresas, experimente un gran crecimiento en los últimos años. En esta arquitectura el cliente ejecuta únicamente una interfaz de usuario, que puede ser nativa o bien estar basada en tecnologías web, mediante la cual el usuario accede a los servicios que ofrece un servidor de aplicaciones. Éste, a su vez, depende de un servidor de datos para la gestión de la información. Es el tipo de configuración en que recurriremos a DataSnap y sus conectores. El servidor DataSnap se ejecutará en el servidor de aplicaciones y conectará con el de datos con dbExpress.

RDBMS, controladores y componentes

Al iniciar el desarrollo de una aplicación de bases de datos hemos de tener en cuenta que, además de Delphi, deberemos disponer de otros elementos fundamentales. El primero de ellos, salvo en el caso más simple de una aplicación con arquitectura local y que use como método de almacenamiento un archivo, será el RDBMS y el software cliente asociado.

En el mercado hay disponibles multitud de productos RDBMS, desde los dirigidos históricamente a grandes corporaciones como son Oracle, IBM DB2 o Informix, hasta los basados en código abierto como MySQL o Firebird, pasando por otros de carácter comercial pero menos ambiciosos[47] que los primeros como pueden ser SQL Server o InterBase. Cada uno de ellos ofrece características diferenciadoras respecto a los demás y, por supuesto, herramientas de administración y programación específicas. Podríamos decir que cada RDBMS es un mundo y requerirá por nuestra parte un cierto nivel de conocimientos que dependerá de que solamente vayamos a usarlo, como programadores, o bien también asumamos el papel de administradores.

Además del software servidor, que será el encargado en último término de gestionar los datos y responder a las solicitudes que se le hagan tanto de forma local como remota, cada RDBMS cuenta también con un componente al que se conoce como software cliente. Como su propio nombre indica, se trata del software que necesitan los ordenadores cliente para poder comunicarse con el servidor. En éste suele instalarse todo el paquete: servidor de datos, software cliente y herramientas de administración, mientras que en los ordenadores donde va a ejecutarse la aplicación solamente se necesita el software cliente. En muchos casos éste se reduce a una biblioteca de código, como puede ser una DLL en Windows.

A fin de que el desarrollador pueda emplear en sus proyectos siempre los mismos componentes, con independencia del RDBMS en que se almacene la información, es necesario colocar un intermediario entre el software cliente del RDBMS y los componentes tanto de interfaz como de almacenamiento

47 En realidad los campos de aplicación de RDBMS como Oracle, SQL Server, InterBase o incluso MySQL y Firebird llegan a solaparse en muchos casos, debido a la comercialización de versiones reducidas de los más potentes y a la incorporación de características cada vez más avanzadas en los que inicialmente se dirigían a pequeñas empresas.

local de los datos. Ese intermediario es el controlador de bases de datos, un elemento que tiene dos caras: una específica para conectar con el software cliente de un RDBMS concreto y otra genérica e idéntica a la del resto de controladores de bases de datos.

Finalmente lo que tenemos es una cadena de componentes comunicándose entre sí en ambos sentidos: la interfaz usa componentes independientes del RDBMS para gestionar localmente los datos, esos componentes se vinculan a un controlador que facilita la conexión con el software cliente, ya específico de cada base de datos, y ese software será el que se comunique con el servidor de datos.

ADVERTENCIA

Los componentes necesarios para desarrollar aplicaciones de bases de datos, entre ellos los controladores dbExpress, no están disponibles en todas las ediciones de Delphi XE2. Si queremos conectar con grandes RDBMS (Oracle, DB2, SQL Server) en aplicaciones cliente/servidor o distribuidas precisaremos la edición Enterprise o superior.

Herramientas de bases de datos en el entorno

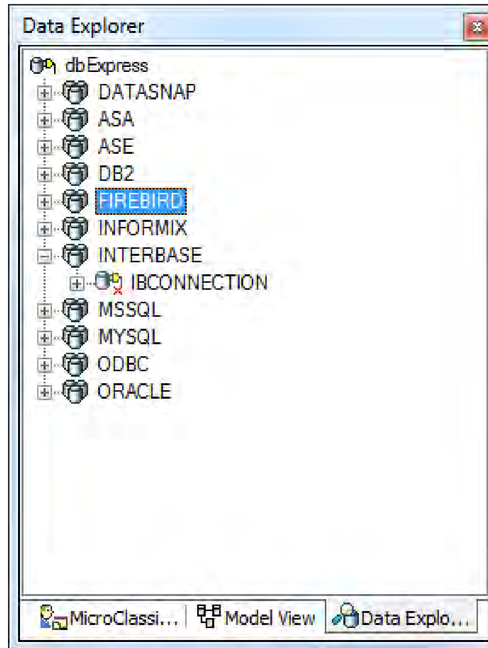
Además de los controladores y componentes dbExpress, dbGo, IBX, etc., Delphi XE2 también integra en el entorno herramientas que facilitarán nuestro trabajo a la hora de desarrollar aplicaciones con acceso a bases de datos.

Por una parte tenemos el panel Data Explorer, situado inicialmente en el margen derecho del entorno como una página de la ventana en la que se encuentra también el Gestor de proyectos. Nos servirá para definir conexiones a bases de datos y manipular tanto su estructura como su contenido.

Por otra están los módulos de datos que, en conjunción con el panel Structure y el Inspector de objetos, nos permitirán preparar los elementos necesarios para acceder a los datos y trabajar sobre ellos.

El Explorador de datos

Esta ventana (véase la imagen inferior) muestra un árbol en el que la raíz es un nodo, llamado dbExpress, del que cuelgan tantos nodos distintos como tipos de bases de datos está preparado Delphi para usar. Dependiendo de la edición del producto tendremos más o menos controladores dbExpress y, en consecuencia, dicha lista será más o menos extensa.



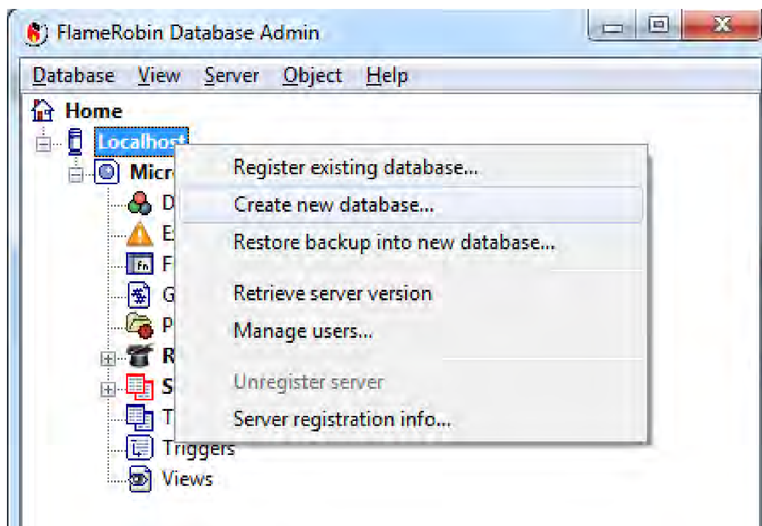
El nodo correspondiente a cada tipo de base de datos contendrá *conexiones*, nombre que reciben los conjuntos de parámetros específicos para acceder a una base de datos concreta. Inicialmente hay una conexión predefinida de cada tipo que nos permite conocer cuáles son los valores habituales para cada parámetro. En la práctica, sin embargo, cada vez que vayamos a conectar con una base de datos definiremos una nueva conexión a medida.

A través de una conexión podemos acceder a información de estructura de la base de datos, obteniendo una lista de sus tablas, vistas, procedimientos almacenados, etc. Para cada uno de esos objetos se ofrecen opciones específicas, por ejemplo para las tablas: información sobre las columnas, acceso a los contenido de la tabla, modificación de la estructura de ésta, etc.

Veamos cuáles serían los pasos para definir una conexión que nos permita acceder a una base de datos, partiendo de que cumplimos las siguientes premisas:

- Tenemos el RDBMS instalado, ya sea en un servidor o en el mismo equipo de desarrollo.
- Hemos creado la base de datos a través de las herramientas específicas de dicho RDBMS.
- Hemos instalado en el equipo de desarrollo el software cliente[48] del RDBMS.

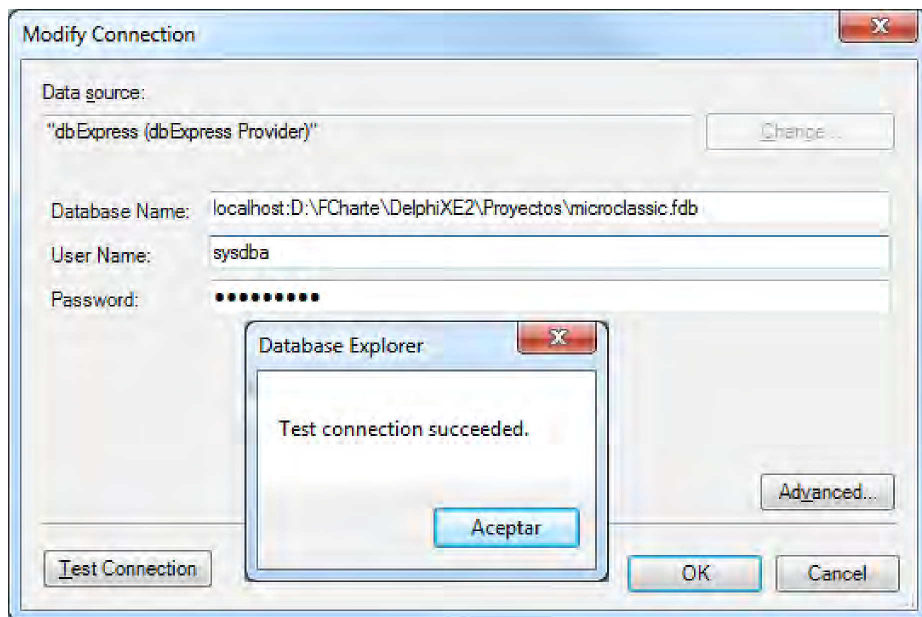
En nuestro caso hemos instalado en el equipo de desarrollo la última versión de Firebird, así como la herramienta de administración de código abierto FlameRobin[49] desde la que conectamos con el servidor Firebird local y creamos una base de datos como puede verse en la imagen inferior. Ésta en principio estará vacía, no conteniendo ningún objeto.



48 Delphi incluye controladores dbExpress que permiten comunicarse con el RDBMS a través de dicho software cliente, pero éste forma parte del producto de bases de datos, no de Delphi.

49 Esta herramienta está disponible en http://flamerobin.org/ para Linux, Windows, MacOS X y otros sistemas operativos.

Para conectar con esta base de datos desde el entorno de Delphi seleccionamos el nodo FIREBIRD del Explorador de datos, abrimos el menú contextual y seleccionamos la opción Add New Connection, facilitando un nombre para la nueva conexión. Acto seguido abrimos de nuevo el menú y usamos la opción Modify Connection para abrir el cuadro de diálogo del mismo nombre. En éste facilitaremos el nombre de la base de datos que, en este caso concreto, tendrá el formato `usuario@servidor:ruta/nombrebd`. Además hay que entregar las credenciales con las que se conectará a la base de datos: nombre de usuario y contraseña. En Firebird por defecto siempre existe un usuario llamado `sysdba` con contraseña `masterkey`. Establecidos estos tres parámetros un clic sobre el botón Test Connection nos confirmará si la conexión es posible o no (véase imagen inferior).

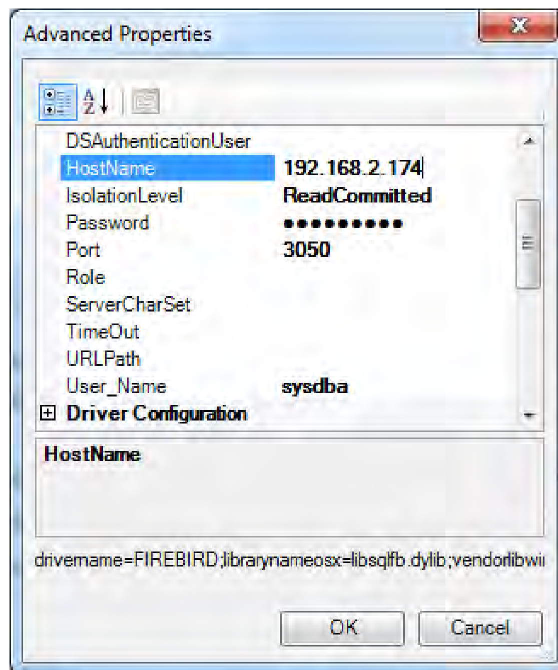


ADVERTENCIA

Si habiendo instalado correctamente el RDBMS y configurado la conexión ésta falla, debemos asegurarnos que la biblioteca cliente del RDBMS ha sido copiada en las carpetas correctas. En el caso de Firebird, por ejemplo,

> el proceso de instalación no copia el módulo fbclient.dll desde Archivos de programa\Firebird\Firebird_2_5 a \Windows\System32 (o Windows\SysWOW64, dependiendo del sistema), por lo que es necesario copiarlo manualmente. Tras esta operación la conexión funcionará correctamente.

Para establecer una conexión con un servidor remoto nos será más cómodo fijar todos los parámetros mediante la ventana a la que da paso el botón Advanced del cuadro de diálogo anterior. Como puede verse en la imagen inferior, la propiedad HostName especifica el nombre o dirección IP del servidor de datos. Opcionalmente, en caso de que hayamos configurado el RDBMS para que opere en un puerto distinto al que se usa por defecto, indicaremos el nuevo puerto en la propiedad Port.



Una vez configurada y comprobada la conexión, en el Explorador de datos podremos desplegar el nodo asociado a ésta y ver cómo aparecen los nodos Tables, Views, Procedures, Functions y Synonyms, inicialmente vacíos.

Manipulación de la información de esquema de una base de datos

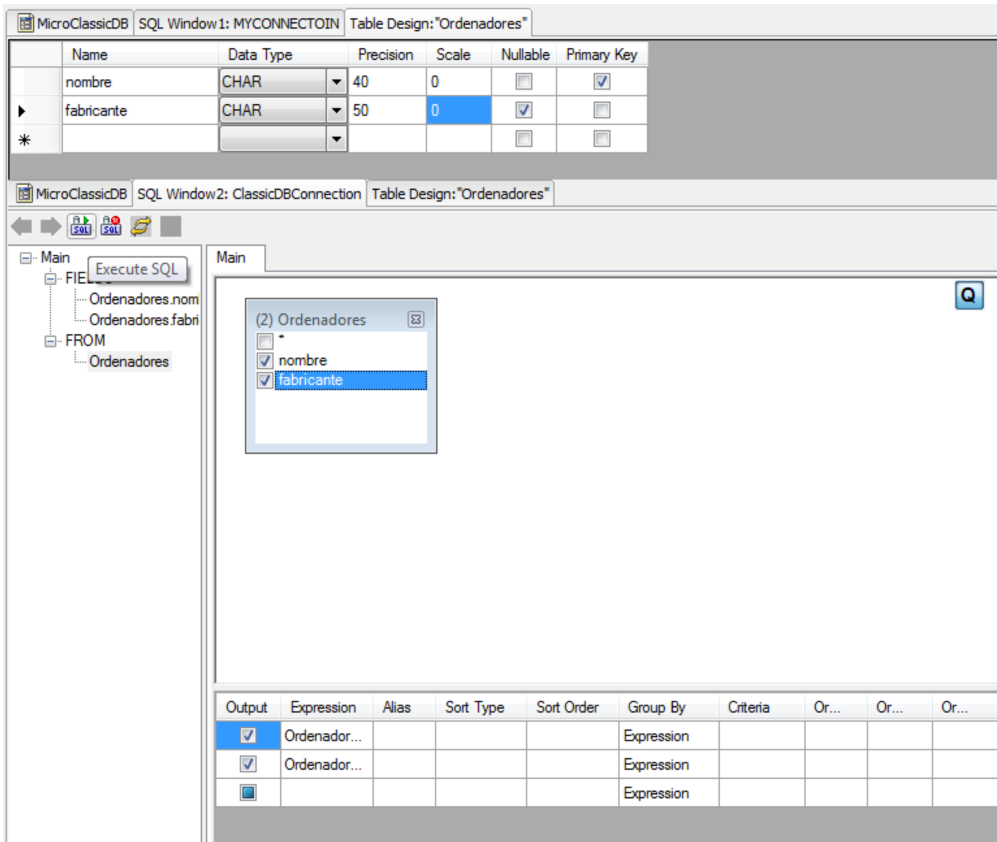
En caso de que recaiga en nosotros la responsabilidad de administrar la base de datos asociada al proyecto, deberemos contar con unos conocimientos sólidos de teoría del diseño de bases de datos, reglas de normalización, lenguaje SQL, etc., aspectos todos éstos en los que no vamos a entrar aquí.

En último término todo nuestro trabajo se traducirá en la creación, modificación y eliminación de objetos en la base de datos, manipulando la información de esquema que regirá la estructura de almacenamiento de los datos y posiblemente también los métodos de acceso y modificación de éstos. Esas operaciones pueden realizarse desde el software de administración propio del RDBMS o bien con las opciones del propio Explorador de datos de Delphi.

Normalmente el primer paso será la creación de las tablas en las que se almacenará la información. El menú contextual del nodo Tables (en el Explorador de datos) cuenta con la opción New Table que da paso a un diseñador (véase parte superior de la imagen en página siguiente) desde el que, por cada columna de la tabla, se configura el nombre, tipo, precisión y las dos restricciones más simples: permitir o no valores nulos y columnas que forman la clave primaria.

Si necesitamos un mayor control sobre los atributos de la tabla a crear, por ejemplo asociando restricciones de tipo CHECK o de integridad referencial, podemos recurrir al panel SQL Window. La opción del mismo nombre que lo abre se encuentra en el menú contextual de la base de datos. Como se aprecia en la parte inferior de la imagen de la página siguiente, esta ventana facilita la configuración de consultas SQL de manera visual. Podemos, no obstante, escribir y ejecutar cualquier tipo de sentencia SQL, ya sea de definición de datos (DDL, *Data Definition Language*) o de manipulación de éstos (DML, *Data Manipulation Language*).

NOTA Aunque en el panel inferior de la SQL Window puede introducirse cualquier tipo de sentencia SQL, el editor no ofrece ningún tipo de asistencia ni ayuda. Por ello, en general, es mejor usar las herramientas de administración que ofrezca el propio RDBMS.



Manipulación interactiva de los datos

A pesar de que quizá no sea la herramienta con la interfaz más amigable y elaborada (hay que pensar que está dirigida no a usuarios finales sino a desarrolladores que tienen una cierta experiencia), desde el Explorador de datos también es posible acceder al contenido de las tablas, es decir, los datos propiamente dichos.

Para ello no hay más que abrir el menú contextual de la tabla en cuestión y elegir la opción Retrieve Data From Table. Se abrirá una nueva página en el área central del entorno con una cuadrícula o rejilla de datos, pudiendo tanto modificar los datos existentes como agregar otros nuevos.

En realidad esta opción hay que verla más como una herramienta que nos servirá para comprobar con inmediatez el contenido de una tabla, por ejemplo a fin de verificar si la ejecución de una función del proyecto realiza correctamente su trabajo, que como una utilidad general de inserción y edición de datos.

Módulos de datos

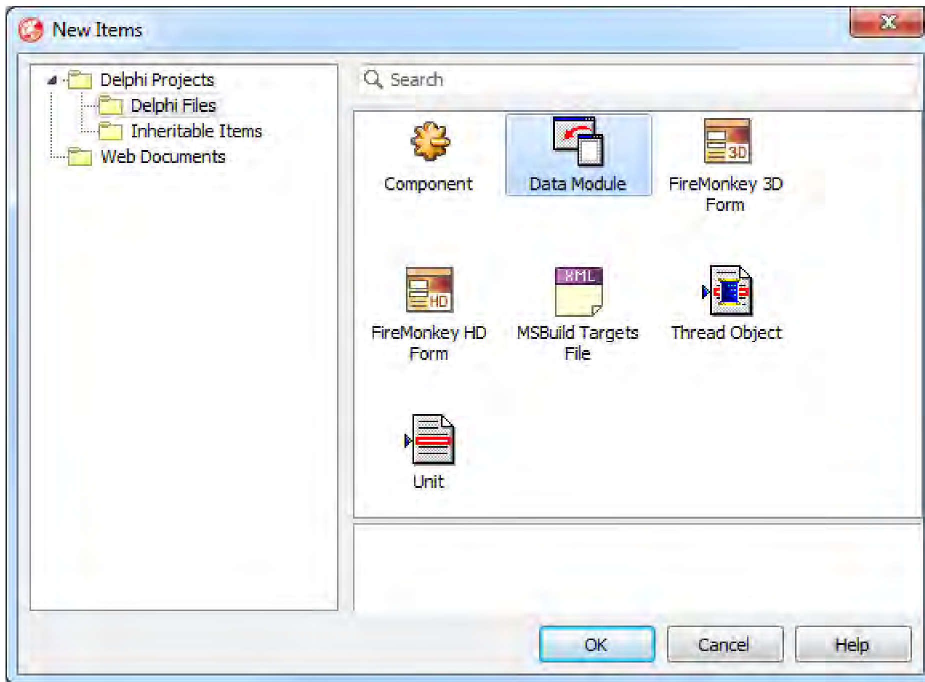
Sirviéndonos del Explorador de datos podemos examinar las bases de datos que usaremos en un proyecto y, con las opciones que se han descrito, conocer todo lo necesario sobre su estructura y contenido, incluso modificándolo si fuese necesario.

Para que la aplicación pueda acceder a esos datos y trabajar con ellos, sin embargo, tendremos que agregar una serie de componentes que se encargarán de establecer la conexión, ejecutar las sentencias SQL, recuperar los datos que resulten de las consultas, etc. Dichos componentes se encuentran en las páginas dbExpress, Data Access, InterBase, dbGo y DataSnap de la Paleta de herramientas.

Esos componentes pueden introducirse prácticamente en cualquier tipo de contenedor, entre ellos un formulario VCL o FMX. En un proyecto real es habitual que se necesite acceder a los datos desde diferentes funciones de la aplicación, lo que normalmente implicaría agregar y configurar los mismos componentes varias veces. A fin de evitar esa duplicidad Delphi cuenta un tipo de contenedor específico denominado Módulo de datos o Data Module.

Al igual que un formulario se define como una clase derivada de `TForm`, la clase base para todos los formularios en su versión VCL o FMX según la biblioteca de componentes usada, un módulo de datos se definirá como una clase derivada de `TDataModule`. Ésta se encuentra definida en el módulo `System.Classes` y puede ser usada con independencia de que el proyecto esté basado en la VCL o en la FMX.

Para agregar un módulo de datos al proyecto usaremos la opción File>New>Other y elegiremos el elemento Data Module de la ventana New Items, como puede verse en la imagen de la página siguiente. El módulo de datos aparecerá en el diseñador como una superficie en blanco y en el Inspector de objetos podremos ver que únicamente cuenta con cuatro propiedades.



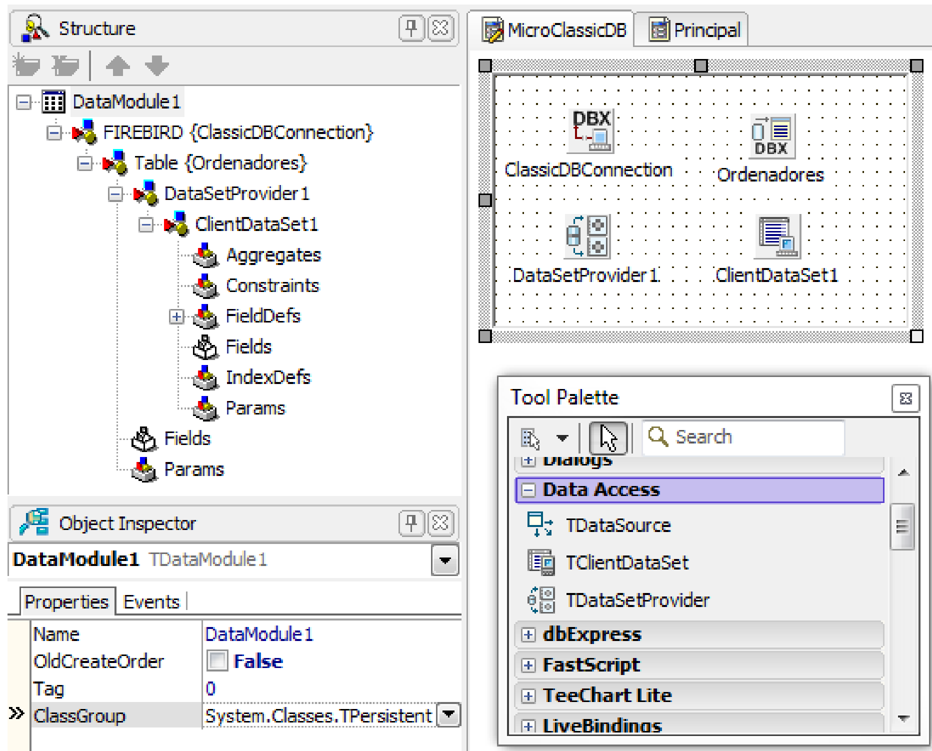
De esas propiedades `Name` nos permite asignar un nombre al módulo de datos, `Tag` asociarle un valor cualquiera. La propiedad realmente interesante es `ClassGroup`, ya que determina los componentes que podremos incluir en el módulo de datos. Si desplegamos la lista asociada a dicha propiedad comprobaremos que hay cuatro opciones:

- `System.Classes.Persistent`: El módulo de datos solamente podrá contener componentes multiplataforma.
- `Vcl.Controls.TControl`: Además de los multiplataforma también se permitirán componentes específicos de la VCL.
- `FMX.Types.TControl`: Además de los multiplataforma también se permitirán componentes específicos de la FMX.
- `FMX_Types.TControl`: Permitirá también el uso de componentes FMX para iOS, aparte de los multiplataforma.

Al modificar el valor de esta propiedad comprobaremos como las páginas y componentes disponibles en la Paleta de herramientas cambia, mostrando unos grupos u otros.

> ADVERTENCIA
>
> Es posible que la Paleta de herramientas no se actualice de manera inmediata. No hay más que cambiar en el diseñador desde el Módulo de datos a otra página y volver. Es un fallo del entorno que posiblemente se resuelva en una actualización futura.

Los componentes de un módulo de datos pueden agregarse manualmente, tomándolos por ejemplo de la página Data Access de la Paleta de herramientas, o bien de manera automática al arrastrar un objeto desde el Explorador de datos. Si tomamos la tabla de una base de datos y la llevamos hasta el Módulo de datos veremos que se agregan dos componentes: un `TSQLConnection` y un `TSQLDataSet`. En la imagen inferior puede verse cómo la ventana Structure nos será también de ayuda en este caso, al mostrar la relación entre todos los elementos alojados en el módulo de datos.



Una vez que hayamos configurado en nuestro módulo de datos todos los elementos necesarios para acceder a la base de datos y trabajar con ella cómodamente no tendremos que volver a repetir esta operación. Desde cualquier formulario de la aplicación podremos agregar una referencia al módulo de datos, sin más, y tendremos a disposición toda su funcionalidad.

NOTA

Aparte de configurar una sola vez los componentes de acceso a datos, con independencia del número de formularios desde el que se acceda a ellos, el uso de un módulo de datos también es útil para compartir código específico para el tratamiento de esos datos. Si asociamos una cierta operación a un evento de un componente en un módulo de datos, por ejemplo al actualizar el contenido de una tabla, dicho código estará centralizado en ese punto y se ejecutará siempre, sin importar el punto de la aplicación que desencadene el evento. Esto también simplifica el mantenimiento de ese código, al existir una sola copia del mismo centralizada en el módulo de datos.

MyBase

Los componentes que tendremos que emplear a la hora de desarrollar una aplicación con acceso a datos, dejando de lado los usados para componer la interfaz de usuario, pueden agruparse en dos categorías: por una parte los dependientes del origen de datos y por otra parte aquellos que son independientes de dicho origen. En el primer grupo podremos optar entre los ya citados dbExpress, dbGo, IBX, etc., mientras que en el segundo siempre encontraremos los mismos elementos. Éstos, además, son los pilares de uno de los métodos de acceso a datos de Delphi: MyBase.

Como se indicó con anterioridad, MyBase permite a una aplicación almacenar datos localmente sin necesidad de instalar ningún software adicional ni conectar con un RDBMS. El componente fundamental en este contexto es `TClientDataSet`.

Como veremos en capítulos posteriores, se trata de un componente que también precisaremos al desarrollar proyectos con arquitecturas cliente/servidor o distribuida. En esta última sección del presente capítulo estudiaremos el desarrollo de una aplicación MyBase con el fin de alcanzar dos objetivos: 1) conocer una de las opciones de almacenamiento local que ofrece Delphi y 2) describir los conceptos asociados a componentes que son independientes del RDBMS y que nos serán útiles en cualquier tipo de proyecto con acceso a datos.

El componente TClientDataSet

Encontraremos este componente en el grupo Data Access de la Paleta de herramientas, junto a otros que son independientes del origen de datos. `TClientDataSet` está definido en el módulo `DataSnap.DBClient` que no pertenece ni a la VCL ni a la FMX, pudiendo utilizarse en aplicaciones que usan cualquiera de esas bibliotecas para componer la interfaz de usuario.

`TClientDataSet` es una clase que deriva de `TDataSet`[50]. Ésta representa un conjunto de datos genérico, compuesto por una serie de campos (o columnas), índices y otros parámetros. Un `TClientDataSet` aparte de la información de estructura el objeto mantiene en memoria los datos correspondientes: las filas con el contenido de cada una de las columnas. Asimismo hereda de `TDataSet` un importante conjunto de métodos y eventos que, por ejemplo, facilitan la navegación por los datos almacenados, el agregado de nuevos datos, la eliminación de éstos, etc.

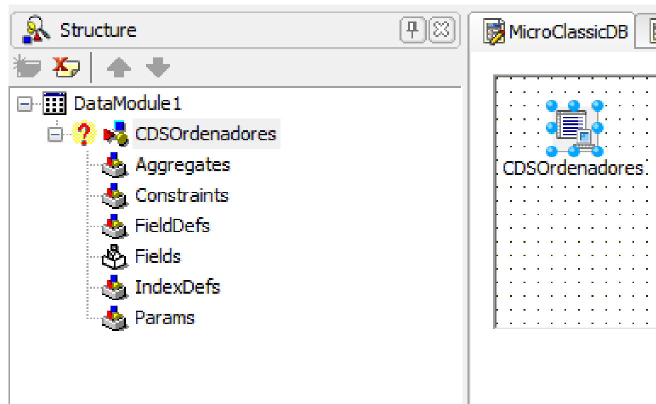
La diferencia fundamental entre `TClientDataSet` y otros derivados de `TDataSet` es que la información, tanto de estructura como datos, que contiene no es directamente obtenida de una conexión a una base de datos. En su lugar se recurre a un proveedor (como veremos en un capítulo posterior) o bien el almacenamiento/recuperación de un archivo local gracias a los métodos `LoadFromFile` y `SaveToFile`.

Aunque no es una exigencia lo habitual es que el formato de dicho archivo sea XML (*eXtensible Markup Language*), un estándar para el intercambio de datos aceptado desde hace años por parte de la industria del software.

---

50 Hay muchas otras clases derivadas directa o indirectamente de `TDataSet`, entre ellas `SqlExpr.TSQLTable`, `SqlExpr.TSQLQuery` y `SqlExpr.TSQLStoredProc`.

Si insertamos en un módulo de datos (o un formulario) un componente TClientDataSet observaremos que en la ventana Structure (véase imagen inferior) aparece como un nodo conteniendo seis hojas llamadas Aggregates, Constraints, FieldDefs, Fields, IndexDefs y Params. Cada una de ellas es una propiedad de TClientDataSet que representa una colección de objetos: TAggregate para obtener totales y subtotales de los datos; TCheckConstraints con condiciones que restringen los valores que pueden ser asignados; TFieldDef que almacenan la definición de las columnas del conjunto de datos; TField con las columnas que no son campos agregados, TIndexDef que almacenan la definición de los índices y, finalmente, TParam para facilitar parámetros de inicialización o ejecución.



Cuando se usa un TClientDataSet como almacenamiento temporal de información procedente de una base de datos, según el esquema que conoceremos posteriormente, la mayor parte de las propiedades anteriores toman sus valores automáticamente, a partir de la información de estructura obtenida de la propia base de datos mediante una llamada al método FetchParams[51].

En una aplicación MyBase, sin embargo, los atributos de cada una de las columnas, restricciones, índices y resto de parámetros habrán de configurarse manualmente en algún momento, tras lo cual se almacenarán y recuperarán de un archivo local.

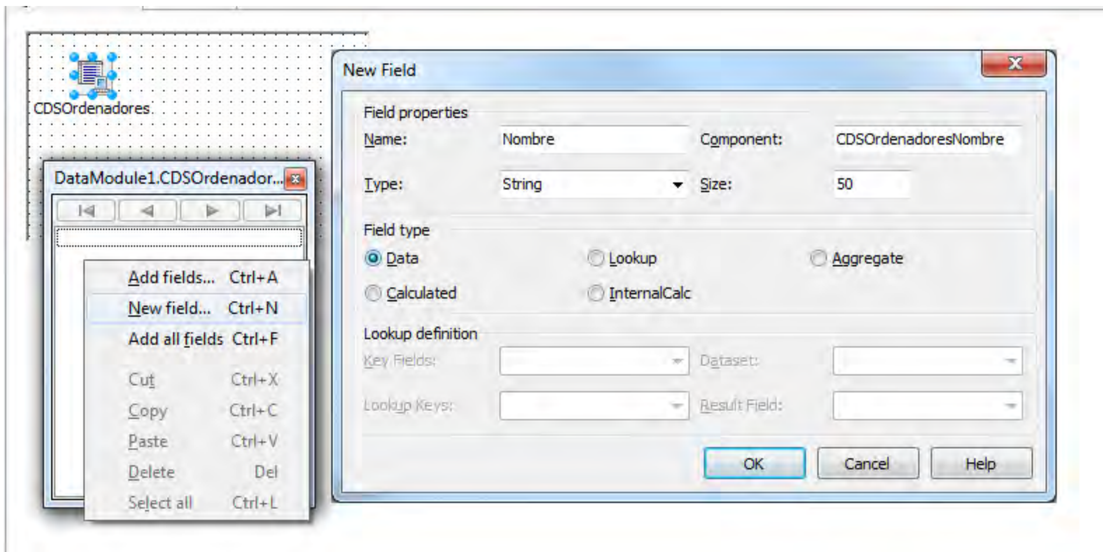
---

51 Es posible invocar a dicho método desde el propio diseñador, abriendo el menú contextual del TClientDataSet y eligiendo la opción Fetch Params. Sin contar con una conexión a bases de datos, sin embargo, únicamente obtendríamos un error.

Definición interactiva de columnas

En el desarrollo de una aplicación MyBase tras agregar un componente `TClientDataSet` al módulo de datos, o directamente en el formulario donde se necesite, el primer paso a dar será la definición de las columnas de datos. Para ello tendremos que recurrir al editor de campos, ya sea con un doble clic sobre el `TClientDataSet` o mediante la opción Fields Editor de su menú contextual.

Vacío en un principio, el editor de campos es una ventana que cuenta con cuatro botones, en su parte superior, que servirán para navegar por los datos, y un menú contextual cuyas tres primeras opciones permiten agregar definiciones de columnas, ya sea recuperando información sobre ellas de un origen de datos o bien manualmente: con la opción New Field. Al elegirla se abrirá el cuadro de diálogo del mismo nombre que puede verse en la imagen inferior.



Por cada columna del conjunto de datos a definir habrá que establecer básicamente tres parámetros: su nombre, el tipo de información que contendrá y, si procede, su tamaño. El resto de la configuración toma valores por defecto: nombre del componente asociado a la columna y tipo de campo. Al hacer clic en OK la nueva columna aparecerá en el editor de campos y podremos repetir la operación, añadiendo las columnas que se precisen.

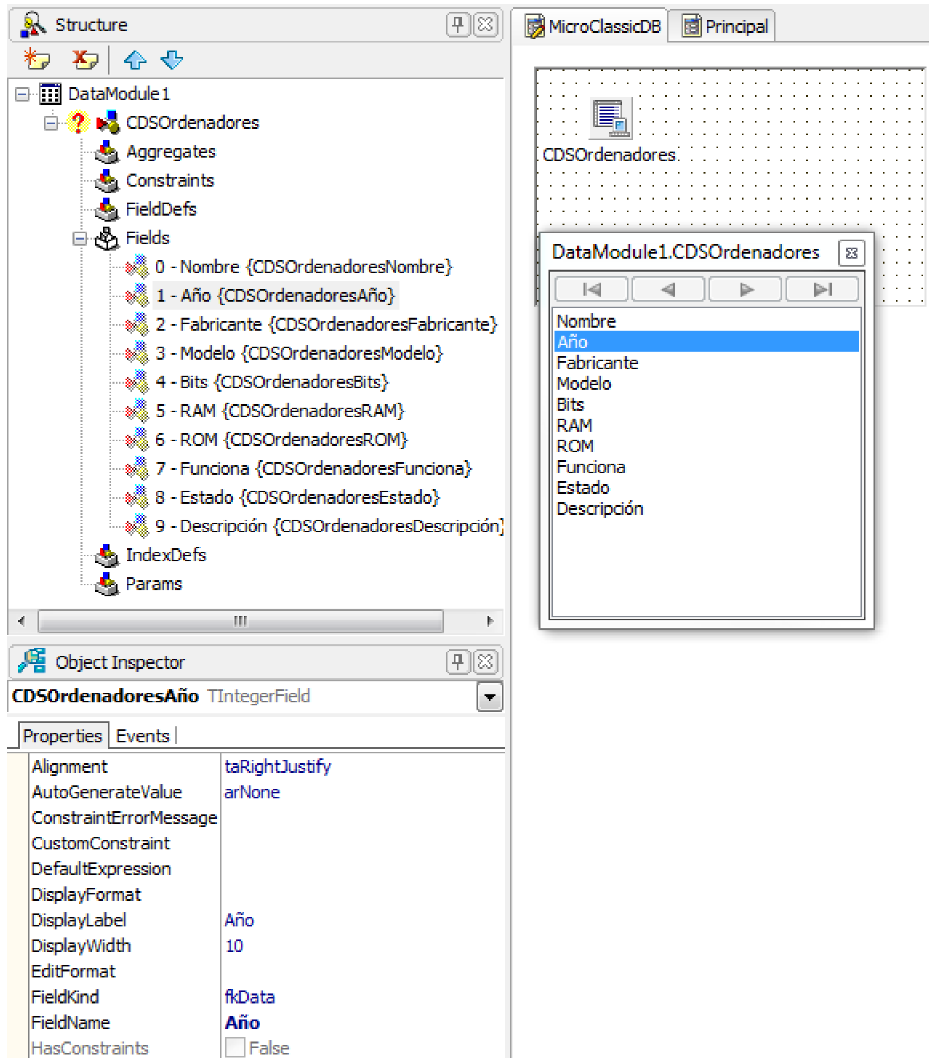
El tipo de campo (apartado Field Type del cuadro de diálogo) por defecto es Data indicando que la columna contendrá un dato. El tipo de dato se establece con la lista desplegable Type, donde encontraremos una extensa variedad: desde números enteros y en punto flotante, booleanos y cadenas hasta tipos que pueden almacenar un gráfico, una matriz o incluso otro conjunto de datos.

Además de columnas que contienen datos también es posible agregar columnas calculadas: obtienen su contenido a partir de la evaluación de una expresión; columnas con agregaciones: subtotales o totales de los contenidos de otras columnas, y columnas de búsqueda (tipo Lookup) cuya finalidad es ofrecer como posibles valores los datos alojados en otra tabla.

Cada columna añadida tendrá asociado un objeto de una clase derivada de `TField` y aparecerá como tal en la rama Fields de la ventana Structure. Seleccionando cualquiera de ellos podremos ver en el Inspector de objetos (véase la imagen de la página siguiente) el tipo concreto: `TStringField` para las columnas de tipo `String`, `TIntegerField` para las de tipo `Integer`, etc. En el mismo Inspector de objetos podremos acceder a todas las propiedades del campo seleccionado, lo cual nos permite personalizarlo como cualquier otro componente.

Si accedemos al código del módulo de datos (pulsando F12) comprobaremos que los componentes asociados a cada columna aparecen como atributos del módulo de datos:

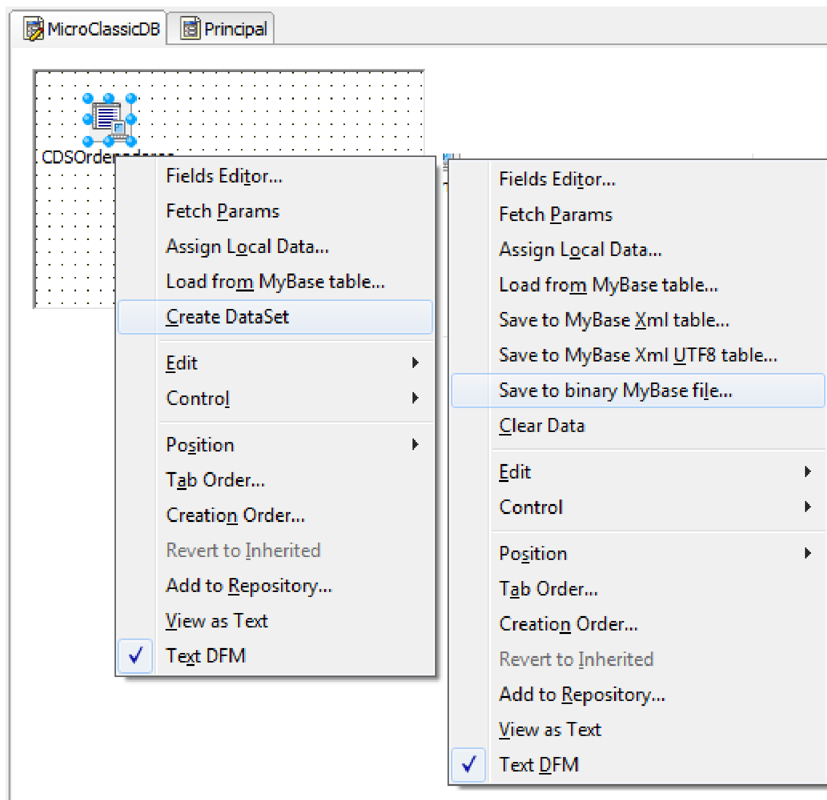
```
type
  TDataModule1 = class(TDataModule)
    CDSOrdenadores: TClientDataSet;
    CDSOrdenadoresNombre: TStringField;
    CDSOrdenadoresAño: TIntegerField;
    CDSOrdenadoresFabricante: TStringField;
    CDSOrdenadoresModelo: TStringField;
    CDSOrdenadoresBits: TSmallintField;
    CDSOrdenadoresRAM: TSmallintField;
    CDSOrdenadoresROM: TSmallintField;
    CDSOrdenadoresFunciona: TBooleanField;
    CDSOrdenadoresEstado: TSmallintField;
    CDSOrdenadoresDescripción: TMemoField;
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```



> NOTA
>
> Las propiedades de cada uno de los componentes que representan a las columnas se encuentran almacenadas en el DFM, como es habitual. Con la combinación de teclas Alt-F12 puede accederse a la descripción textual, volviéndose al diseñador con el mismo atajo de teclado.

Crear y almacenar el dataset

Habiendo definido las columnas que precise la aplicación llega el momento de generar el conjunto de datos en memoria, usando para ello la opción Create DataSet del menú contextual del componente `TClientDataSet`. Aparentemente no ocurrirá nada, pero si observamos la ventana Structure veremos que en la rama `FieldDefs` aparece la definición de cada una de las columnas. La mayor diferencia, no obstante, la encontraremos en el propio menú contextual que ofrecerá un conjunto de opciones distinto. En la imagen inferior puede verse el menú antes (izquierda) y después de haber creado el conjunto de datos.



Internamente el `TClientDataSet` ya dispone de las estructuras necesarias para almacenar los datos, sin importar si estos proceden de una base de datos o no.

El paso siguiente será guardar el conjunto de datos de manera persistente, en un archivo en disco, a fin de que la aplicación pueda recuperarlo al inicio de cada ejecución. En el mismo menú contextual encontramos tres opciones Save to … que se diferencian únicamente en el formato que emplearán para almacenar la información. En nuestro caso usaremos Save to MyBase Xml UTF8 table, facilitaremos el nombre para el archivo y ejecutaremos la operación.

Si abrimos el archivo, directamente en el propio editor de Delphi, su contenido debería ser similar al siguiente:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
 <DATAPACKET Version="2.0">
  <METADATA>
   <FIELDS>
    <FIELD attrname="Nombre" fieldtype="string"
                                    WIDTH="50"/>
    <FIELD attrname="Año" fieldtype="i4"/>
    <FIELD attrname="Fabricante" fieldtype="string"
                                    WIDTH="50"/>
    <FIELD attrname="Modelo" fieldtype="string"
                                    WIDTH="50"/>
    <FIELD attrname="Bits" fieldtype="i2"/>
    <FIELD attrname="RAM" fieldtype="i2"/>
    <FIELD attrname="ROM" fieldtype="i2"/>
    <FIELD attrname="Funciona" fieldtype="boolean"/>
    <FIELD attrname="Estado" fieldtype="i2"/>
    <FIELD attrname="Descripción" fieldtype="bin.hex"
                                SUBTYPE="Text"/>
   </FIELDS>
   <PARAMS/>
  </METADATA>
  <ROWDATA>
  </ROWDATA>
 </DATAPACKET>
```

El archivo se divide en dos secciones: METADATA y ROWDATA. La primera contiene la definición de las columnas de datos: nombre, tipo y longitud. En la segunda, vacía por el momento, se almacenarán los datos propiamente dichos: el valor que toma cada una de esas columnas en cada fila de la tabla.

ADVERTENCIA
Ha de tenerse en cuenta que lo que se almacena en el archivo XML es la información de esquema de los datos y los propios datos, pero no la configuración de los componentes asociados que reside en el DFM.

Integrar MyBase en la aplicación

Tras completar los pasos previos el componente TClientDataSet ya está preparado para almacenar toda la información, en este caso relativa a los ordenadores de la colección, y gestionar su manipulación. Nos falta integrar el módulo de datos con el resto de la aplicación, una tarea que completaremos dando los siguientes pasos:

- Asignaremos a la propiedad FileName del TClientDataSet el nombre del archivo en el que se guarda la información. No tenemos más que hacer clic en el botón que aparece en el margen derecho de dicha propiedad y seleccionar el archivo donde habíamos guardado los datos en el punto previo.
- Necesitamos que el TClientDataSet recupere automáticamente los datos cada vez que se ejecute la aplicación. Para ello basta con invocar a su método Open desde el evento OnCreate del propio módulo de datos[52].
- Asimismo es preciso que los datos se guarden en el mismo archivo al cerrar la aplicación, momento en el que se generará el evento OnDestroy del módulo de datos. Desde el método asociado a dicho evento llamaremos al método SaveToFile del TClientDataSet.
- Por cada uno de los formularios del proyecto desde el que se precise acceso a la información habrá que agregar una referencia al módulo de datos, ya sea manualmente modificando la cláusula uses o bien mediante la opción File>Use Unit.
- Finalmente se establecerá la vinculación entre los componentes de la interfaz y los componentes de datos, habilitando tanto la visualización como la edición. En este aspecto tenemos dos alternativas si la aplicación usa la VCL y solamente una si está basada en la FMX: los LiveBindings. Es el tema que nos ocupará en el capítulo siguiente.

---

52 Selecciona el módulo de datos en la ventana Structure o haciendo clic sobre él en el diseñador, abre la página de eventos en el Inspector de objetos y haz doble clic sobre el evento OnCreate, introduciendo la llamada al método Open del TClientDataSet.

El código asociado al módulo de datos, encargado de recuperar los datos al inicio y salvaguardarlos al final, se reducirá a dos sentencias asociadas a los eventos OnCreate y OnDestroy. Como puede verse a continuación, ni el método Open ni SaveToFile precisan parámetro alguno ya que el nombre del archivo se encuentra almacenado en la propiedad FileName del TClientDataSet (al que hemos llamado CDSOrdenadores).

```
procedure TDataModule1.DataModuleCreate(Sender: TObject);
begin
  CDSOrdenadores.Open;
end;

procedure TDataModule1.DataModuleDestroy(Sender: TObject);
begin
  CDSOrdenadores.SaveToFile;
end;
```

Podemos completar este código agregando la posibilidad de guardar periódicamente la información siempre que haya experimentado cambios. Para ello agregamos al módulo de datos un componente TTimer, asignamos el valor 30000 a su propiedad Interval para que el autoguardado se ejecute cada 30 segundos y asociamos el código siguiente al único evento de que dispone, OnTimer:

```
procedure TDataModule1.Timer1Timer(Sender: TObject);
begin
  if CDSOrdenadores.ChangeCount > 0 then
   begin
     CDSOrdenadores.ApplyUpdates(-1);
     CDSOrdenadores.SaveToFile;
   end;
end;
```

La propiedad ChangeCount del TClientDataSet contiene el número de cambios pendiente de aplicar al conjunto de datos. Si dicho número es mayor que 0 usamos el método ApplyUpdates para que esos cambios se combinen con la información previa del ClientDataSet y después lo guardamos con SaveToFile.

> NOTA
>
> La política de tratamiento de los datos propuesta aquí hace innecesaria la intervención del usuario. Un método alternativo consistiría en dejar que fuese éste quien recuperara y guardara la información a demanda, solicitándole el nombre del archivo mediante un cuadro de diálogo.

Distribución de una aplicación MyBase

Finalizado el desarrollo del proyecto llega el momento de distribuir la aplicación entre los clientes, efectuando la instalación en los ordenadores donde finalmente será explotada[53]. Hemos de tener en cuenta que aparte del ejecutable hemos de distribuir dos elementos adicionales:

- El archivo XML que hace las veces de base de datos, colocándolo en la carpeta adecuada para que la aplicación pueda encontrarlo.
- El módulo `Midas.dll` que contiene el código necesario para el funcionamiento del componente `TClientDataSet`.

El componente `TClientDataSet` no forma parte de la VCL, la FMX ni la RTL, por lo que el código asociado se almacena en una DLL específica y no se incorpora automáticamente al código ejecutable del proyecto. Sin embargo la distribución de bibliotecas adicionales, y su correcta instalación en el equipo de destino, puede ser origen de problemas en el funcionamiento de la aplicación.

Una alternativa, si no queremos incluir la citada DLL y que el programa trabaje sin problemas, consiste en agregar el módulo `MidasLib` a la cláusula `uses` del módulo de datos (o el formulario que proceda). De esta forma el código del componente `TClientDataSet` pasará a formar parte del ejecutable, ahorrándonos esta tarea de configuración.

> NOTA
>
> En caso de que también nos interese, podemos incluir la creación del archivo XML asociado al `TClientDataSet` desde el código del programa y evitar así su distribución. Básicamente tendríamos que controlar el error que se produciría en la apertura al no existir el archivo y, solamente en ese caso, invocar al método `CreateDataSet` y después a `SaveToFile` tras haber asignado el nombre del archivo a la propiedad `FileName`.

53 No debemos olvidar activar la configuración Release en el Gestor de proyectos y compilar de nuevo para obtener la versión final de la aplicación, sin incluir información de depuración.

Conclusión

Delphi XE2 ofrece múltiples métodos de acceso a bases de datos, algunos heredados de las primeras versiones del producto (BDE), otros dependientes del sistema operativo (dbGo) y otros específicos para ciertos RDBMS (IBX). El método preferente será siempre dbExpress ya que es multiplataforma, independiente por tanto del sistema operativo, y cuenta con controladores para trabajar prácticamente con cualquier RDBMS, ya sea directamente o a través del nuevo controlador ODBC.

Al desarrollar una aplicación con acceso a datos emplearemos un conjunto de componentes u otro, según el método elegido y la arquitectura de la propia aplicación. La arquitectura más simple, en la que datos y programa residen en la misma máquina y no existe necesidad de acceso simultáneo por parte de varios usuarios, puede implementarse mediante MyBase tal y como se ha mostrado en este capítulo.

El componente `TClientDataSet`, que hemos conocido como parte de MyBase, juega en realidad un papel fundamental en cualquier proyecto con acceso a datos desarrollado con Delphi como tendremos ocasión de comprobar.

A continuación

Una parte fundamental de una aplicación que trabaja con información almacenada externamente, ya sea en una base de datos o en un archivo XML como la usada a modo de ejemplo en puntos previos, es el enlace entre esos datos y los componentes que conforman la interfaz de usuario.

En el pasado la interfaz de usuario se diseñaba con componentes específicos, pensados para ser conectados a datos. En Delphi XE2 sigue existiendo esa posibilidad al trabajar con la VCL, pero también se incluye como novedad un sistema denominado LiveBindings, disponible para aplicaciones tanto VCL como FMX. Ése será el tema que abordemos en el siguiente capítulo.

Capítulo 5: LiveBindings

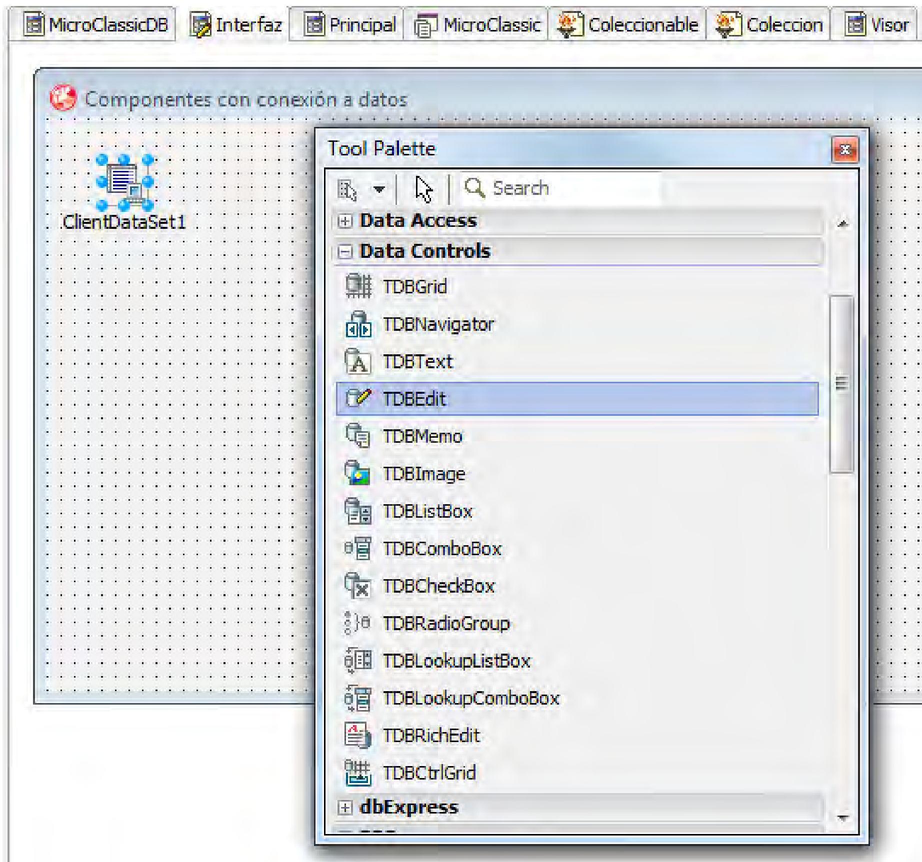
Podemos agrupar las aplicaciones que tratan información externa, por regla general almacenada en una base de datos, en dos categorías fundamentales: las que llevan a cabo un proceso sistemático y automatizado sobre los datos, por ejemplo para elaborar informes o generar agregaciones a analizar con herramientas especializadas, y aquellas que actúan como intermediarias entre las personas y los propios datos, ofreciendo una interfaz de usuario que de manera interactiva muestre la información y permita editarla.

Mostrar los datos en la interfaz de usuario implica la existencia de una conexión que sirve para trasladar la información desde su origen hacia los componentes visuales. Para que los cambios efectuados en éstos se conviertan en acciones en sentido inverso, enviando los nuevos datos hacia el lugar en el que se almacenan, dicha conexión ha de ser bidireccional.

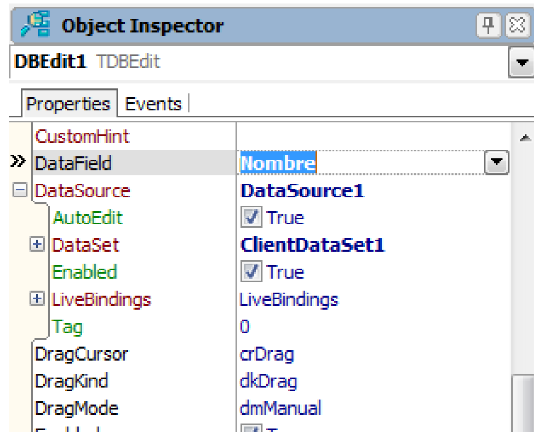
Frente al modelo heredado de versiones previas del producto, disponible únicamente para aplicaciones VCL, Delphi XE2 ofrece una alternativa multiplataforma denominada LiveBindings.

Componentes visuales con conexión a datos

El método clásico para diseñar con Delphi una interfaz de usuario conectada a una base de datos pasaba por el uso de los Data Controls (véase la imagen inferior), un conjunto de componentes funcionalmente equivalentes a los usados en cualquier interfaz: `TListBox`, `TCheckBox`, `TEdit`, etc., pero cuyo nombre comienza con el prefijo `TDB` en lugar de `T` y que se caracterizan por contar casi todos ellos con dos propiedades adicionales: `DataSource` y `DataField`.



Mediante la propiedad DataSource el control visual se conecta con un componente TDataSource que, a su vez, estará conectado con el TClientDataSet a través de la propiedad DataSet[54]. Por su parte, la propiedad DataField establece el enlace con una de las columnas del conjunto de datos, tal y como puede apreciarse en la siguiente imagen.



Este esquema cuenta con ciertas limitaciones, comenzando por el hecho de que no es posible usar cualquier componente en el diseño de la interfaz estando limitados a los del citado grupo Data Controls. Además cada uno de estos controles establece implícitamente cuál de sus propiedades es la que se enlaza con el dato indicado por DataField, no permitiéndose personalización alguna en este sentido.

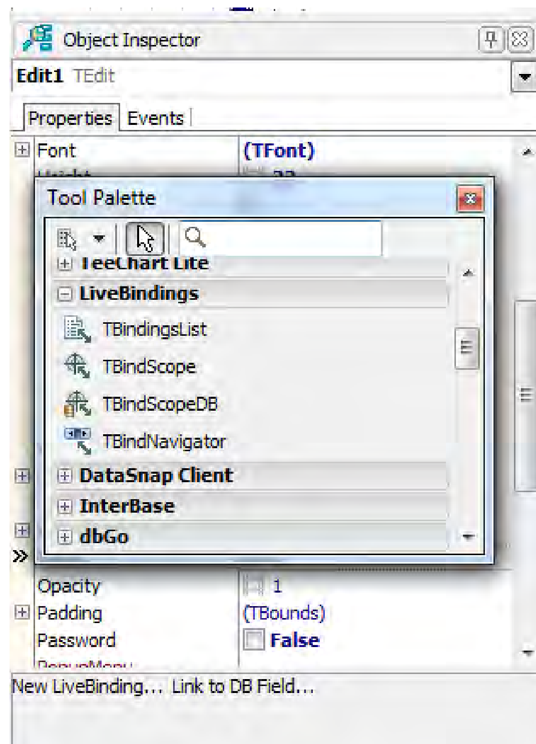
NOTA Si creamos una aplicación basada en la biblioteca VCL, con conexión a una base de datos y unas necesidades básicas en cuanto a funcionalidad, el uso de este método heredado que únicamente requiere seleccionar el control de datos adecuado y configurar dos propiedades puede ser la mejor opción precisamente por su sencillez. Para aplicaciones FMX no es una alternativa ya que no existen controles específicos con conexión a datos.

54 En realidad el TDataSource podría enlazarse con cualquier componente derivado de TDataSet, no ha de ser necesariamente un TClientDataSet.

Introducción a LiveBindings

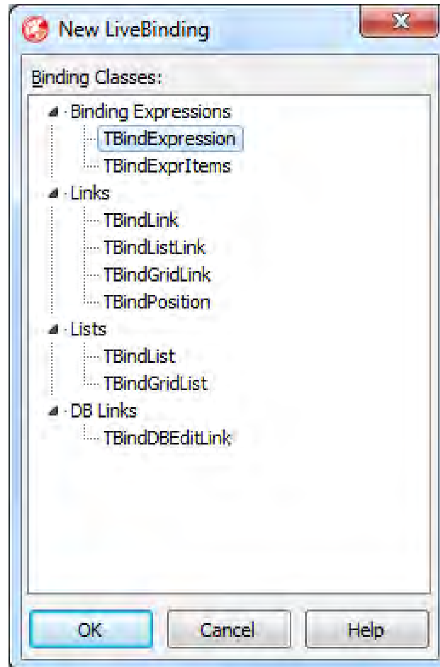
El mecanismo denominado LiveBindings de Delphi XE2 es un motor de evaluación de expresiones en las que participan propiedades de objetos. Éstos actúan como objetos origen o como objetos controlados, papel que puede invertirse y que puede recaer incluso en un mismo componente (enlazado a sí mismo).

LiveBindings está disponible tanto para aplicaciones VCL como FMX[55], por lo que representa un enfoque multiplataforma respecto al método heredado descrito en el apartado anterior. En la página LiveBindings de la Paleta de herramientas encontraremos los componentes asociados a este mecanismo, como puede apreciarse en la imagen inferior.



55 También puede utilizarse en aplicaciones de consola pero sin la ayuda del diseñador de Delphi, estableciendo la configuración de los objetos y creación de los enlaces mediante código.

En la mayoría de los casos no tendremos que agregar y configurar manualmente dichos componentes, ya que tanto en la parte inferior del Inspector de objetos como en el menú contextual asociado a los controles visuales se ofrecen opciones que, de una forma asistida, se encargarán de ese trabajo por nosotros. La opción New LiveBinding da paso al cuadro de diálogo inferior, desde el que puede crearse cualquiera de los tipos de enlace disponibles.



> ADVERTENCIA
> Los elementos LiveBindings que aparecerán en el cuadro de diálogo New LiveBinding dependen tanto del tipo de proyecto en que estemos trabajando: VCL o FMX, como del componente que tengamos seleccionado en el diseñador en ese momento.

Tipos de LiveBindings

Los LiveBinding se agrupan en dos categorías distintas: los *managed* y los *unmanaged*. Estos últimos se asemejan en cierta forma al enlace que ofrecen los componentes clásicos con vinculación a datos de la VCL, ya que implícitamente *saben* la propiedad que tienen que enlazar para cada componente y se encargan de llevar a cabo la notificación que permite que un cambio en un extremo (la información almacenada en un componente de datos) se refleje en el opuesto (un control visual).

Internamente los *unmanaged* LiveBindings emplean el patrón observador[56] para, en cierta manera, automatizar la comunicación entre los componentes enlazados enviándoles un aviso cuando los datos han cambiado en un extremo o el otro. El componente más usado en esta categoría es `TBindDBEditLink`[57] que enlaza un campo de una tabla con un componente `TEdit` o similar para facilitar su edición.

A diferencia de los anteriores, los *managed* LiveBindings dependen por completo del programador para su funcionamiento. Es necesario definir las expresiones de enlace en origen y destino, así como notificar explícitamente los cambios a fin de que esas expresiones se evalúen y actualicen las propiedades de los componentes. El pilar fundamental de los *managed* LiveBindings es el componente `TBindExpression`.

Cabría preguntarse qué sentido tienen los *managed* LiveBinding cuando se dispone de los *unmanaged*, más cómodos de usar. La razón es la flexibilidad que ofrecen los primeros frente a los segundos, especialmente cuando lo que se necesita es enlazar componentes no con información procedente de una base de datos sino de cualquier otro componente de la aplicación. Si hemos usado en alguna ocasión el evento `OnChange` de un control simplemente para asignar su contenido a otro control entenderemos rápidamente la utilidad de los *unmanaged* LiveBindings.

---

56 Observador es un patrón software pensado para que un elemento del sistema (el observador) pueda vigilar los cambios que se producen en otro (el observado). Un mismo componente puede tener múltiples observadores. Más información sobre este patrón en http://es.wikipedia.org/wiki/Observer_(patr%C3%B3n_de_dise%C3%B1o). En Delphi el patrón está implementado en los tipos `TObservers` y `TObserverMapping`, definidos en `System.Classes`.

57 Este componente es parte de la FMX. A pesar de que LiveBindings está disponible tanto en proyectos VCL como en FMX, no hay una correspondencia exacta en cuanto a los componentes.

Participantes y expresiones de un LiveBinding

Como se indicaba antes, en un LiveBindings participa un componente que actúa como origen (*source*) y otro que es el componente controlado (*control*). La información parte del primero y afecta al segundo. Además existe una expresión de origen y otra de control. Tomando como referencia un `TBindExpression` las propiedades clave a considerar son las indicadas a continuación:

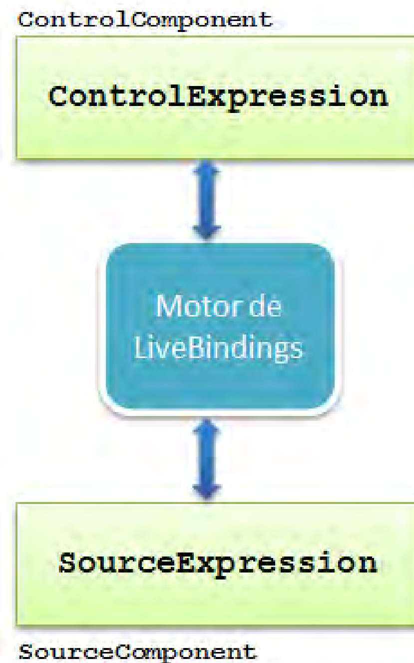
- `ControlComponent`: Referencia al componente controlado por el enlace. Normalmente esta propiedad se establece de manera automática tomando como valor el componente seleccionado en el diseñador en el momento de usar la opción New LiveBinding.
- `SourceComponent`: Referencia al componente del que procede originalmente la información que interesa vincular.
- `ControlExpression`: Es la expresión que se evaluará en el componente controlado. En el caso más sencillo es el nombre de una de sus propiedades: la propiedad a la que se asignará el dato obtenido.
- `SourceExpression`: Expresión a evaluar en el componente de origen y de la que se obtendrá la información. Al igual que en el caso anterior, puede ser simplemente el nombre de una propiedad.
- `Direction`: Determina la dirección en que se llevará a cabo la transferencia de información. Los valores posibles son tres: `dirSourceToControl`, `dirControlToSource` y `dirBidirectional`. El primero es el tomado por defecto. El segundo invierte el sentido, de manera que el componente controlado se convierte en origen y a la inversa. El tercero permite establecer una sincronización bidireccional.

Al usar un LiveBinding de tipo *unmanaged* el valor de estas propiedades se establecerá sin ninguna intervención por nuestra parte, fijando como componente origen al componente que está conectado a la base de datos y como componente controlado el que forma parte de la interfaz. La expresión de origen seleccionará una de las columnas disponibles y la de destino indicará la propiedad que permita mostrar/editar la información. Por regla general se usará un enlace bidireccional.

Las expresiones asociadas al LiveBinding no son procesadas por el compilador de Delphi, sino que se almacenan como cadenas de caracteres a interpretar y ejecutar por un motor propio de análisis de expresiones LiveBinding. Aunque, como se ha apuntado antes, en su caso más sencillo la expresión sería el nombre de una propiedad, en ella es posible utilizar un conjunto reducido de operadores y funciones (principalmente de conversión).

El conjunto de operadores está formado por los operadores aritméticos básicos y los relacionales, permitiéndose también el uso de las constantes `nil`, `true` y `false`. Con ello podríamos, por ejemplo, comprobar la coincidencia del contenido de dos propiedades en el componente de origen y generar como resultado un valor booleano que sería transferido al componente controlado. Las funciones disponibles en una expresión facilitan la conversión de números en punto flotante a enteros o de números a cadenas: `Round` y `ToStr`; la conversión de fechas: `StrToDateTime` y `FormatDateTime`; operaciones sobre cadenas de caracteres: `UpperCase`, `LowerCase` o `Format`, etc.

La figura inferior resume esquemáticamente los elementos citados:



La expresión del componente controlado, almacenada en la propiedad ControlExpression, será evaluada por parte del motor de LiveBindings en el contexto del componente indicado por ControlComponent. Es algo a tener en cuenta a la hora de introducir referencias a propiedades. De manera análoga, la expresión en SourceExpression se evaluará en el contexto del componente SourceComponent.

Conexión genérica entre propiedades

Utilizando como base el proyecto MicroClassic desarrollado en capítulos previos, vamos a poner en práctica el uso de LiveBindings tanto de tipo *managed* como *unmanaged*. Partiremos con el primer tipo aplicándolo a la conexión genérica entre propiedades de objetos cualesquiera.

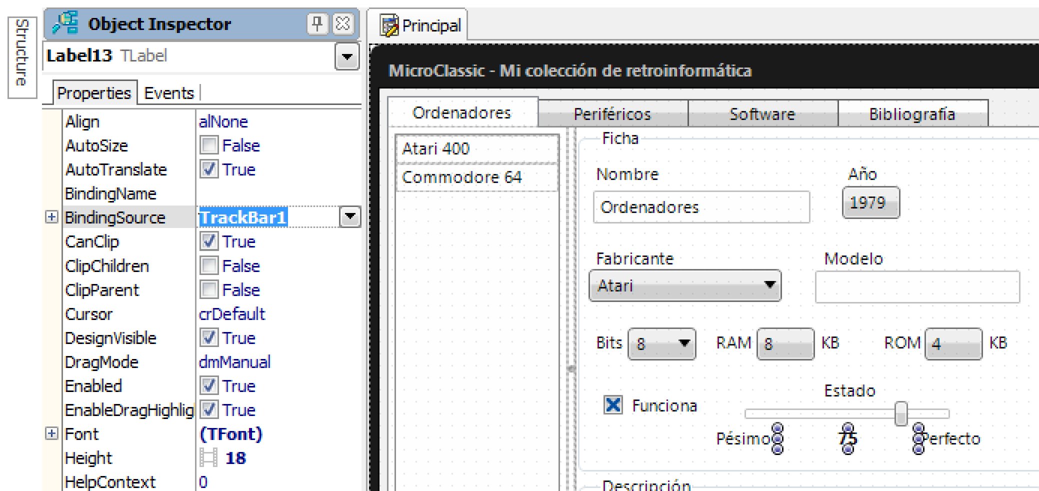
Enlace simple entre componentes FMX

Si lo único que necesitamos es crear un enlace unidireccional entre las propiedades características de dos componentes cualesquiera, siempre que éstos sean FMX, lo único que necesitamos es fijar la propiedad BindingSource de uno de ellos apuntando al otro.

La interfaz de MicroClassic cuenta entre sus elementos con un componente TTrackBar, al que se asoció la etiqueta Estado, que sirve para que el usuario de la aplicación indique el estado de conservación y funcionamiento en que se encuentra el ordenador. Aunque internamente ese control almacena un valor exacto, concretamente en su propiedad Value, visualmente se muestra tan solo como una posición aproximada del deslizador a lo largo del eje horizontal.

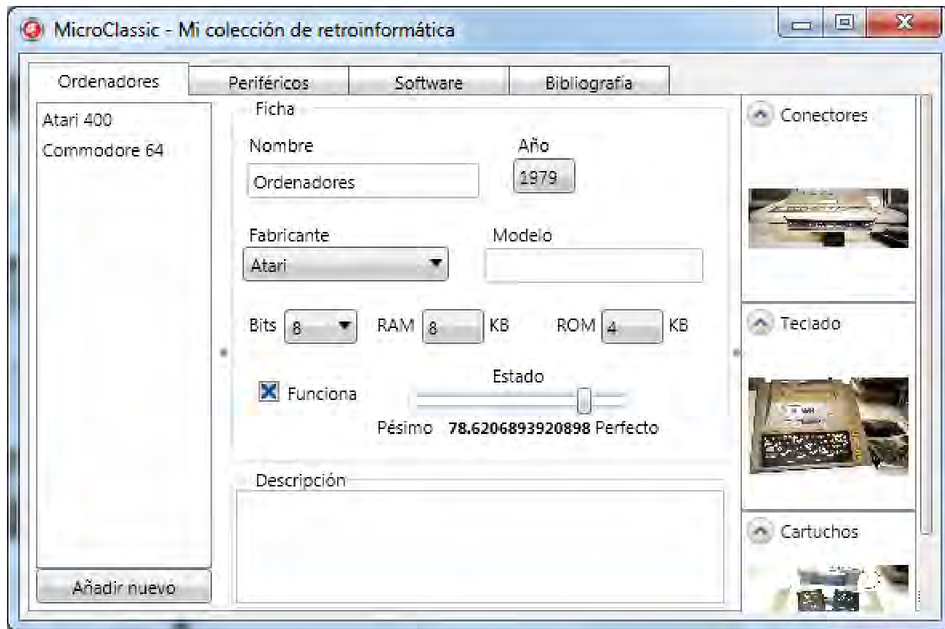
Supongamos que nuestro cliente, a pesar de que le gusta la forma de fijar el estado de cada ordenador, quiere conocer ese valor exacto. Para ello tendremos que introducir un pequeño cambio en la interfaz, agregando un componente TLabel (una etiqueta de texto) justo debajo del TTrackBar. Ajustaremos las propiedades que rigen su apariencia, como Font y TextAlign, a fin de diferenciarla de las etiquetas de los extremos.

El paso fundamental será la creación del enlace que permitirá que, de manera automática, el contenido de la propiedad `Value` del `TTrackBar` sea enviado a la propiedad `Text` del `TLabel` cada vez que se produzca un cambio. Para ello seleccionaremos la etiqueta de texto recién añadida y, como se aprecia en la imagen inferior, cambiaremos su propiedad `BindingSource`. No hay más que desplegar la lista asociada a dicha propiedad, en la que aparecen todos los componentes existentes en el diseñador, y elegir el elemento `TrackBar1`.



Si ejecutamos el proyecto tras esta modificación comprobaremos cómo cualquier cambio en la posición del `TTrackBar` se ve reflejada de inmediato en la etiqueta de texto. El contenido de la propiedad `Value` se transfiere tal cual y, dado que esa propiedad es un número en punto flotante, la etiqueta mostrará normalmente un gran número de decimales tal y como puede apreciarse en la imagen de la página siguiente.

> NOTA
>
> Teóricamente el deslizador del `TTrackBar` puede ser posicionado en cualquier punto del intervalo continuo entre los extremos establecidos por `Min` y `Max`. En la práctica al usar el teclado los saltos serán de unidad en unidad y al utilizar el ratón dependerán de la resolución de éste.



Sin disponer de la propiedad BindingSource una tarea tan sencilla como ésta habría implicado responder al evento OnChange del TTrackBar y escribir el código necesario para asignar el contenido de su propiedad Value a la propiedad Text del TLabel previa conversión de número a cadena de caracteres.

Enlace estándar

En su estado actual la aplicación MicroClassic permite seleccionar en la lista del margen izquierdo un ordenador, ya sea de los existentes inicialmente o añadido durante la ejecución, pero esa acción se traduce en una actualización de los datos mostrados por los controles que ocupan el área central, comenzando por el TEdit que muestra el nombre del ordenador, gracias a que en su momento agregamos varias sentencias asociadas al evento OnChange del TListBox. Asimismo la modificación del dato, por ejemplo en el TEdit que contiene el nombre, es transferido de vuelta como título del elemento activo en la lista porque respondemos al evento OnExit del TEdit ejecutando una sentencia de asignación.

Para enlazar el contenido del TEdit con la propiedad Text del objeto TListBoxItem que representa el elemento elegido en la lista no podemos recurrir a la propiedad BindingSource, como en el ejemplo anterior. Primero porque precisamos un enlace bidireccional y segundo porque, a diferencia del TEdit, el control TListBox tiene una estructura compleja y cada uno de sus elementos es un componente con su propio conjunto de propiedades.

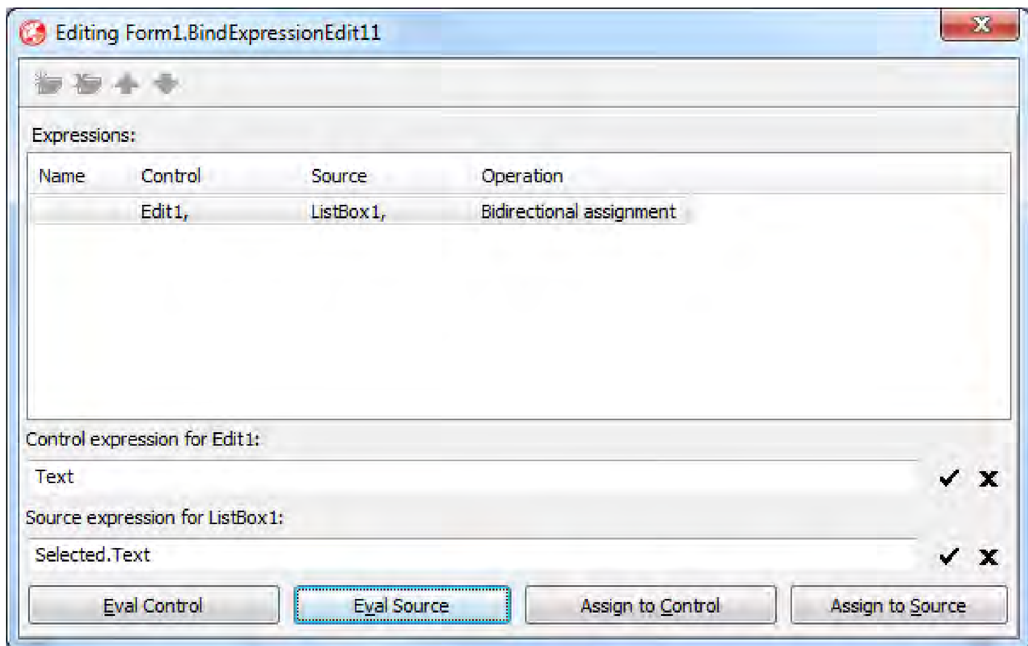
Recurriremos por tanto a un enlace estándar de tipo *managed* creando un componente TBindExpression y asignando el valor adecuado a sus propiedades. Los pasos a seguir son los indicados a continuación:

- Seleccionamos en el diseñador el control TEdit que contiene el nombre del ordenador y usamos la opción New LiveBinding de la parte inferior del Inspector de objetos para crear el enlace.
- De la ventana que enumera los tipos de enlaces disponibles escogemos TBindExpression y hacemos clic en OK.
- Veremos aparecer en el formulario un componente TBindingsList y, aunque no es visible en el formulario pero sí en la ventana Structure, un TBindExpression. Éste es el componente seleccionado y sus propiedades aparecen en el Inspector de objetos.
- Desplegamos la lista asociada a la propiedad SourceComponent y elegimos el elemento ListBox1 que es la lista que contiene los nombres de los ordenadores.
- Modificamos la propiedad Direction asignándole el valor dirBidirectional a fin de que el vínculo sea bidireccional.

En este momento tenemos vinculados los dos componentes, pero aun no hemos establecido las expresiones que han de evaluarse para llevar a cabo su sincronización. Aunque podemos introducirlas sin más en las propiedades SourceExpression y ControlExpression, dado que son sencillas, también podemos abrir el cuadro de diálogo de evaluación de expresiones mediante la opción Expressions que ha aparecido en la parte inferior del Inspector de objetos.

Utilizar este cuadro de diálogo comporta varias ventajas. La más importante es que podemos introducir y evaluar expresiones de prueba y comprobar cuál es el resultado antes de asignarlas definitivamente a las propiedades correspondientes.

Como puede verse en la imagen inferior el cuadro de diálogo dispone de dos cajas de texto: Control expression y Source expression. Introducimos en la primera la expresión que permitirá obtener/asignar la información en el control `Edit1` que, en este caso, es sencillamente el nombre de la propiedad que almacena el nombre del ordenador: `Text`. A continuación escribimos en el siguiente recuadro la expresión para el control `ListBox1`: una referencia compuesta que permite leer y escribir el texto del elemento elegido en cada momento en la lista.



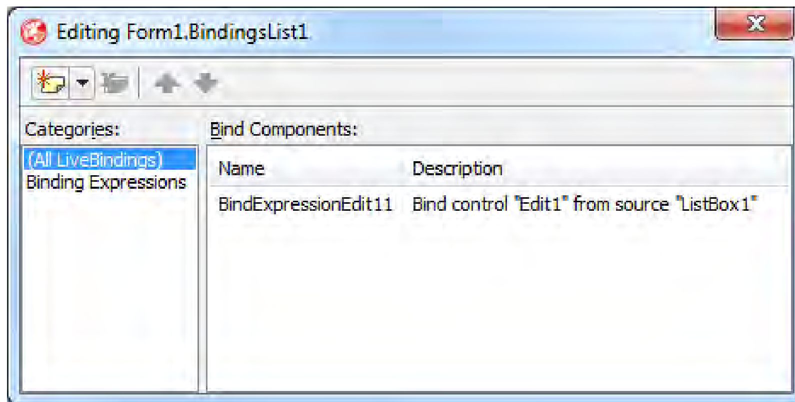
Con los botones Eval Control y Eval Source podemos evaluar independientemente las dos expresiones, obteniendo en una ventana aparte el tipo de dato y el contenido actual. En caso de que la expresión fuese incorrecta se generaría un mensaje de error. Los otros dos botones facilitan la asignación en un sentido y el opuesto.

Una vez verificada la corrección de las expresiones, los botones que hay a la derecha de cada caja de texto, con el típico símbolo de "visto bueno", sirven para asignarlas a las propiedades `ControlExpression` y `SourceExpression` del componente `TBindExpression`.

Dado que estamos usando un enlace de tipo *managed* la notificación de cuándo se produce un cambio en la información, a fin de que el componente asociado la actualice, no se produce de forma automática. Hemos de escribir el código apropiado para enviar dicha notificación.

Aunque en este caso tenemos únicamente un LiveBinding definido en una aplicación bastante sencilla, en un proyecto real puede haber decenas de ellos conectando una misma información con múltiples objetos. Es la razón de que al crear el primer LiveBinding se haya agregado a nuestro formulario un componente `TBindingsList`, cuya finalidad es mantener agrupados todos esos objetos y facilitar su manipulación.

Al hacer doble clic sobre el citado componente se abrirá un cuadro de diálogo como el siguiente. En él se enumeran los LiveBindings definidos, en este caso solamente uno, y se ofrecen opciones en una barra de botones para crear otros o eliminar el seleccionado. Un doble clic sobre cualquier elemento de la lista Bind Components dará paso a la ventana que usábamos anteriormente para configurar el LiveBinding.



El componente `TBindingsList` cuenta entre sus miembros con un método `Notify` que permite notificar cambios a todos los enlaces que contiene. Dicho método precisa dos argumentos: la referencia al componente que genera la notificación y el nombre de la propiedad que ha cambiado, dato que puede omitirse.

Para completar el enlace, por tanto, abriremos el método correspondiente al evento `OnChange` tanto del `TEdit` como del `TListBox` e introduciremos como única sentencia una invocación al citado método, tal y como se muestra a continuación:

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
  BindingsList1.Notify(Sender, '');
end;
```

Al ejecutar nuevamente el programa comprobaremos cómo la elección de un elemento de la lista se refleja de inmediato en el recuadro de texto. Los cambios efectuados en éste serán enviados a la lista en el momento en que se produzca. Incluso podemos añadir nuevos ordenadores a la lista y establecer su nombre.

Enlace compuesto

Si bien se trata de un enlace de tipo *managed*, el del punto anterior sigue conectando únicamente dos propiedades de dos componentes. En un enlace compuesto la expresión de origen o control (normalmente no ambas) puede hacer referencia a más de un objeto, así como incluir operadores y llamadas a funciones. Son enlaces necesariamente unidireccionales y resultan útiles en muchas ocasiones.

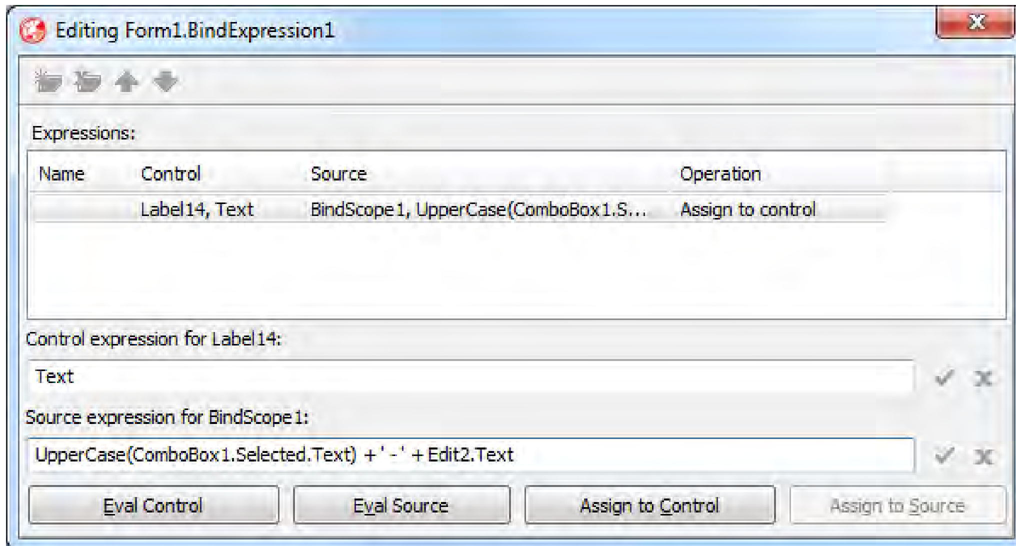
El problema que plantean este tipo de vínculos estriba en que las propiedades `SourceComponent` y `ControlComponent` únicamente pueden almacenar la referencia a un componente. Si en la expresión asociada, en la propiedad `SourceExpression` o `ControlExpression` respectivamente, se introduce el nombre de varias propiedades se asumirá que todas ellas pertenecen a ese componente, a ningún otro.

La solución estriba en que la expresión emplee referencias completas en las que aparezca también el nombre del componente, resolviendo así el ámbito en que ha de evaluarse cada propiedad. Estas referencias completas no están permitidas si en `SourceComponent/ControlComponent` se introduce el nombre de uno de los componentes implicados. En su lugar es necesario agregar al formulario un componente `TBindScope`.

Vamos a agregar a la interfaz de `MicroClassic` un componente `TStatusBar`, una barra de estado que se colocará automáticamente en el margen inferior de la ventana. Si es necesario modificaremos el tamaño del `TTabControl` y su propiedad `Align` para que el ajuste siga siendo perfecto. En el interior del `TStatusBar` introduciremos un control `TLabel`. La finalidad de esta barra de estado será mostrar una cadena compuesta del nombre del fabricante del ordenador, en mayúsculas, y el nombre de éste.

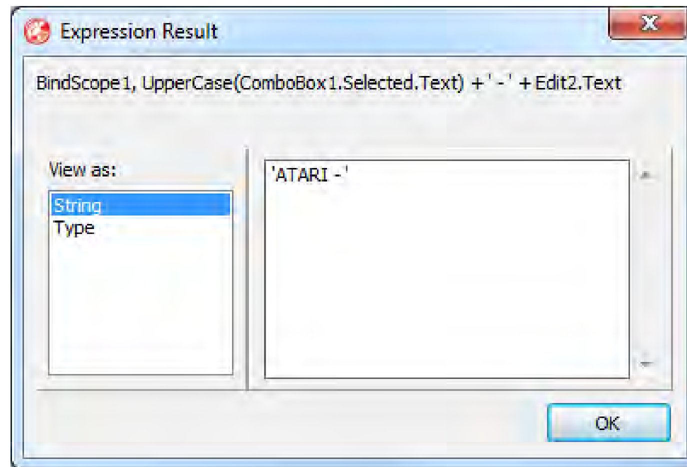
También añadiremos al formulario un componente TBindScope. A continuación, teniendo seleccionado el control TLabel, usaremos la opción New LiveBinding y agregaremos un nuevo TBindExpression. La propiedad ControlComponent apuntará al TLabel, la propiedad SourceComponent la modificaremos seleccionando el objeto BindScope1.

Para configurar las expresiones utilizaremos de nuevo el cuadro de diálogo que conocimos en el apartado previo. La expresión asociada al TLabel, como puede verse en la imagen inferior, es una simple referencia a su propiedad Text. La parte interesante está en la expresión de origen, encargada de leer el texto del elemento seleccionado en el TComboBox (nombre del fabricante), convertirlo a mayúsculas y concatenarlo con el contenido del TEdit que almacena el modelo de ordenador.



Mediante el botón Eval Source podemos comprobar si la expresión se evalúa correctamente[58] (véase imagen de la página siguiente). Si no fuese así el resultado sería nil. Incluso podemos hacer clic en Assign to Source para apreciar el efecto en la barra de estado.

58 En el cuadro de diálogo Expression Result por defecto aparecerá el tipo resultante de la expresión, que en este caso será String. Eligiendo dicho tipo, en la lista que hay a la izquierda, podremos ver el resultado en el panel derecho.



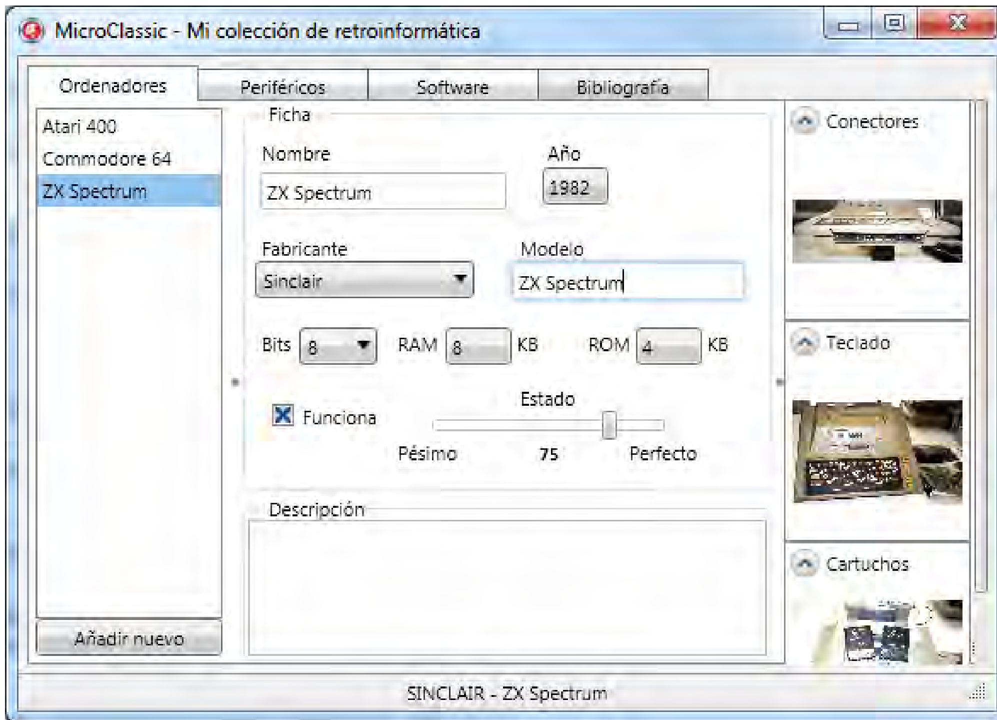
Dado que se trata de un enlace de tipo *managed* tendremos que agregar el código necesario para notificar los cambios en el origen que provoquen la actualización del destino. Para ello debemos seleccionar conjuntamente los dos componentes: `TComboBox` y `TEdit`, y en el Inspector de objetos abrir la página de eventos y hacer doble clic en el evento `OnChange`. El método se asociará tanto al `TComboBox` como el `TEdit`.

> NOTA
>
> Para seleccionar más de un componente, lo cual permite tanto editar simultáneamente sus propiedades comunes como asociar un mismo método de gestión a sus eventos, tendremos que mantener pulsada la tecla Control mientras hacemos clic sobre ellos. En la parte superior del Inspector de objetos se indicará cuántos componentes tenemos seleccionados en cada momento.

La única sentencia de este método sería la invocación al método `Notify` del componente `TBindingsList`, como en el ejemplo previo:

```
procedure TForm1.ComboBox1Change(Sender: TObject);
begin
  BindingsList1.Notify(Sender, '');
end;
```

Completados estos pasos podremos volver a ejecutar el programa y comprobar el resultado. En la imagen inferior puede verse la barra de estado tras haber agregado un nuevo ordenador, elegido el fabricante `Sinclair` de la lista desplegable e introducido `ZX Spectrum` como modelo.



> ADVERTENCIA
>
> Los enlaces compuestos casi siempre son unidireccionales, a pesar de lo cual en el Inspector de objetos podremos asignar el valor `dirBidirectional` a la propiedad `Direction` del `TBindExpression` sin ningún problema. Sin embargo a la hora de ejecutar el proyecto se generará un error de manera casi inmediata, ya que no es posible evaluar la expresión asociada a `SourceExpression` en orden inverso, es decir, para partiendo del contenido de la barra de estado asignar valores al `TComboBox` y `TEdit`.

Enlace con componentes de acceso a datos

Muchos de los objetos que forman parte de LiveBindings cuentan con versiones específicas para la conexión con componentes de acceso a datos. Éstos se diferencian por llevar el sufijo `DB` en el nombre de clase, por ejemplo `TBindScopeDB` que sería equivalente al `TBindScope` que hemos utilizado en el último ejemplo.

Al igual que los enlaces de tipo *managed* éstos podemos crearlos manualmente o bien a través de diversos asistentes. El más usual es al que da paso la opción Link to DB Field que encontraremos tanto en la parte inferior del Inspector de objetos, siempre que tengamos seleccionado un componente, como en el menú contextual de éste. Antes de poder usarla, no obstante, es necesario que en el proyecto exista un origen de datos.

Estos vínculos, como se explicó anteriormente, son *unmanaged* y, por tanto, no requieren tanto trabajo de configuración por parte del programador y en la mayoría de los casos tampoco código asociado para la notificación[59] de los cambios.

El punto de partida de los siguientes apartados será el proyecto `MicroClassic` tras haber añadido el módulo de datos del capítulo previo, en el que existía un componente `TClientDataSet` con una serie de columnas definidas. Agregaremos al módulo de datos un componente `TDataSource`, cuya propiedad `DataSet` modificaremos para que apunte al `TClientDataSet`, que servirá como intermediario entre el origen de los datos y los controles que forman la interfaz de usuario.

Volviendo al módulo principal del proyecto, el correspondiente al formulario que actúa como interfaz de usuario del programa, usaremos la opción File>Use Unit para agregar una referencia al módulo de datos. De esta manera podremos acceder a todo su contenido, tanto desde el código como durante la fase de diseño. Sin este paso no podríamos, por ejemplo, vincular un `TEdit` con una columna de datos del `TClientDataSet`.

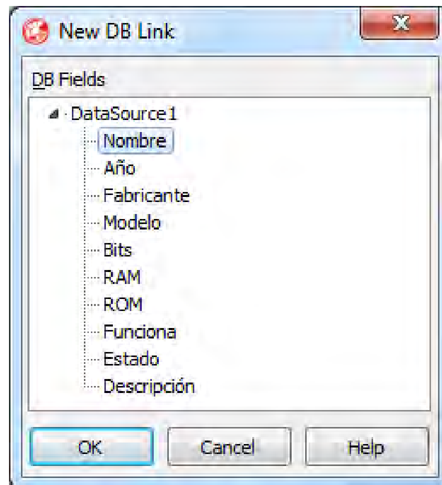
---

59 Esto es cierto siempre que los componentes vinculados implementen el patrón observador, explicado con anterioridad, a través de `TObserver`.

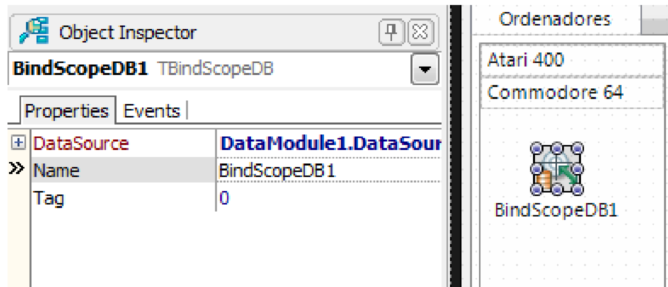
Enlazando componentes simples

Establecer un enlace entre ciertos tipos de controles visuales, entre los que se encuentran los TEdit, TCheckBox, TNumberBox y TMemo, es una tarea muy sencilla. Basta con seleccionar el control, hacer clic en la opción Link to DB Field que aparece en la parte inferior del Inspector de objetos y seleccionar del cuadro de diálogo New DB Link la columna de datos.

La imagen inferior corresponde al citado cuadro de diálogo y ofrece todas las columnas del TClientDataSet. Asumiendo que hemos elegido el primer TEdit de la ventana, asociado al nombre del ordenador, escogeríamos la columna Nombre y haríamos clic en el botón OK.

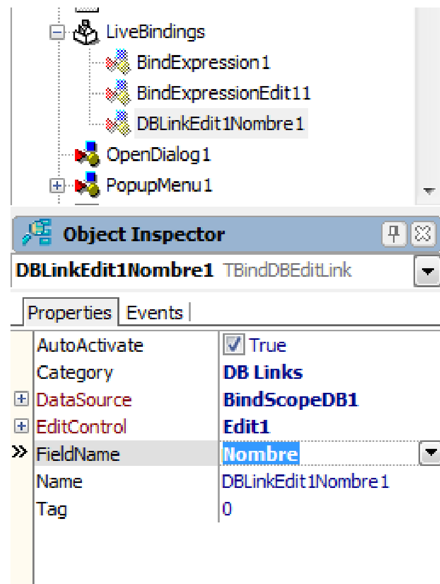


Al crear el primer enlace de este tipo veremos cómo se agrega al formulario un componente TBindScopeDB (véase imagen inferior). Éste se encarga de la vinculación entre el ámbito actual (el formulario) y el TDataSource.



No tenemos que preocuparnos de este componente, su configuración se establece de forma automática. Únicamente habremos de seleccionarlo como `SourceComponent` en los casos en que debamos ajustar manualmente un enlace como veremos más adelante.

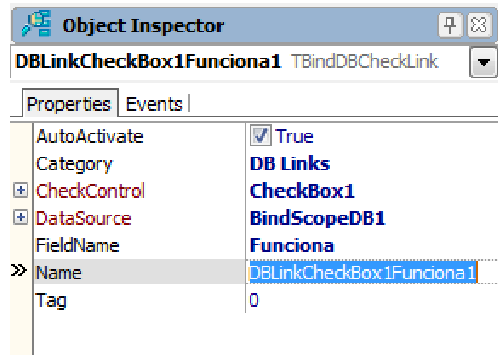
Examinando el grupo LiveBindings en la ventana Structure veremos que también se ha creado un `TBindDBEditLink`. Se trata de un *unmanaged* LiveBinding específico para actuar con controles como `TEdit` o `TNumberBox`. Dispone de una propiedad `EditControl` en sustitución de la propiedad `ControlComponent` que teníamos en `TBindExpression`. De manera análoga la propiedad `SourceComponent` es sustituida por `DataSource` y la expresión de `SourceExpression` por la propiedad `FieldName` que selecciona una de las columnas de datos disponibles. En la figura inferior puede apreciarse la configuración del `TBindDBEditLink` que vincula el primer `TEdit` del formulario con la columna `Nombre`.

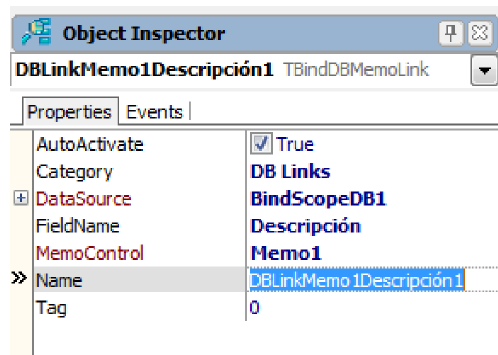


NOTA

Aunque normalmente no es necesario, el componente `TBindDBEditLink` genera eventos como `OnAssignedValue` y `OnAssigningValue` que nos permite ejercer cierto control sobre el proceso de vinculación de datos.

Siguiendo el mismo procedimiento enlazaremos el resto de controles de la interfaz con sus respectivas columnas. Los TEdit y TNumberBox generarán un componente TBindDBEditLink, el control TCheckBox un componente TBindDBCheckLink y el control TMemo (la descripción del ordenador) un componente TBindDBMemoLink tal y como puede comprobarse en las siguientes imágenes del Inspector de objetos.

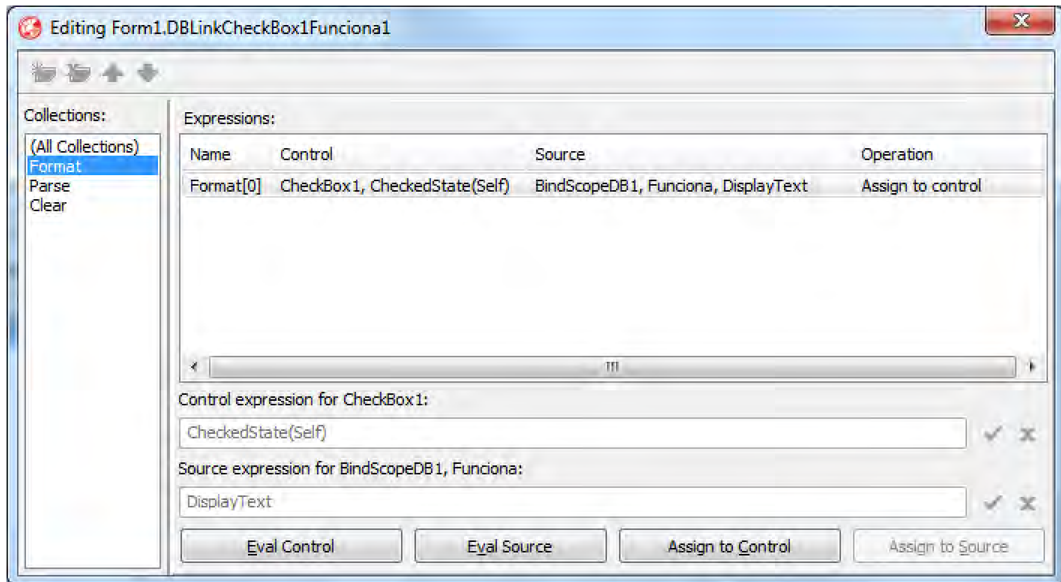




El enlace de los control TComboBox que hay en la interfaz requeriría algo más de trabajo, ya que actualmente no tenemos almacenados externamente la lista de valores. Lo que haremos será sustituirlos por sendos TEdit que facilitarán la edición del fabricante y el número de bits.

El control TTrackBar no implementa TObserver, por ello no puede enlazarse directamente con una columna de datos. Por el momento podemos sustituirlo también por un TEdit como en el caso de los TComboBox. Más adelante veremos cómo crear vínculos para este tipo de componentes.

Cada uno de los componentes `TBindDBxxxLink` tiene asociadas varias expresiones, si bien éstas no aparecen directamente en el Inspector de objetos. Usando la opción `Expressions` (en su menú contextual o el propio Inspector de objetos) abriremos un cuadro de diálogo como el mostrado en la imagen inferior. En este caso corresponde al enlace asociado al control `TCheckBox`.



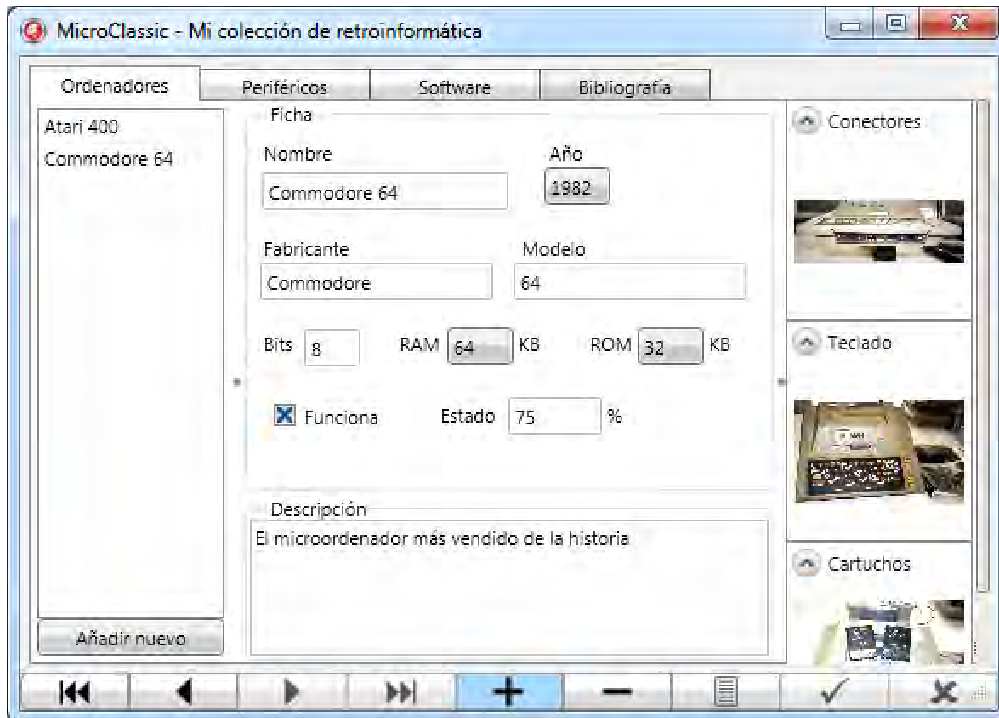
En el margen izquierdo aparecen las categorías de las expresiones que, en este caso, son tres: `Format`, `Parse` y `Clear`. La primera es la encargada de controlar la transferencia de la información desde la columna de datos hacia el control de interfaz, mientras que la segunda se ocupa de la transferencia en sentido inverso. La tercera tiene la misión de inicializar el control, estableciendo el valor que debe mostrar cuando no exista información en el origen de datos.

Las expresiones de estos componentes `TBindDBxxxLink` está *bloqueadas*, no podemos modificarlas, eliminarlas ni agregar otras nuevas. Se trata de componentes especializados en el enlace entre tipos concretos de objetos, para los cuales están fijadas las propiedades de enlace y las expresiones de evaluación.

Continuando con el proyecto `MicroClassic`, el último paso de su rediseño será agregar un componente `TBindNavigator` en el interior del

TStatusBar (la barra de estado introducida en un paso previo), eliminando la etiqueta de texto que había antes en ese área de la interfaz. El TBindNavigator se enlazará con el componente TBindDBScope a través de su propiedad BindScope y su objetivo será facilitar la navegación por los datos y ejecución de operaciones estándar: agregar un nuevo ordenador, eliminarlo, confirmar o descargar los cambios, etc.

La imagen inferior muestra la interfaz del programa en funcionamiento[60], tras haber añadido algunos ordenadores. Esos datos ahora se conservan entre ejecuciones de la aplicación, ya que en su momento habíamos agregado el código necesario al módulo de datos para guardar el contenido del TClientDataSet al salir y recuperarlo al iniciar la ejecución.



60 En este momento se encuentran vinculados a sus respectivos datos todos los controles del área central de la interfaz de usuario, pero no la lista de ordenadores que hay en el margen izquierdo ni la de imágenes que ocupa el área derecha.

Enlaces TBindLink

Podemos encontrarnos con casos en los que no exista un componente TBindDBxxxLink específico para el control que necesitamos usar en la interfaz de usuario. En esta situación siempre podemos recurrir a un enlace *unmanaged* de tipo genérico: el componente TBindLink. Podemos agregarlo desde la ventana New LiveBinding, como el TBindExpression.

La diferencia fundamental entre un TBindLink y los componentes de enlace usados en el punto previo es que el primero no tiene preestablecidas las expresiones que se almacenan en las propiedades FormatExpressions, ParseExpressions y ClearExpressions. Una vez seleccionados el componente de origen y controlado, con las propiedades SourceComponent y ControlComponent, debemos definir personalmente esas expresiones.

Por lo demás un TBindLink opera como cualquier otro LiveBinding de tipo *unmanaged*, lo cual implica que los componentes enlazados han de implementar TObserver para que la notificación de cambios sea automática.

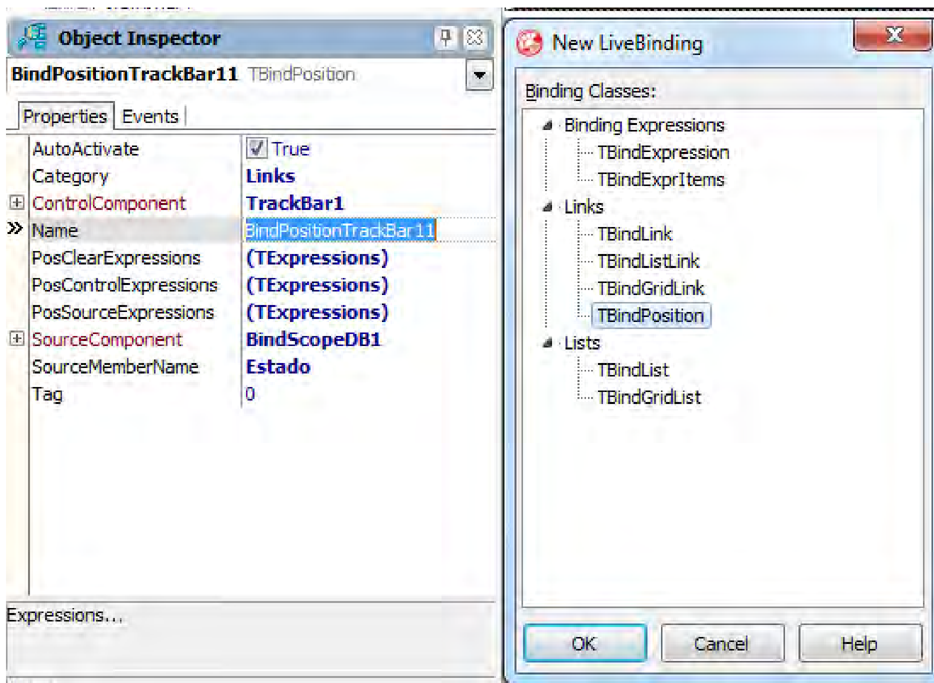
Enlaces TBindPosition

TBindPosition es otro de los componentes LiveBindings que podemos encontrar en la ventana New LiveBinding. Su finalidad es facilitar la vinculación de controles que almacenan un texto o un número, como es el caso de TEdit, TMemo o TNumberBox, sino un valor que se asocia a una posición entre dos extremos, como es el caso del control TTrackBar.

Al igual que los demás LiveBinding, el componente TBindPosition cuenta con las propiedades ControlComponent y SourceComponent que apuntarán a los componentes a vincular, así como con la propiedad SourceMemberName que selecciona una de las columnas existentes en caso de que SourceComponent haga referencia a un componente TBindScopeDB o similar.

Como en el caso de TBindLink este componente también dispone de tres propiedades en las que se almacenan expresiones de evaluación, en este caso para establecer la posición en el componente controlado: PosControlExpressions, en el origen: PosSourceExpression y poner a cero la posición: PosClearExpressions.

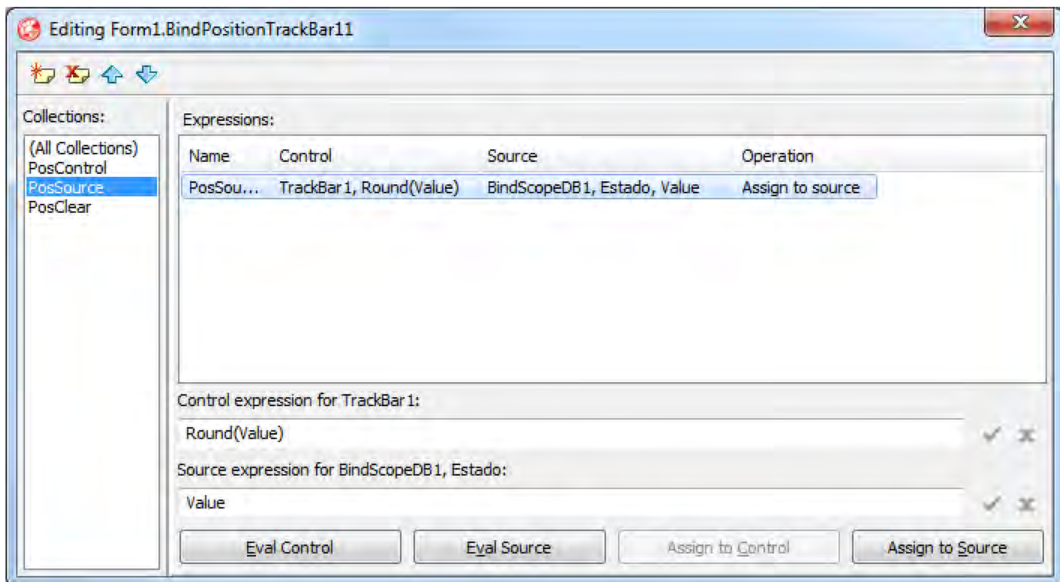
Usando un `TBindComponent` podríamos vincular el control `TTrackBar` que existía originalmente en la interfaz de nuestro programa con la columna que almacena el estado del ordenador. Agregamos el `TTrackBar` y, usando la opción New Live Binding, le asociamos un `TBindPosition`. La propiedad `ControlComponent` de ésta ya tiene el valor correcto, por lo que modificamos `SourceComponent` y `SourceMember` para que hagan referencia a la citada columna de datos como puede apreciarse en la imagen inferior.



A continuación haremos clic en la opción Expressions (parte inferior del Inspector de objetos) para abrir la ventana de edición de expresiones. Encontrándonos en el grupo `PosControl` hacemos clic en el primero de los botones que hay en la barra superior de dicha ventana, creando una nueva expresión vacía. Ésta será la que se utilice para transferir el contenido de la columna de datos al control `TTrackBar`. Tanto éste como la columna `Estado` del `TClientDataSet` cuentan con una propiedad `Value`, que es la que almacena la información, por lo que no tenemos más que introducir el nombre de dicha propiedad como expresión de control y origen, sin más.

En el grupo PosSource repetimos la operación, agregando una nueva expresión y facilitando los datos necesarios para, en este caso, asignar el contenido del TTrackBar a la columna Estado del TClientDataSet. Hemos de tener en cuenta, sin embargo, que la propiedad Value del TTrackBar almacena un dato en punto flotante, mientras que la columna Estado es de tipo SmallInt. Asignar un entero a una propiedad que contiene un número en punto flotante no plantea ningún problema, de ahí que en la expresión de control basta con indicar el nombre de las propiedades, pero la asignación en sentido inverso conlleva pérdida de información y no puede realizarse directamente.

Como se muestra en la imagen inferior, en la expresión que obtendrá el contenido del TTrackBar para enviarlo a la propiedad Value de la columna de datos llamamos a la función Round. Ésta redondeará el valor en punto flotante almacenado en la propiedad Value del TTrackBar al entero más próximo, siendo éste el dato a asignar.



Finalmente introduciremos la expresión de inicialización en la categoría PosClear. En la expresión de control estará la propiedad Value del TTrackBar y en la de origen el valor 0, de forma que se ponga a 0 cuando no existan datos en el origen.

Si ejecutáramos el proyecto tras estos últimos cambios veríamos cómo al cambiar de ordenador (asumiendo que tenemos varios) la posición del TTrackBar se establece de acorde al contenido de la columna Estado. Sin embargo si intentamos actuar directamente sobre el TTrackBar, modificando su posición, obtendremos un mensaje error.

Cuando se inicia la modificación de un control vinculado a datos el LiveBinding correspondiente se encarga de poner en modo de edición el origen, en este caso el TClientDataSet. El problema estriba en que el TBindPosition no se encarga de esa tarea cuando cambia la posición del TTrackBar, por lo que habremos de ocuparnos nosotros.

No tenemos más que hacer doble clic sobre el TTrackBar, abriendo el método asociado a su evento OnChange, y llamar al método Edit del TClientDataSet para poner éste en modo de edición:

```
procedure TForm1.TrackBar1Change(Sender: TObject);
begin
  DataModule1.CDSOrdenadores.Edit;
end;
```

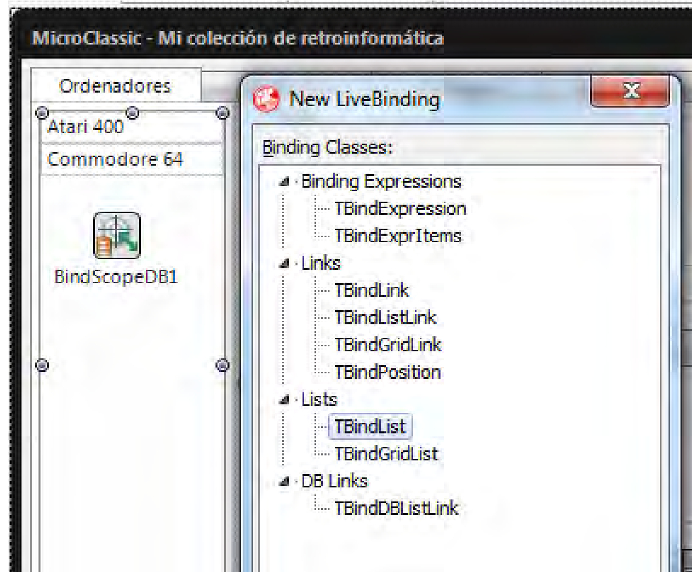
A partir de este momento podremos usar el TTrackBar como cualquier otro control enlazado a datos, reflejando el valor actual y facilitando su modificación.

Enlaces TBindList

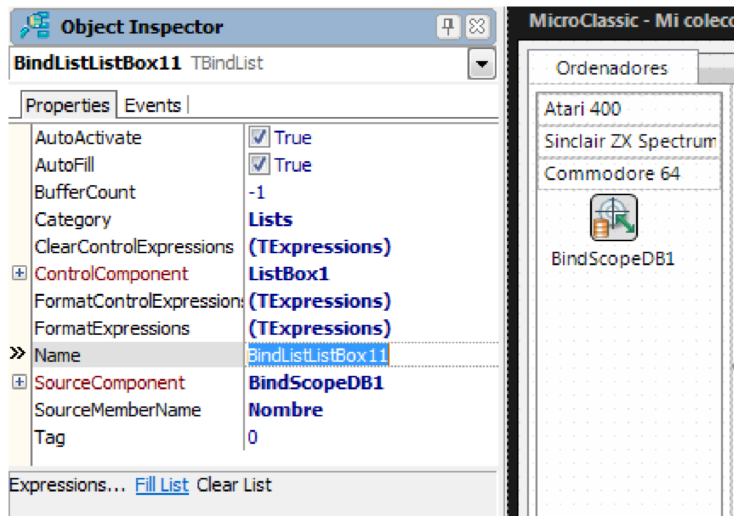
Uno de los elementos de la interfaz de usuario que queda sin enlazar con la base de datos es la lista en la que aparecen los nombres de ordenador. Se trata de un control TListBox y se diferencia, respecto a otros como TEdit o TCheckBox, en que no muestra el dato que contiene una columna en el registro (ordenador) actual, sino el contenido de una cierta columna para todos los registros existentes.

Para controles como éste (también para TComboBox) existe el LiveBinding TBindList, especializado en trabajar con listas de datos. A las habituales propiedades ControlComponent, SourceComponent y SourceMemberName se añaden otras tres cuya función es el almacenamiento de otras tantas colecciones de expresiones: FormatExpressions, FormatControlExpressions y ClearControlExpressions. Si lo único que necesitamos es llenar la lista con información procedente de la base de datos bastará con una expresión en la primera colección.

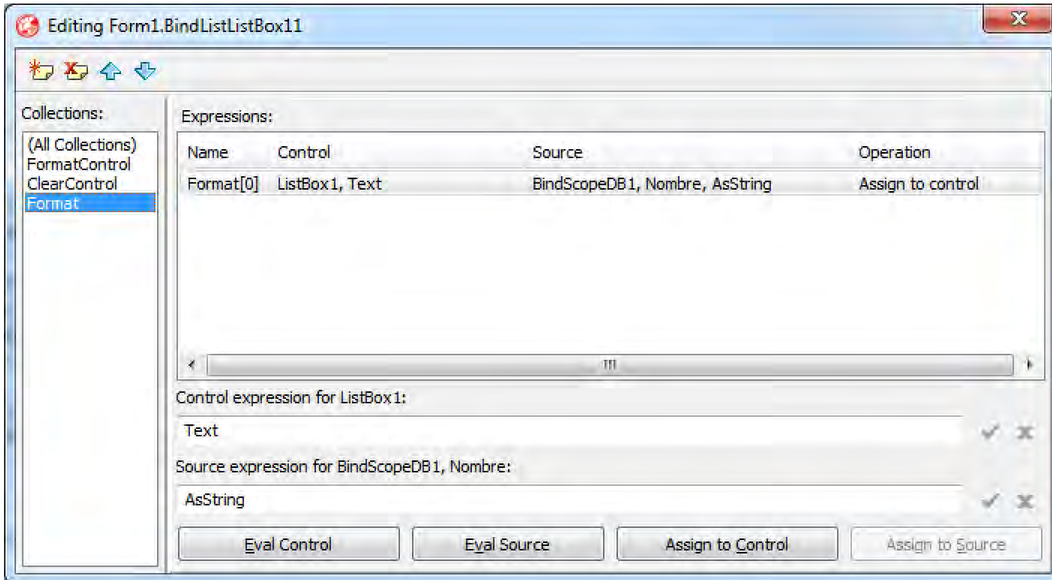
Veamos en la práctica cómo usar este tipo de LiveBinding. Comenzamos seleccionando la lista de ordenadores, que aún muestra los elementos que introdujimos manualmente en un principio, usamos la opción New Live Binding y seleccionamos el elemento TBindList.



A continuación modificamos SourceComponent y SourceMemberName para hacer referencia a la columna Nombre del TClientDataSet (véase la imagen inferior).



Mediante la opción Expressions abrimos el cuadro de diálogo mostrado en la siguiente imagen. En ella ya hemos añadido una expresión al grupo `Format`, seleccionando como destino la propiedad `Text` del `TListBox` y como origen la propiedad `AsString` de la columna Nombre.



Tras este paso podemos usar la opción Fill List del `TBindList` (véase parte inferior de la imagen de la página anterior) para rellenar la lista y comprobar el resultado. Los tres elementos que aparecen han sido leídos del `TClientDataSet`. En realidad no hace falta rellenar la lista durante la fase de diseño, es una mera comprobación. Podemos usar la opción Clear List para dejarla vacía. En el momento en que se ejecute el programa se rellenará automáticamente.

> ADVERTENCIA
>
> Para que la lista funcione correctamente es necesario eliminar el código (agregado en un capítulo previo) de respuesta a los eventos `OnChange` y `OnExit` que asignaba al `TEdit` el texto del elemento elegido y a éste el texto introducido en el `TEdit`. Asimismo habrá que dar el valor `False` a la propiedad `Sorted` del `TListBox`.

El enlace TBindList se encargará de introducir en la lista los nombres de los ordenadores al iniciarse la aplicación, pero no mantendrá un vínculo entre el elemento elegido en ésta y la posición en el TClientDataSet (el registro actual). Tampoco actualizará automáticamente el contenido del TListBox si se añaden o eliminan ordenadores.

Podríamos crear LiveBindings adicionales, por ejemplo un TBindPosition para sincronizar la posición en el TClientDataset[61] con el elemento seleccionado en la lista. Es un trabajo que también podemos realizar mediante código, posiblemente de manera más sencilla en este caso particular.

Cada vez que cambie el elemento elegido en el TListBox queremos que se seleccione el ordenador adecuado. Para ello aprovecharemos el evento OnChange de la lista, leeremos el nombre del ordenador elegido de Selected.Text y lo usaremos para realizar una búsqueda en el TClientDataSet mediante su método Locate[62], tal y como puede verse a continuación:

```
procedure TForm1.ListBox1Change(Sender: TObject);
begin
  DataModule1.CDSOrdenadores.Locate(
       'Nombre',ListBox1.Selected.Text, []);
end;
```

De manera análoga, cada vez que se cambie el ordenador actual con alguno de los botones del TBindNavigator ha de elegirse el elemento adecuado de la lista. Además hay botones que agregan y eliminan registros, operaciones ante las cuales es necesario actualizar el TListBox. Podemos hacerlo, como se muestra a continuación, con el método FillList del TBindListBox:

```
procedure TForm1.BindNavigator1Click(Sender: TObject;
                              Button: TBindNavigateBtn);
begin
 BindListListBox11.FillList;
 if DataModule1.CDSOrdenadores.RecNo <> -1 then
  ListBox1.ItemIndex:=DataModule1.CDSOrdenadores.RecNo-1;
end;
```

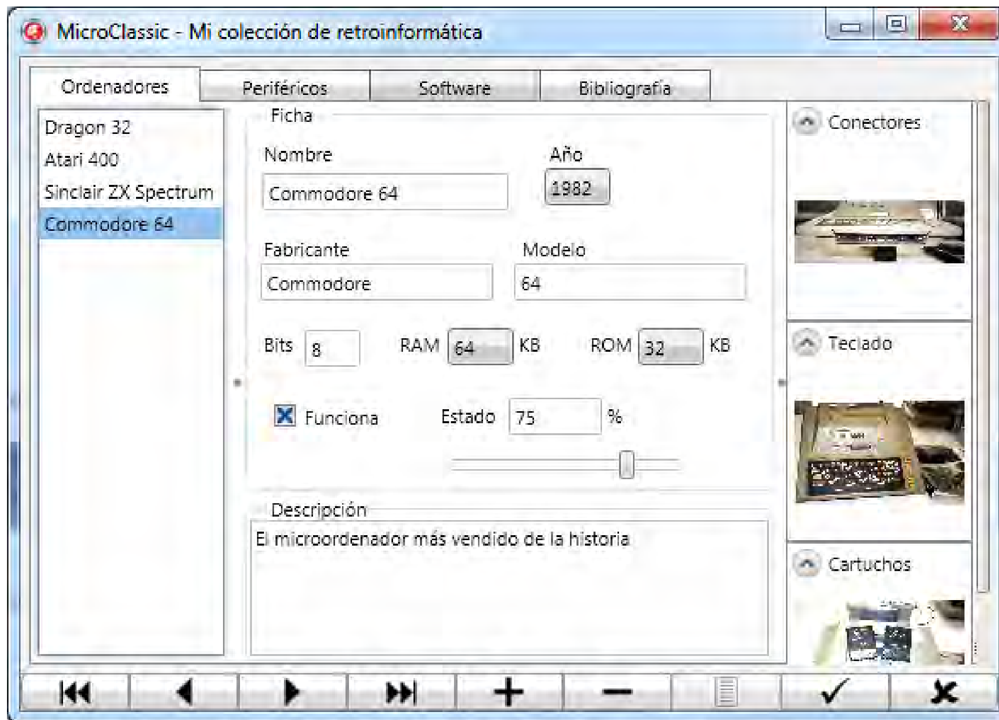
---

61 Dicha posición está almacenada en la propiedad RecNo de la clase TClientDataSet y puede ser tanto leída como modificada.

62 Este método necesita tres parámetros: el nombre de la columna por la que se buscará, el dato a buscar y un conjunto de opciones que, por ejemplo, permiten indicar si ha de distinguirse o no entre mayúsculas y minúsculas.

La comprobación de si el registro actual en el TClientDataSet, almacenado en la propiedad RecNo, no es -1 es necesaria ya que al añadir y eliminar datos no hay un registro actual y la asignación a la propiedad ItemIndex del TListBox generaría un error.

Dado que el TBindNavigator facilita todas las operaciones necesarias sobre los datos, incluido el registro de nuevos ordenadores, ya no tiene sentido la existencia del botón Añadir nuevo que tenemos debajo de la lista. Podemos eliminarlo junto con el código que tenía asociado. La interfaz del programa quedaría como puede verse en la imagen inferior.



En su estado actual el programa nos permite tanto navegar por los registros secuencialmente, con los botones anterior y siguiente, como seleccionar directamente un ordenador desde el componente TListBox. La sincronización entre el área central de la interfaz y la lista es bidireccional, si bien parte del trabajo para ello se realiza mediante código.

Enlace de las imágenes de un ordenador

Para completar el almacenamiento de los datos del proyecto MicroClassic faltaría el enlace de las imágenes asociadas a cada ordenador. Es una tarea que se deja como ejercicio y para el que se ofrecen las siguientes indicaciones:

- En el módulo de datos habrá que añadir un nuevo TClientDataSet que tendría una columna para almacenar una imagen y otra con el nombre del ordenador y que servirá como enlace[63] entre éste y el primer TClientDataSet.
- Agregar un segundo TDataSource asociado al nuevo TClientDataSet para facilitar el acceso a sus datos desde el formulario.
- Definir un enlace que mantenga sincronizado el ordenador actual con las imágenes mostradas, así como los LiveBinding necesarios para que el TImage obtenga la información del TClientDataSet.

Una alternativa sería agregar al actual TClientDataSet una columna adicional para almacenar una imagen, vinculándola con un componente que sustituyese a la lista que hay en el margen derecho. Esto sin embargo limitaría a una el número de imágenes por ordenador.

Conclusión

Delphi XE2 incorpora un mecanismo totalmente nuevo para enlazar componentes a fin de facilitar la transferencia de información de unos a otros, ya sea de forma unidireccional o bidireccional. Es un método disponible tanto para aplicaciones VCL como FMX, si bien para proyectos VCL sigue siendo posible usar la técnica heredada de versiones previas y que se basa en la existencia de componentes específicos con vinculación a datos.

---

63 Sería mucho más eficiente que cada ordenador tuviese asociado un código numérico (la clave principal) y que ése se utilizase en el segundo TClientDataSet, a modo de clave foránea, para enlazar las imágenes con el ordenador al que corresponden. Es una mejora a este proyecto que está pensado como ejercicio didáctico más que como modelo de eficiencia.

LiveBindings es un mecanismo muy flexible y en este capítulo hemos aprendido lo necesario para poder usar prácticamente cualquier tipo de enlace. Los de tipo *managed* requieren la notificación explícita cuando se producen cambios, pero es posible escribir una sola sentencia invocando al método `Notify` en un método y asociarlo con el evento `OnChange` de todos los componentes. Si no tuviésemos LiveBindings esa notificación tendría que realizarse con un tratamiento individual para cada caso, como se venía haciendo hasta ahora.

Los LiveBindings *unmanaged* se asemejan a los controles vinculados a datos de la VCL, ya que implícitamente saben qué propiedades son las que intervienen y la notificación de cambios es automática, pero siguen ofreciendo más flexibilidad a través de componentes como `TBindLink` o `TBindPosition`.

De los componentes de enlace a controles de interfaz únicamente no hemos tratado `TBindDBGridLink`, cuya finalidad es enlazar un origen de datos, como puede ser un `TDataSource`, con una cuadrícula. Es tan sencillo como, partiendo de un formulario vacío, agregar un componente `TStringGrid` y usar la opción Link to DB DataSource para crear el enlace. La cuadrícula mostrará de inmediato toda la información del `TClientDataSet`.

A continuación

Junto con el anterior, el presente capítulo nos ha servido para saber cómo podemos almacenar información de manera persistente en Delphi XE2, en principio localmente mediante un componente `TClientDataSet`, y cómo enlazar esa información con una interfaz de usuario para facilitar su manipulación.

En el capítulo siguiente avanzaremos en el desarrollo de nuestro proyecto, trasladando la información desde un archivo local en formato XML a una base de datos gestionada por un RDBMS remoto. Para ello debemos conocer los componentes que forman parte de dbExpress y aprender a usarlos conjuntamente con los que ya conocemos: `TClientDataSet` y LiveBindings.

Capítulo 6: dbExpress

En el capítulo previo hemos aprendido a almacenar información localmente gracias a uno de los métodos que incorpora Delphi XE2: MyBase. En el cuarto capítulo supimos que para la conexión con bases de datos, tanto locales como remotas (RDBMS), Delphi nos ofrece múltiples opciones: BDE, dbGo, IBX y dbExpress entre otras. Apuntábamos entonces que, salvo por razones suficientes que forzasen otra decisión, el método preferente sería siempre dbExpress.

Partiendo de lo que ya sabemos, principalmente el uso del componente `TClientDataSet` y el enlace entre datos e interfaz mediante LiveBindings, el objetivo de este capítulo es describir el proceso a seguir para conectar con una base de datos mediante dbExpress y trabajar con la información almacenada en ella.

Se asume que ya contamos con algún RDBMS en funcionamiento, ya sea en el propio equipo de desarrollo o en un servidor, así como las herramientas de administración necesarias para crear una base de datos.

Componentes dbExpress

Vamos a comenzar esbozando una visión general sobre cómo usar dbExpress para acceder a la información almacenada en una base de datos, identificando los diferentes componentes que necesitaremos emplear. En los siguientes apartados del capítulo profundizaremos en los atributos y configuración de cada uno de ellos, aplicando esa teoría en la práctica sobre una versión modificada del proyecto `MicroClassic` que hemos ido evolucionando en capítulos previos.

Por regla general dbExpress se usa en aplicaciones con arquitectura en dos (cliente/servidor, como el caso mostrado en la imagen de la página siguiente) o más capas. El RDBMS se ejecuta en una máquina que actúa como servidor de datos, siendo el único software con acceso directo a los datos, mientras que la aplicación es ejecutada en una o más máquinas cliente: una por usuario del sistema. En esas máquinas se encontrará el software cliente del RDBMS, encargado de comunicarse con el servidor a través de la infraestructura de red existente.

El puente entre nuestra aplicación y el software cliente del RDBMS[64] está representado por el componente `TSQLConnection`, parte de dbExpress. A éste pueden conectarse otros como `TSQLTable`, `TSQLQuery`, `TSQLStoredProc` o `TSQLDataSet` que facilitan el acceso a una tabla, una consulta, un procedimiento almacenado o un conjunto de datos genérico, respectivamente.

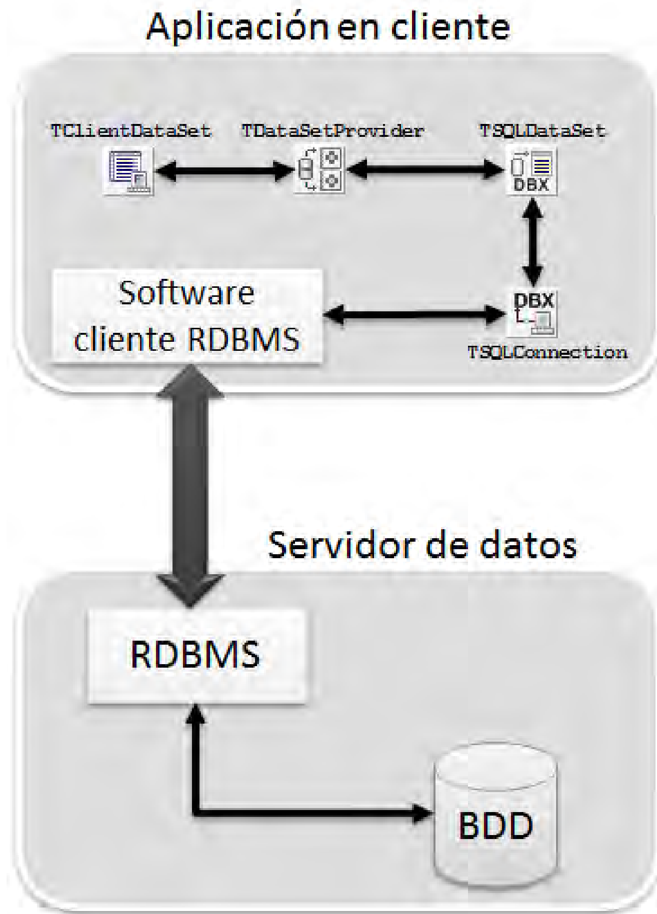
Los componentes dbExpress encargados de ejecutar operaciones de obtención de datos operan por regla general sobre información que es solamente de lectura: un conjunto de una o más filas recuperadas a partir de una consulta. Además dichos conjuntos de filas suelen ser unidireccionales, por lo que únicamente podrían recorrerse de principio a fin.

Para facilitar la actualización de los datos y la navegación por éstos se recurre a los componentes `TDataSetProvider` y `TClientDataSet`, conectando el primero a uno de los componentes dbExpress y el segundo a su vez al primero. El almacenamiento local que ofrece el `TClientDataSet` facilitará la navegación por los datos, mientras que el `TDataSetProvider` será el encargado de enviar al RDBMS las operaciones de actualización.

---

64 En algunos casos el software cliente del RDBMS puede estar embebido en la propia aplicación pero, por regla general, se instalará de manera independiente.

Uniendo todas las piezas que acaban de enumerarse llegaríamos a la estructura representada en el esquema inferior. En él no aparece la conexión del `TClientDataSet` a un componente `TDataSource` que, como vimos en el capítulo previo, permitiría la conexión de los datos con la interfaz de usuario de la aplicación mediante LiveBindings.



Ésta sería la estructura general de una aplicación conectada a una base de datos mediante dbExpress, pero no siempre se precisarán todos los componentes. Si únicamente queremos ejecutar una sentencia SQL sobre el RDBMS, sin recuperar información, bastaría con el `TSQLConnection` y un `TSQLQuery` conectado a él.

Conexión con la base de datos

Para poder acceder a una base de datos mediante dbExpress lo primero que necesitaremos será configurar la conexión, normalmente a través de un componente TSQLConnection. Básicamente tenemos dos alternativas posibles:

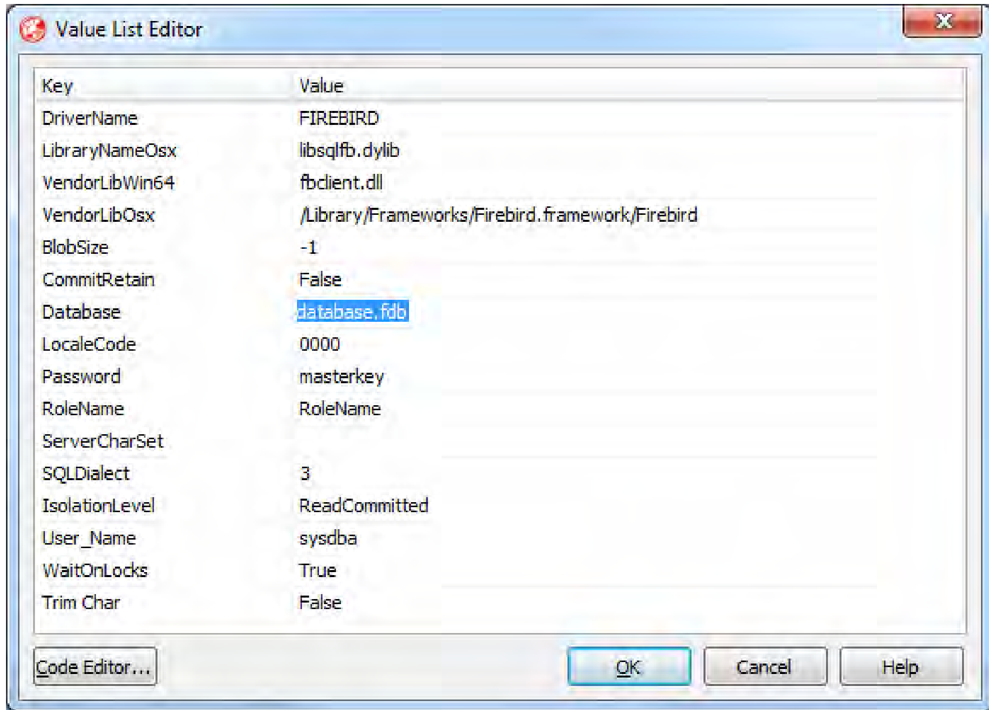
- Configurar la conexión de forma manual asignando los valores apropiados a las propiedades de TSQLConnection, ya sea en la fase de diseño (desde el Inspector de objetos) o durante la ejecución (mediante código).
- Crear la conexión desde el Explorador de datos de Delphi, lo cual permite comprobar su funcionamiento de manera interactiva, recuperando la configuración en el TSQLConnection simplemente seleccionando el nombre de la conexión en la propiedad ConnectionName.

Si optamos por el primer método deberemos asignar a la propiedad Driver uno de los valores ofrecidos en la lista desplegable, seleccionando el tipo de base de datos con la que vamos a trabajar. Esta acción desencadenará la lectura de los parámetros necesarios para establecer el contenido de propiedades como LibraryName, VendorLib y GetDriverFunc con datos obtenidos del archivo de configuración dbxdrivers.ini [65]. Es la información que precisa el componente TSQLConnection para localizar tanto el controlador dbExpress como el software cliente del RDBMS y poder así comunicarse con este último.

Además tendremos que facilitar el nombre (o dirección IP) del servidor y la ruta donde se encuentra la base de datos, así como las credenciales: usuario y contraseña, que se usarán para establecer la conexión. Estos parámetros, junto a otros muchos, están almacenados en la propiedad Params de TSQLConnection, una propiedad que cuenta con un editor específico (véase la imagen de la página siguiente). Los parámetros imprescindibles son tres: Database, User_Name y Password.

---

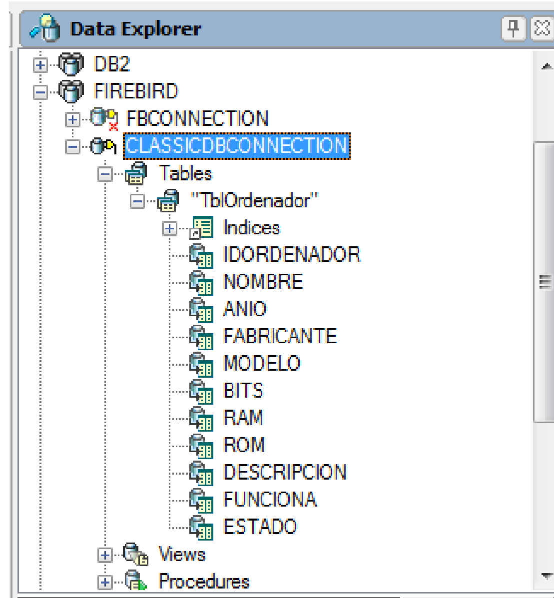
65 Este archivo, junto con dbxconnections.ini, se almacena por defecto en %PUBLIC%\Documents\RAD Studio\dbExpress\9.0 durante la instalación de Delphi XE2. El contenido de la lista asociada a la propiedad Driver se obtiene del mismo archivo. El número de controladores disponibles dependerá de la edición del producto que esté utilizándose.



Las credenciales serán solicitadas directamente al usuario de la aplicación, con independencia de los datos asignados a los parámetros `User_Name` y `Password`, si la propiedad `LoginPrompt` tiene el valor `True` que es el que toma por defecto. Si el usuario ya ha tenido que identificarse para acceder al sistema y poder iniciar la aplicación, que es lo habitual, esta nueva solicitud de identificación suele delegarse en la propia aplicación, asignando el valor `False` a `LoginPrompt` y entregando en las citados parámetros el nombre y contraseña del usuario actual.

En cuanto al parámetro `Database` su contenido se ajustará al formato `servidor:ruta/nombrebdd`, siendo servidor el nombre o dirección IP del RDBMS. En caso de que tengamos el servidor de datos en el propio equipo de trabajo, algo habitual durante el proceso de desarrollo, podemos usar `localhost` como nombre de servidor u omitir esta parte. Tanto la ruta como el nombre de la base de datos dependerán en cierta medida del RDBMS empleado. Con Firebird, por ejemplo, el nombre de la base de datos es el del archivo donde reside ésta, incluyendo la extensión correspondiente.

La segunda alternativa apuntada anteriormente, asumiendo que tenemos definida la conexión en el Explorador de datos y que desde ella podemos acceder a la base de datos como está haciéndose en la imagen inferior, es mucho más inmediata. Al desplegar la lista asociada a la propiedad ConnectionName y elegir la conexión se establecen todos los parámetros enumerados hasta ahora, Driver y Params entre ellos.

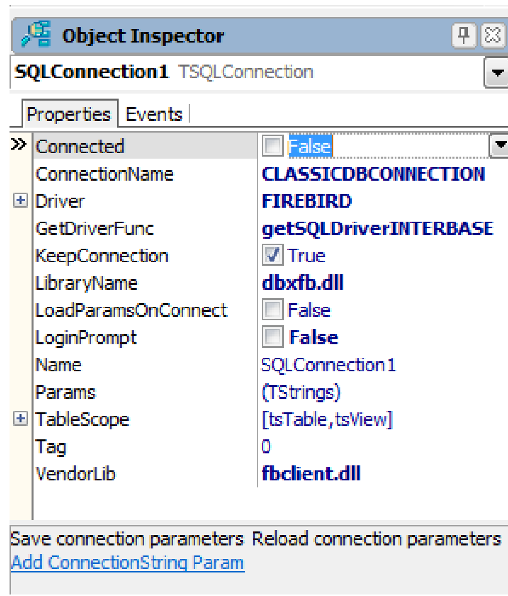


Cada vez que desde el Explorador de datos creamos una nueva conexión toda la información asociada se almacena en el archivo dbxconnections.ini. Delphi lee dicho archivo obteniendo el nombre de todas las conexiones predefinidas para mostrar sus nombres en la propiedad ConnectionName. Cuando se elige una de ellas se leen del mencionado archivo todos los parámetros de configuración y se asignan a las propiedades adecuadas de TSQLConnection, de forma que este componente replica la conexión definida previamente de manera interactiva.

Independientemente de cómo se haya efectuado la configuración de la conexión, a partir de una conexión predefinida o de forma manual, podemos comprobar la validez de los parámetros activando la propiedad Connected del componente TSQLConnection. Si todo va bien no aparecerá ningún mensaje de error, únicamente la ventana de identificación en caso de que la propiedad LoginPrompt tenga el valor True.

Almacenar y recuperar los parámetros de conexión

Al editar las propiedades del componente `TSQLConnection` aparecerán en el Inspector de objetos (véase la imagen inferior) varias opciones relativas al almacenamiento y recuperación de los parámetros de conexión. Éstas nos permiten, tras haber cambiado uno o más de esos parámetros, guardar la nueva configuración en el archivo `dbxconnections.ini` o bien restablecer el contenido de las variables a partir de ese archivo (Reload connection parameters). Si no usamos la primera opción cualquier cambio realizado en los datos de conexión afectarán únicamente al componente `TSQLConnection`, cuyas propiedades se almacenan en el módulo `.fmx`.



La opción Add ConnectionString Param agregará a la lista de parámetros almacenados en la propiedad `Params` una entrada adicional, con el nombre `ConnectionString`, que contendrá toda la información precisa para configurar tanto el controlador dbExpress como la conexión con la base de datos. La cadena asignada a `ConnectionString` podría almacenarse de manera independiente, en un archivo externo, o bien integrarse en el código del proyecto.

Configuración de la conexión al ejecutar

Dado que las propiedades del componente TSQLConnection son almacenadas en el ejecutable obtenido, al igual que las propiedades de cualquier otro componente Delphi, la configuración de la conexión a la base de datos será restaurada con los parámetros establecidos en la fase de diseño. No es necesario que en el ordenador cliente existan los archivos dbxdrivers.ini y dbxconnections.ini, al ejecutar el programa se usa el contenido de las propiedades LibraryName, VendorLib, GetDriverFunc y los parámetros almacenados en Params, pero no se hace uso de ConnectionName ni Driver.

En caso de que sea necesario ajustar alguno de los parámetros, suele ser un requerimiento para fijar el nombre de usuario y la contraseña, podemos recurrir a la propiedad Params.Values de TSQLConnection. El momento adecuado para ello suele ser al producirse el evento OnLogin o, si no hemos dato el valor True a la propiedad LoginPrompt, el evento BeforeConnect. Un ejemplo típico sería el siguiente:

```
procedure TDataModule1.CLASSICDBCONNECTIONBeforeConnect(
              Sender: TObject);
begin
  with CLASSICDBCONNECTION do begin
    Params.Values['User_Name'] := 'Francisco';
    Params.Values['Password'] := 'micontraseña';
  end;
end;
```

Almacenar la configuración de conexión en un archivo externo puede resultar más cómodo en ciertos casos, cuando el nombre del servidor, de la propia base de datos o las credenciales de identificación pueden cambiar y lógicamente no se quiere tener que modificar el código y recompilar la aplicación cada vez que esto ocurra. Para que el componente TSQLConnection lea su configuración del archivo dbxconnections.ini en cada ejecución[66] no hay más que dar el valor True a la propiedad LoadParamsOnConnect que, por defecto, tiene el valor False.

---

66 Obviamente esto conlleva la necesidad de distribuir el citado archivo de configuración de manera conjunta a la aplicación. Normalmente se toma el dbxconnections.ini original y se elimina todo su contenido a excepción de la entrada correspondiente a la conexión que precisa la aplicación, asignada a la propiedad ConnectionName del componente TSQLConnection.

> TRUCO
> En lugar de asignar el valor True a la propiedad LoadParamsOnConnect y distribuir el archivo dbxconnections.ini podemos usar el método LoadParamsFromIniFile, dejando a False la citada propiedad. El resultado final es el mismo: la lectura de los datos de conexión desde un archivo de configuración, pero LoadParamsFromIniFile realizará la lectura desde el archivo que le indiquemos como parámetro cuyo nombre no ha de ser necesariamente dbxconnections.ini. Lo que sí debe respetarse, no obstante, es el formato de dicho archivo.

Es posible recuperar la configuración de conexión desde un archivo externo y, además, completar/modificar los parámetros que se necesiten recurriendo a los eventos antes citados, por ejemplo para fijar las credenciales de acceso a la base de datos.

Una vez que el componente TSQLConnection tiene la información necesaria, para abrir la conexión con el RDBMS usaremos el método Open o bien daremos el valor True a la propiedad Connected. La conexión puede llevarse a cabo implícitamente, sin necesidad de usar Open o Connected, por ejemplo al ejecutar un comando para recuperar un conjunto de datos.

El método Close es el encargado de cerrar la conexión, una operación que también se realiza automáticamente cuando se detecta que no hay ningún conjunto de datos abierto salvo que la propiedad KeepConnection tenga el valor True. Mantener activa la conexión evita tener que volver a abrirla cada vez que se ejecuta un comando sobre el RDBMS, proceso que suele requerir un cierto tiempo al tener que procesar cada vez las credenciales de identificación.

Obtener información de conexiones y controladores dbExpress

En la mayor parte de las aplicaciones se conoce previamente la base de datos con la que ha de trabajarse y, por lo tanto, es viable definir una conexión y guardar los parámetros en un archivo de configuración o directamente el componente TSQLConnection.

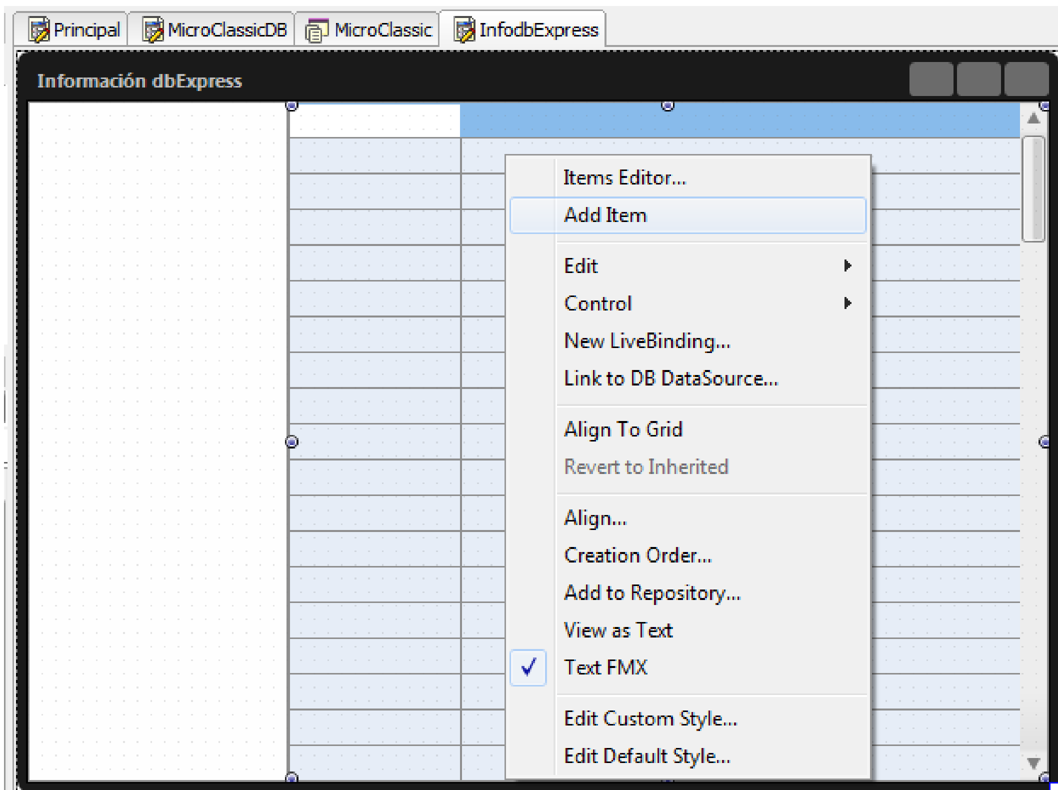
Si el proyecto que vamos a desarrollar no cuenta con dicha información preliminar, por ejemplo si se quiere crear una utilidad general para bases de datos, podemos obtenerla durante la ejecución de distintas formas. Una de ellas es a través de los métodos de la clase TDBXConnectionFactory definida en el módulo DBXCommon. El procedimiento a seguir sería el indicado a continuación:

- Agregar el módulo DBXCommon a la cláusula uses de nuestro propio módulo de código.
- Obtener una referencia al *singleton* TDBXConnectionFactory mediante su método GetConnectionFactory.
- Recuperar la lista de controladores y conexiones definidas mediante los métodos GetDriverNames y GetConnectionItems. Ambos toman como parámetro un objeto TStrings en el que devolverán las entradas existentes.
- Obtener la información que se precise de un controlador o conexión concreto mediante los métodos GetDriverProperties y GetConnectionProperties. Ambos toman como parámetro una cadena, con el nombre del controlador o conexión, y devuelven un objeto TDBXProperties conteniendo las propiedades.

Una vez que dispongamos de los datos de una conexión la propia clase TDBXConnectionFactory nos permitirá obtener a partir de ella un objeto TDBXConnection para conectar con la base de datos. El método encargado de crear esa conexión es GetConnection y podemos invocarlo de dos maneras diferentes: facilitando el nombre de la conexión y las credenciales de conexión (nombre de usuario y contraseña) o bien entregando como parámetro un objeto TDBXProperties con las propiedades de la conexión. Éste puede ser el obtenido con GetConnectionProperties y en el que se habrían modificado, en caso necesario, los parámetros User_Name y Password.

Construir una interfaz que, apoyándose en los objetos y métodos citados, facilite información sobre todas las conexiones definidas en el sistema es realmente simple. Partiendo de un formulario vacío agregamos un control TListBox y un TStringGrid. Ajustamos el primero al margen izquierdo de la ventana y el segundo para que ocupe el resto del área disponible, modificando sus propiedades Align que tomarían los valores alLeft y alClient, respectivamente.

El componente TStringGrid tendrá una sola columna que vamos a crear desde su menú contextual (véase la imagen inferior), mediante la opción Add Item. Podemos también ocultar la cabecera de la cuadrícula dando el valor False a la propiedad ShowHeader. Tanto el ancho como el contenido de la columna se determinarán al ejecutar el programa.



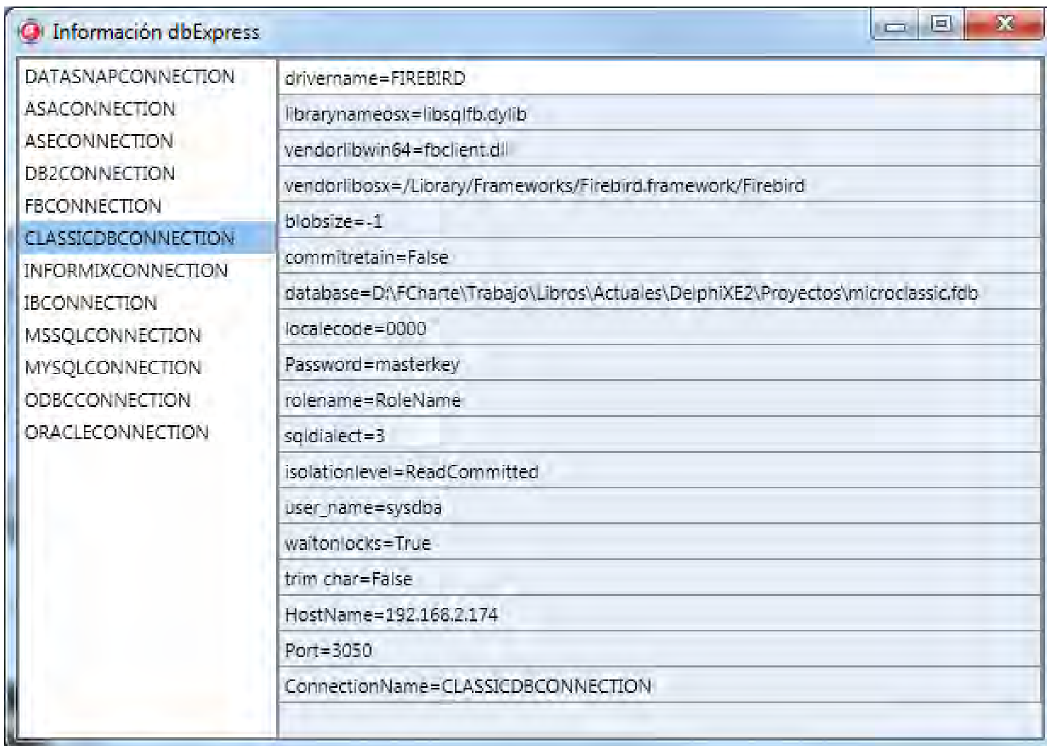
La finalidad del TListBox es enumerar las conexiones disponibles, obtenidas mediante el método GetConnectionItems citado antes. El código irá asociado al evento OnCreate del propio formulario como puede comprobarse a continuación:

```
procedure TForm3.FormCreate(Sender: TObject);
begin
  TDBXConnectionFactory.GetConnectionFactory.GetConnection
Items(ListBox1.Items);
end;
```

Cada vez que se cambie el elemento elegido en la lista (evento `OnChange`) deberemos obtener las propiedades de esa conexión, con el método `GetConnectionProperties`, y mostrarlas en el `TStringGrid`. El código para ello es el siguiente:

```
procedure TForm3.ListBox1Change(Sender: TObject);
var
  nombres, valores: TWideStringArray;
  i: Integer;
begin
  TDBXConnectionFactory.GetConnectionFactory.GetConnection
Properties(ListBox1.Selected.Text).GetLists(nombres,
valores);
  with StringGrid1 do begin
    Columns[0].Width := StringGrid1.Width;
    RowCount := Length(nombres);
    for i := 0 to Length(nombres)-1 do
      Cells[0,i] := nombres[i] + '=' + valores[i];
  end;
end;
```

En la imagen inferior puede verse el programa en funcionamiento.



Si obtenemos un objeto TDBXConnection asociado a una conexión, con el citado método GetConnection, podemos recuperar datos sobre estructura (los metadatos) mediante comandos especializados que ofrecen listas de tablas, vistas o índices, columnas de las tablas e incluso el código de los procedimientos almacenados y funciones. Si nos interesa este tipo de funcionalidad hemos de recurrir al módulo DBXMetaDataNames y su casi medio centenar de clases.

Aunque con menor abundancia de detalles, el componente TSQLConnection también cuenta con métodos que facilitan información sobre la base de datos a la que está conectado, entre ellos GetTableNames entrega una lista de nombres de tablas, GetFieldNames las columnas de una tabla dada, GetIndexNames los índices asociados a una tabla, etc.

```
procedure TDataModule1.CLASSICDBCONNECTIONBeforeConnect(Sender: TC
begin
  with CLASSICDBCONNECTION do begin
    Params.Values['User_Name'] := 'sysdba';
    Params.Values['Password'] := 'masterkey';

    |
  en
end;

proc
begi
   w



   e
end;
```

```
function   ExecuteDirect(const SQL: string): Integer;
procedure  GetFieldNames(const TableName: string; List: TStrings);
procedure  GetIndexNames(const TableName: string; List: TStrings);
procedure  GetProcedureNames(List: TStrings);
procedure  GetPackageNames(List: TStrings);
procedure  GetSchemaNames(List: TStrings);
procedure  GetCommandTypes(List: TStrings);
procedure  GetServerMethodNames(List: TStrings);
function   GetDefaultSchemaName: string;
procedure  GetProcedureParams(ProcedureName: string; List: TList);
procedure  GetTableNames(List: TStrings; SystemTables: Boolean = False);
procedure  LoadParamsFromIniFile(FFileName: string = '');
```

Combinando la información que ofrecen las clases del módulo DBXCommon, en particular el nombre de la conexión que puede ser asignado a la propiedad ConnectionName de un TSQLConnection para abrir una conexión, y los datos de los citados métodos podríamos ampliar el ejemplo anterior para obtener una funcionalidad similar a la del Explorador de datos de Delphi. Como se apuntaba anteriormente, en la mayoría de los proyectos esto no es necesario porque la aplicación se diseña para trabajar con una base de datos cuya estructura es bien conocida, lo cual permite construir una interfaz de usuario a medida en lugar de una de tipo genérico.

Ejecución de comandos

Establecida la conexión con la base de datos, nuestra aplicación dispone de un canal para hacer llegar al RDBMS los comandos que precise ejecutar. Aunque el componente TSQLConnection cuenta con un método Execute que permite enviar un comando cualquiera y obtener la respuesta del RDBMS, salvo que la sentencia SQL a ejecutar sea de tipo DDL[67] siempre suele recurrirse a uno de los componentes dbExpress derivados de TCustomSQLDataSet. Todos ellos tienen miembros comunes, al derivar de una base común, entre los que se encuentra la propiedad SQLConnection cuya finalidad es enlazar este componente con el TSQLConnection. Normalmente será la primera propiedad que establezcamos, sencillamente abriendo la lista asociada en el Inspector de objetos y eligiendo entre las conexiones disponibles.

De TCustomSQLDataSet derivan cuatro componentes que encontramos en la página dbExpress de la Paleta de herramientas:

- TSQLTable: Adecuado cuando se quiere obtener el contenido de una tabla al completo, todas sus filas y columnas. El nombre de la tabla se indicará mediante la propiedad TableName que, en fase de diseño, tendrá asociada una lista desplegable con todas las tablas disponibles. La tabla se abrirá al dar el valor True a la propiedad Active, lo cual permite comprobar su funcionamiento durante el diseño.
- TSQLQuery: Útil para ejecutar una consulta arbitraria sobre la base de datos para, por ejemplo, seleccionar parte de las columnas y filas de una o más tablas. Sus propiedades características son SQL y Params. La primera es una lista de cadenas de caracteres en la que se introducirá la consulta, ya sea durante el diseño o en ejecución, mientras que la segunda contiene objetos TParam que actúan como parámetros sustituibles sobre la consulta, lo cual permite usar una misma estructura de consulta sobre diferentes tablas, cambiando el filtro de selección de filas, etc.

---

67 *Data Definition Language.* Es el subconjunto del lenguaje SQL formado por las sentencias empleadas para manipular los objetos de una base de datos: definición de nuevas tablas, creación de índices, etc., en contraposición a las sentencias de manipulación de datos que se agrupan bajo la denominación DML (*Data Manipulation Language*).

- TSQLStoredProc: Solamente nos servirá si en nuestra base de datos hay definidos procedimientos almacenados. Tras conectarlo con el TSQLConnection, la propiedad StoredProcName ofrecerá una lista de los procedimientos existentes para elegir uno. Dependiendo del RDBMS es posible que también se requiera un nombre de paquete en la propiedad PackageName. Los parámetros que precise el procedimiento se entregarán mediante la propiedad Params, como en el caso del componente TQuery.
- TSQLDataSet: Este componente es una generalización de los tres anteriores, en realidad de cualquier conjunto de datos que pueda obtenerse a través de la ejecución de un comando sobre el RDBMS.

Este último componente será el que utilicemos preferentemente dada su flexibilidad, tratando de manera homogénea tablas, consultas y procedimientos almacenados.

NOTA

En realidad dbExpress sigue incluyendo los componentes TSQLTable, TSQLQuery y TSQLStoredProc por compatibilidad con los métodos de acceso a datos heredados de versiones previas de Delphi, a fin de facilitar la transición a los desarrolladores.

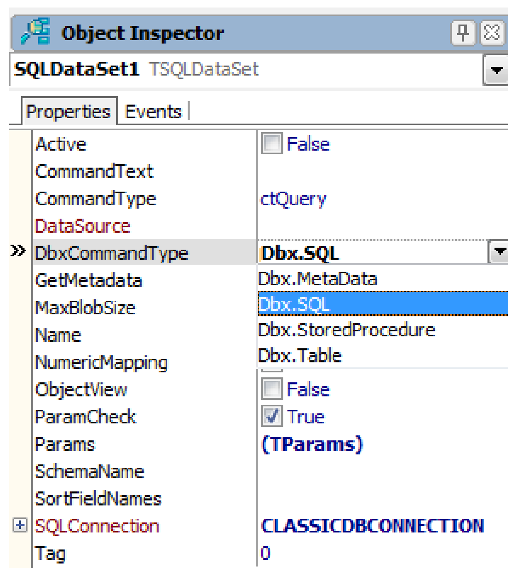
El componente TSQLDataSet

Al igual que todos los derivados de TSQLCustomDataSet el componente TSQLDataSet cuenta con las propiedades SQLConnection y Active. Con la primera vincularemos este componente con el de conexión, facilitando el envío de los comandos y recuperación de los resultados de su ejecución. La segunda será la que desencadene la operación, para lo cual antes es necesario configurar propiedades como CommandType y CommandText, entre otras.

Con un TSQLDataSet podemos abrir una tabla, ejecutar una consulta o un procedimiento almacenado, entre otras acciones. Dependiendo de la operación a realizar asignaremos a CommandType un valor u otro, siendo cuatro los disponibles: ctTable, ctQuery, ctStoredProc y ctServerMethod.

La propiedad CommandType opera conjuntamente con otra llamada DbxCommandType, de forma que la asignación de un valor a la primera modifica la segunda y viceversa. La diferencia es que la primera es de tipo enumerado, por lo que únicamente puede tomar los valores ya mencionados, mientras que la segunda almacena una cadena con el nombre del tipo de comando, lo cual ofrece mucha más flexibilidad.

Aunque DbxCommandType también cuenta con una lista desplegable[68] asociada en el Inspector de objetos, y normalmente varios de los valores serán equivalentes a los disponibles en CommandType, en la práctica podemos introducir cualquier nombre de comando válido, es decir, que sea reconocido por el controlador dbExpress y traducido adecuadamente al lenguaje del RDBMS. La imagen inferior muestra los cuatro valores existentes para DbxCommandType cuando se trabaja sobre una base de datos Firebird.



68 Los valores disponibles en esta lista son recuperados por el componente TSQLDataSet de manera interactiva a través de la conexión establecida con la base de datos, por lo que es imprescindible asignar primero dicha conexión a la propiedad SQLConnection. Será el método GetCommandTypes del componente TSQLConnection el encargado de facilitar los tipos de comandos existentes. Siempre serán al menos cuatro: DbxSQL, DbxStoredProcedure, DbxTable y DbxMetaData, a los que se añadirán otros específicos del controlador obtenidos mediante el procedimiento DerivedGetCommandTypes.

Una vez establecido el tipo de comando que se desea emplear hemos de asignar a la propiedad CommandText el contenido adecuado: nombre de la tabla o del procedimiento almacenado, texto de la consulta a ejecutar, etc. En caso necesario la propiedad Params aportará los valores actuales para los parámetros del procedimiento almacenado o los parámetros sustituibles usados en la consulta.

El caso de uso más simple para un TSQLDataSet es la apertura de una tabla: basta con asignar ctTable a CommandType y el nombre de la tabla a CommandText. Esta última propiedad, siempre que exista un enlace con la conexión a través del componente TSQLConnection, ofrecerá una lista desplegable con las tablas disponibles. Si queremos establecer la tabla durante la ejecución basta con una asignación a dicha propiedad: una cadena de caracteres con el nombre de la tabla. Éste puede obtenerse a partir del método GetTableNames de TSQLConnection, al que deben facilitarse dos parámetros: un objeto TStrings en el que devolver la lista con los nombres de las tablas y un Boolean indicando si han de incluirse (True) o no (False) las tablas de sistema[69] de la base de datos.

La configuración de un TSQLDataSet para ejecutar un procedimiento almacenado es similar a la anterior, con la diferencia de que el valor asignado a CommandType será ctStoredProc y que en la propiedad CommandText se ofrecerá una lista con los procedimientos almacenados existentes en la base de datos. Si el procedimiento precisa parámetros éstos aparecerán en la propiedad Params que, a través del editor que tiene asociado, permite ver el nombre y tipo de cada parámetro así como establecer el valor a usar.

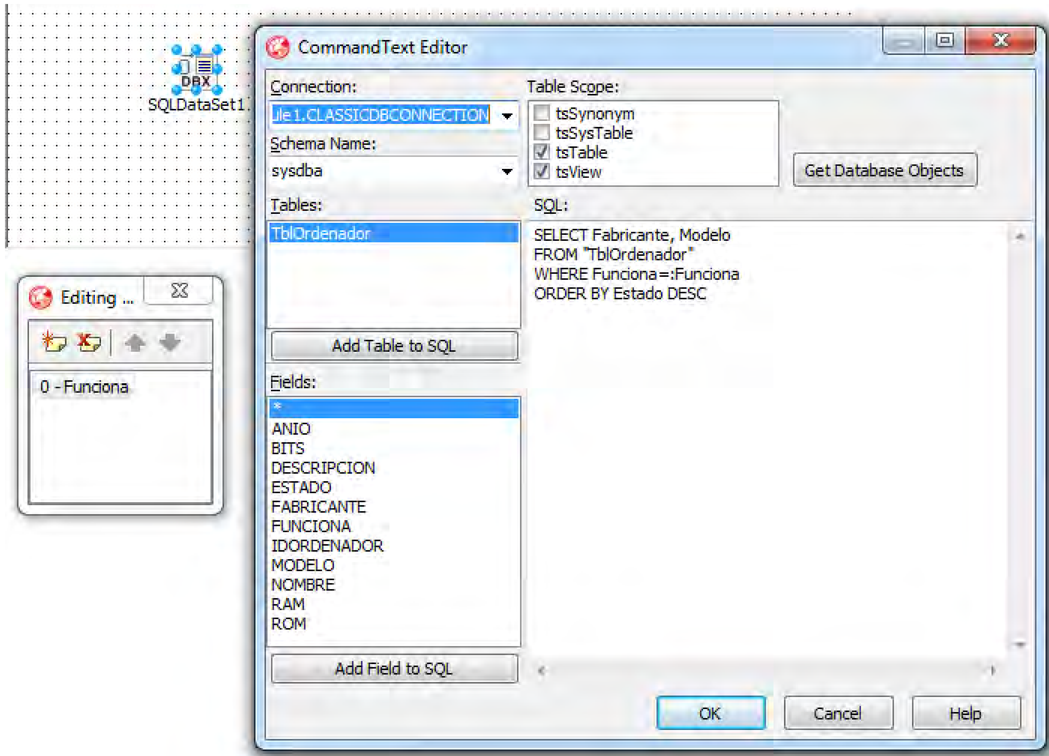
Consultas y parámetros

Cuando el componente TSQLDataSet se emplea para ejecutar una consulta sobre la base de datos el contenido de la propiedad CommandText es, aparte de más extenso, generalmente también más complejo. La mayoría de las veces la consulta implicará una selección y proyección de los datos, posiblemente un filtrado y también agrupamientos y ordenación.

---

69 Aparte de las tablas que definamos para almacenar los datos propiamente dichos los RDBMS recurren a un conjunto propio de tablas, conocidas como *de sistema*, en las que conservan meta-información de la base de datos: qué tablas existen, qué columnas tiene cada tabla, etc.

Por ello cuando se asigna el valor `ctQuery` a CommandType la lista desplegable de `CommandText` deja paso a un cuadro de diálogo específico llamado CommandText Editor (véase la imagen inferior). En éste normalmente ya se encontrará seleccionada la conexión y aparecerán las tablas y vistas que contenga la base de datos, pero podemos cambiar la configuración inicial y usar el botón Get Database Objects para actualizar la información ofrecida.



El elemento central de esta ventana es el recuadro de texto SQL en el que, con ayuda de la lista de tablas y de campos, habremos de escribir la consulta a ejecutar. Un ejemplo podría ser el siguiente:

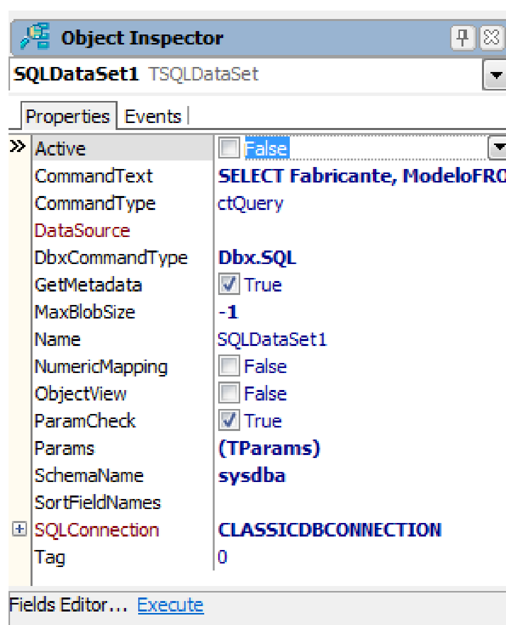
```
SELECT Fabricante, Modelo
FROM TblOrdenador
WHERE Funciona=:Funciona
ORDER BY Estado DESC
```

La consulta introducida en el recuadro de texto se guardará en la propiedad `CommandText` al hacer clic en el botón OK para cerrar la ventana.

En la consulta se ha destacado en negrita el elemento `:Funciona` que, al ir precedido de dos puntos, es identificado por dbExpress como un parámetro sustituible de la consulta. Su finalidad es activar un filtro comparando la consulta `Funciona` de la tabla de ordenadores con un valor que es necesario establecer antes de ejecutar la consulta. Para ello, en fase de diseño, abriremos el editor asociado a la propiedad `Params` y allí encontramos el citado parámetro. Lo seleccionamos y usamos el Inspector de objetos para establecer su tipo, en este caso `String`, y su contenido, que sería `True` o `False`.

> NOTA
>
> El tipo de los parámetros se obtendrá de la base de datos cuando el comando haga referencia a un procedimiento almacenado, pero en el caso de las consultas con parámetros sustituibles ese tipo ha de indicarse a través del objeto `TParam` que representa al parámetro.

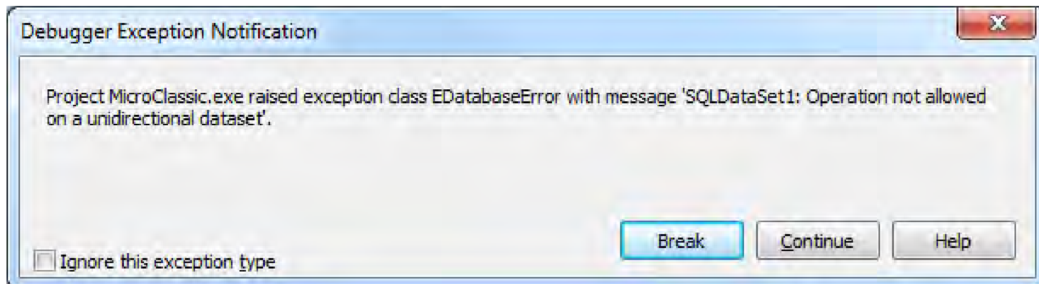
Definida la consulta y asignado valor al parámetro, para comprobar si la sentencia SQL resultante es válida no hay más que usar la opción Execute que aparece en la parte inferior del Inspector de objetos (véase la imagen inferior). Si no se genera un mensaje de error es que todo va bien.



Navegación y actualización

Todos los derivados de TCustomSQLDataSet, entre ellos TSQLDataSet, son conjuntos de datos unidireccionales. Esto significa que una vez abierta la tabla o ejecutada la consulta el componente recuperará un paquete de filas del conjunto de datos resultante, permitiendo recorrerlo secuencialmente hasta llegar al último registro, momento en que se transferirá desde el servidor el siguiente paquete de filas y así sucesivamente hasta llegar al final de los resultados.

Por tanto podemos usar un TSQLDataSet si únicamente necesitamos hacer un recorrido de principio a fin sobre los datos, por ejemplo para elaborar un informe u obtener subtotales y totales. Aunque el diseñador de Delphi no nos impedirá que enlacemos un TDataSource con el TSQLDataSet y creemos los LiveBinding necesarios para vincular la base de datos con una interfaz de usuario, en cuanto intentemos navegar por éstos obtendremos un error como el mostrado en la imagen inferior.



Además de ser unidireccional, el conjunto de datos obtenido es solamente de lectura y, en consecuencia, no podemos modificar el contenido de una fila, eliminar filas o agregar otras nuevas. Éstas son operaciones a realizar mediante comandos que den lugar a las habituales sentencias UPDATE, DELETE e INSERT del lenguaje SQL.

Tanto para facilitar la navegación por los datos como la actualización de éstos, especialmente a través de una interfaz gráfica de usuario con LiveBindings, el componente TSQDataSet necesita ayuda (en realidad no es su función). La solución es consiste en delegar esas tareas en otros componentes más especializados.

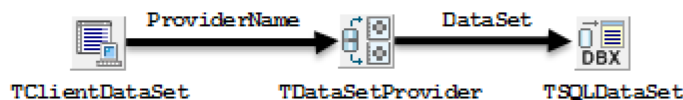
Almacenamiento local de datos y cambios

El componente que necesitamos para almacenar localmente en nuestra aplicación la información procedente de la base de datos, haciendo posible la navegación, lo conocimos en el capítulo previo: `TClientDataSet`. Ya sabemos cómo conectarlo con una interfaz de usuario a través de LiveBidings, por lo que solamente necesitamos conocer la manera en que la información se obtiene desde la base de datos y no de un archivo local como en una aplicación MyBase.

Además de los datos el componente `TClientDataSet` también almacena un registro de los cambios efectuados sobre esos datos, concretamente en una propiedad llamada `Delta`. Ésta empaqueta información relativa a cada registro insertado, eliminado o modificado en el conjunto de datos, ya sea a través de una interfaz de usuario como la de `MicroClassic` (basada en LiveBindings) o desde el código de la aplicación.

No es posible conectar directamente un `TClientDataSet` a un `TSQLDataSet`, ya que el primero no *sabe* cómo pedir al segundo los datos obtenidos del RDBMS y el segundo tampoco está preparado para interpretar el contenido de la citada propiedad `Delta` a fin de traducirla a las sentencias SQL correspondientes para actualizar la base de datos. El elemento que actuará como intermediario entre ambos será un componente `TDataSetProvider`. Lo encontramos en la página Data Access de la Paleta de herramientas de Delphi.

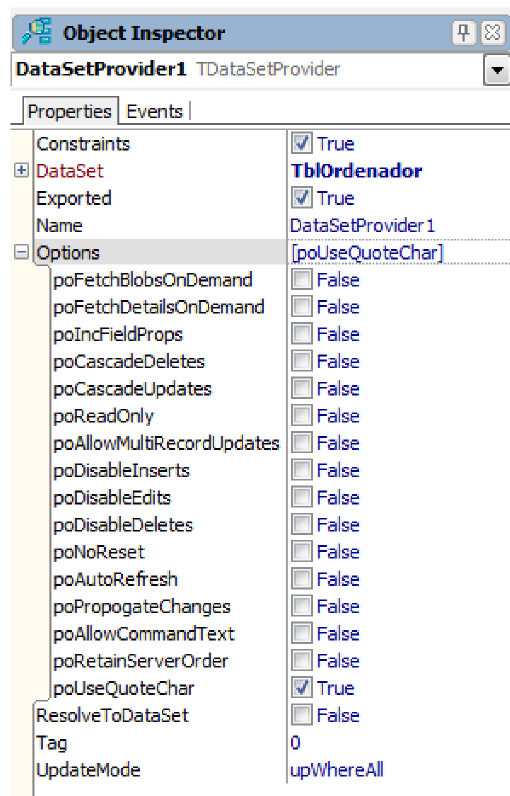
El esquema de conexión es el representado en la figura inferior: la propiedad `ProviderName` del componente `TClientDataSet` le enlaza con el `TDataSetProvider`. Éste, a su vez, se conecta con el `TSQLDataSet` mediante la propiedad `DataSet`. Como es habitual, durante la fase de diseño el Inspector de objetos ofrece listas desplegables asociadas a esas propiedades, de forma que no hay más que abrirlas y elegir entre los elementos disponibles el que corresponda.



Por una parte el `TDataSetProvider` obtendrá del `TSQLDataSet` las filas de datos resultantes de ejecutar la consulta, procedimiento almacenado o tabla,

según los casos. A continuación creará con los datos un paquete interpretable por el `TClientDataSet` y se lo entregará a dicho componente. Por otra parte, con el sentido de la comunicación a la inversa, el `TDataSetProvider` es el responsable de recoger el contenido de la propiedad `Delta` del `TClientDataSet` cuando se invoca al método `ApplyUpdates` de éste, interpretar las operaciones de inserción, borrado y actualización y traducirlas en comandos que serán enviados al `TSQLDataSet`.

La información empaquetada por el `TDataSetProvider` con destino al `TClientDataSet` se verá afectada por la propiedad `Options` del primer componente que, de manera indirecta, influye en las operaciones posibles sobre los datos. Podríamos, por ejemplo, permitir la modificación de datos e inserción de nuevas filas pero impedir los borrados activando la opción `opDisableDeletes`.



Generación de comandos de actualización

El paquete de cambios alojado en la propiedad `Delta` del `TClientDataSet` llegará al `TDataSetProvider` a través del método `ApplyUpdates` del primero. La conversión de ese paquete a comandos SQL comprensibles por el RDBMS se verá afectado fundamentalmente por dos propiedades de `TDataSetProvider`: `UpdateMode` y `ResolveToDataSet`.

Al construir las sentencias SQL de actualización de datos es necesario determinar las filas de las tablas que se verán afectadas, para lo cual se recurre a definir un filtro mediante la cláusula `WHERE`. Dicho filtro puede construirse acorde a tres enfoques distintos que, lógicamente, se corresponden con los tres valores que es posible asignar a la propiedad `UpdateMode`. Son los siguientes:

- `upWhereAll`: Es el valor tomado por defecto al crear un nuevo `TDataSetProvider`. Construye el filtro usando todas las columnas existentes, de manera que la actualización únicamente se completará si el contenido de la fila afectada no ha sufrido ningún cambio desde que fue leída de la base de datos.
- `upWhereChanged`: El filtro se construirá usando las columnas que formen parte de la clave y todas aquellas que hayan sido modificadas en el `TClientDataSet`, por lo que los cambios que pudieran haberse producido en otras columnas no serán tenidos en cuenta.
- `upWhereKeyOnly`: La selección de la fila afectada se realizará usando únicamente las columnas que formen la clave primaria de la tabla, sin comprobar si el contenido de las demás ha cambiado o no desde que recuperó de la base de datos.

Hemos de tener en cuenta que, al trabajar con un RDBMS que opera en una máquina independiente como servidor de datos, pueden darse accesos concurrentes a la misma información por parte de varios usuarios (sea con la misma aplicación o desde programas distintos). En consecuencia existe la posibilidad de que varias personas estén realizando cambios en las mismas filas de una tabla dada, un conflicto que es necesario resolver eligiendo una de las políticas que nos ofrece la propiedad `UpdateMode`.

La política conocida como *el último gana* se corresponde con el valor `upWhereKeyOnly`. Consiste básicamente en que si varias personas están editando la misma información solamente se conservarán los cambios que lleguen en último lugar al RDBMS, descartándose los anteriores. Se trata de

un enfoque que puede implicar pérdida de datos y, lo que es peor, que las personas que han llevado a cabo los cambios en primer lugar no tengan conocimiento de que su trabajo se ha perdido.

En contraposición a la anterior está la política más conservadora, representada por el valor `upWhereAll`, consistente en aplicar los cambios únicamente en caso de que la fila no se haya modificado desde su recuperación, es decir, tras asegurarse de que no hay nueva información en dicha fila que pudiera perderse.

> NOTA
>
> Es posible obtener un control más afinado de las columnas a comprobar en el filtro de actualización gracias a la propiedad `ProviderFlags` de la clase `TField`. Obtenido el objeto `TField` que representa a una cierta columna, dicha propiedad establece si ha de usarse o no en las sentencias de actualización, en los filtros de recuperación o conjuntamente con la clave primaria de la tabla.

Si no modificamos el valor por defecto de la propiedad `ResolveToDataSet` éste será `False`, indicando que los comandos de actualización derivados del registro de cambios no han de ser entregados al derivado de `TDataSet` al que esté conectado el `TDataSetProvider`, sino traducidos directamente a sentencias SQL a enviar al RDBMS a través de la conexión indicada por `SQLConnection`. Es el método a utilizar cuando se usan conjuntos de datos unidireccionales solamente de lectura, como es el caso de `TSQLDataSet`.

Partiendo de esa configuración, el componente `TDataSetProvider` se encarga internamente de crear un objeto `TSQLResolver` que, a la postre, será el que genere las sentencias SQL a enviar al RDBMS. Dicho objeto cuenta entre otros con los métodos `GenSelectSQL`, `GenInsertSQL`, `GenDeleteSQL` y `GenUpdateSQL` que, como es fácil suponer, preparan la sentencia SQL de selección, inserción, borrado y actualización de datos. Todo este trabajo es transparente para nosotros que, por regla general, nos limitaremos a llamar al método `ApplyUpdates` del `TClientDataSet` y gestionar los conflictos que pudiesen surgir en la actualización según la política elegida.

Resolución de conflictos de actualización

Acabamos de ver cómo la llamada al método ApplyUpdates de un componente TClientDataSet desencadena toda una secuencia de operaciones: envío del paquete de cambios almacenado en la propiedad Delta al componente TDataSetProvider, creación por parte de éste de un objeto TSQLResolver, generación de las sentencias necesarias de actualización, borrado e inserción según los cambios registrados y transferencia al servidor de datos para su ejecución.

En ese instante, al llegar al RDBMS, puede surgir un conflicto en caso de que las filas afectadas hayan sido modificadas por un tercero y nuestra política de actualización no sea *el último gana*. En ese momento el sentido de la comunicación se invierte, de forma que el fallo comunicado por el RDBMS llega hasta el TDataSetProvider y desde éste al TClientDataSet, componente que se encargará de producir el evento OnReconcileError. El objetivo es darnos una oportunidad para resolver el conflicto de manera que no se pierda información alguna, ni la de la propia aplicación ni la que se ha cambiado en la base de datos.

El método asociado a ese evento recibirá cuatro parámetros, tal y como puede apreciarse en la cabecera del siguiente fragmento de código:

```
procedure TDataModule1.ClientDataSet1ReconcileError(
  DataSet: TCustomClientDataSet; E: EReconcileError;
  UpdateKind: TUpdateKind; var Action: TReconcileAction);
begin
 ...
end;
```

Normalmente lo primero que nos interesará será saber qué operación ha fallado, para lo cual consultaremos el argumento UpdateKind. Los valores posibles son tres: ukModify, ukDelete y ukInsert, indicando que no ha podido completarse una actualización, borrado o inserción[70], respectivamente. Del parámetro E, de tipo EReconcileError, podemos recuperar el código concreto del error que se ha producido y el mensaje asociado, algo no demasiado útil salvo que vayamos a mostrarlo directamente al usuario.

---

70 Una operación de inserción raramente provocará un fallo salvo que se infrinja una restricción de unicidad, por ejemplo al duplicar el valor asignado a las columnas que forman la clave primaria.

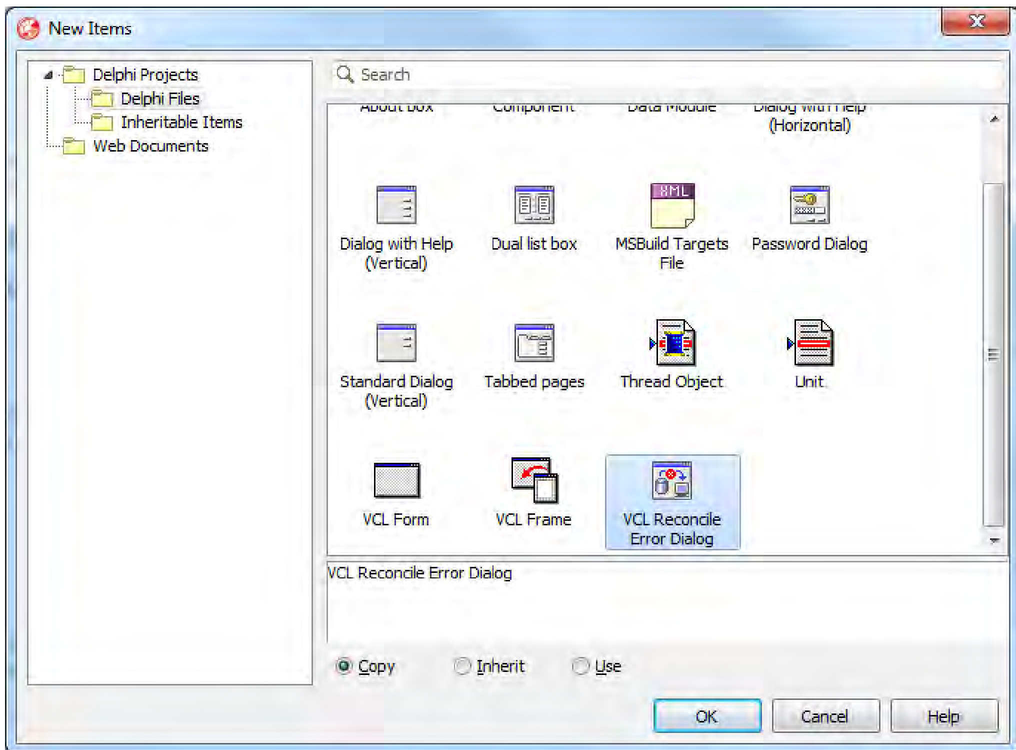
Además de la operación también necesitaremos saber qué información es la que contiene actualmente la base de datos, a fin de compararla con la que estábamos intentando almacenar y posiblemente también la que tenía originalmente. El parámetro DataSet recibido por el método anterior almacena en las propiedades OldValue, CurValue y NewValue los valores originales de la fila afectada por el fallo, los que tiene actualmente en la base de datos y los que nosotros queríamos asignarle. Es todo lo que necesitamos para, por ejemplo, combinar el contenido actual del RDBMS y las modificaciones hechas por el usuario, ya sea tras mostrarle a éste información sobre el problema o de manera automatizada.

El evento OnReconcileError espera que el método asociado asigne al último de los parámetros, llamado Action y de tipo TReconcileAction, un valor indicando la acción que debe llevarse a cabo en respuesta al conflicto. TReconcileAction es una enumeración de media docena de valores, dependiendo del usado podremos tratar el problema de las siguientes formas:

- raCorrect: Se escribirán en la base de datos los valores que había enviado nuestra aplicación (almacenados en la propiedad NewValue del parámetro Dataset) sobrescribiendo los cambios de otros.
- raMerge: Los cambios detectados en la base de datos se combinarán con los propuestos por nuestra aplicación, resultando en una nueva fila que será la que finalmente se conserve.
- raAbort: Se cancelará la operación de resolución del conflicto, de manera que la fila quedará como se encuentre actualmente en la base de datos sin llevar a cabo ninguna acción adicional.
- raSkip: La fila que ha generado el fallo quedará como se encuentra en la base de datos, pero la actualización que no ha podido aplicarse permanecerá en el registro de cambios quedando como pendiente.
- raCancel: Se descartarán todos los cambios que ha sufrido la fila, tanto los propios como los de terceros, de forma que vuelva a tener el mismo contenido que cuando se recuperó de la base de datos (almacenados en la propiedad OldValue del parámetro Dataset).
- raRefresh: Similar a la operación anterior pero descartando únicamente los cambios propios, de forma que la fila tomará los valores que tiene actualmente en la base de datos y que reflejan los cambios efectuados por terceros.

Determinadas acciones tienen sentido únicamente cuando la operación que ha generado el conflicto ha sido de tipo `ukModify`, una actualización de datos, como es el caso de `raMerge` o `raRefresh`.

Es posible construir una interfaz genérica que se encargue de los conflictos de actualización de datos, un formulario que informe al usuario de la aplicación de cuál es el problema, cuál la información original, actual y que se quería introducir y le pregunte qué desea hacer. Delphi XE2 incluye de hecho un formulario con ese fin, si bien está disponible únicamente para aplicaciones basadas en VCL (véase la imagen inferior).



Ese formulario aparte de la interfaz contiene una importante cantidad de código especializado en el tratamiento de los conflictos de conciliación de datos, código que puede servirnos perfectamente como plantilla de partida con independencia de que nuestro proyecto sea VCL o FMX ya que los componentes de datos son exactamente los mismos y también el lenguaje de programación.

El componente TSimpleDataSet

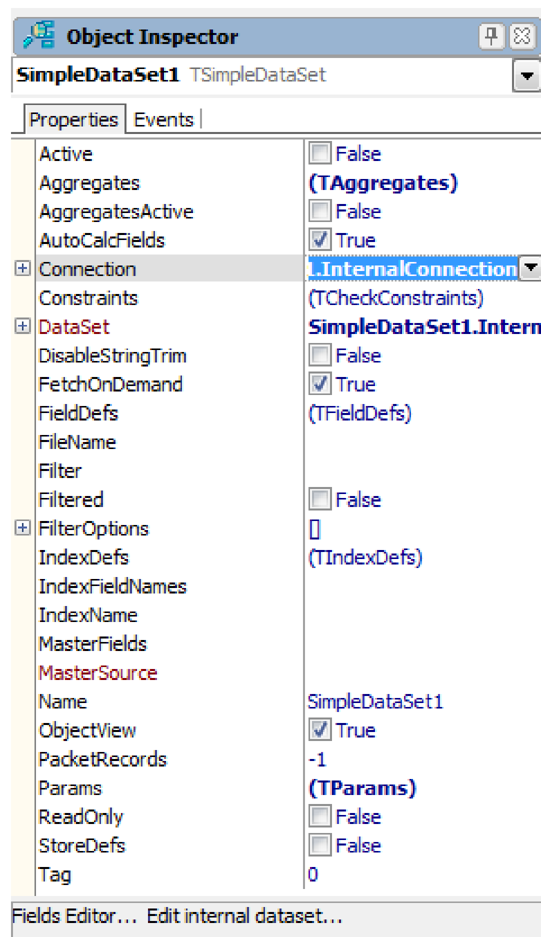
Siempre que necesitemos ofrecer en un proyecto opciones de navegación y edición de un conjunto de datos, y lo habitual es que en una misma aplicación haya múltiples de ellos, tendremos que usar un `TClientDataSet` conectado a un `TDataSetProvider` que, a su vez, estará conectado a un `TSQLDataSet` y éste a un `TSQLConnection`. Se trata de un cuarteto de componentes prácticamente inseparables en la mayoría de escenarios, por lo que tendría sentido combinar su funcionalidad en uno solo.

Esto es precisamente lo que nos ofrece el componente `TSimpleDataSet` que encontramos al final de la página dbExpress de la Paleta de herramientas: un derivado de `TDataSet` con almacenamiento local de datos y cambios, como `TClientDataSet`, con objetos internos que se encargan de la conexión, generación de comandos y selección de datos. En cierta manera es una combinación de los cuatro citados antes.

Al inspeccionar las propiedades de un componente `TSimpleDataSet` (véase la imagen de la página siguiente) comprobaremos que hay una llamada `Connection`. Es una referencia a un componente de conexión interno que no tenemos que crear explícitamente. En el Inspector de objetos no tenemos más que hacer clic en el botón que hay a la izquierda de la propiedad para acceder a sus miembros: `Driver`, `ConnectionName`, `Params`, etc., los mismos que en un `TSQLConnection`.

Análogamente la propiedad `DataSet` mantiene una referencia a un objeto `TInternalSQLDataSet`, que no es más que un `TSQLDataSet` que ya tiene asociado un `TDataSetProvider`. Cuenta con las mismas propiedades `CommandType`, `CommandText`, `Params`, etc., que `TSQLDataSet`, pudiendo acceder a ellas como en el caso anterior, desplegando la propiedad principal, o bien mediante la opción Edit internal dataset que aparece en el parte inferior del Inspector de objetos.

La principal ventaja de usar este componente es, como su propio nombre indica, la sencillez, ya que reduce a una cuarta parte el número de elementos necesarios para operar sobre el contenido de una tabla de la base de datos. Si en un módulo de datos tenemos componentes para trabajar sobre una docena de tablas o consultas esa reducción será importante y nos facilitará el trabajo.



Lógicamente, al tratarse de la combinación de varios componentes, TSimpleDataSet ofrece un conjunto de eventos que recoge los que sus diferentes integrantes. Así tenemos eventos relacionados con la conexión: BeforeConnect, OnLogin, etc.; otros pertenecientes al tratamiento de los datos: BeforeOpen, AfterRefresh u OnCalcFields, y finalmente todos los propios de un TClientDataSet: OnNewRecord, BeforeApplyUpdates y OnReconcileError entre ellos.

MicroClassic colaborativo

Al finalizar el capítulo previo nuestro proyecto `MicroClassic` ya era una interfaz de usuario conectada a una base de datos MyBase, es decir, almacenaba localmente la información en un archivo XML usando un componente `TClientDataSet`. Si esta aplicación fuese hipotéticamente usada por varias personas lógicamente cada una vería sus propios datos, no los de los demás.

Supongamos que la aplicación tiene el objetivo de crear una base de datos centralizada con datos de microordenadores, de forma que múltiples personas puedan tanto agregar nuevos datos como ampliar y corregir los ya existentes. Sería un proyecto de trabajo colaborativo, lo cual implica que la información no puede residir en el ordenador de cada uno.

Lo primero que tendremos que hacer será habilitar una máquina para que funcione como servidor de datos, instalando en ella el software RDBMS que convenga. La identificación de esa máquina: su nombre en la red o su dirección IP, serán imprescindibles para configurar la conexión de la aplicación a la base de datos. Asimismo habrá que instalar el software cliente del RDBMS en cada uno de los ordenadores en los que vaya a utilizarse la aplicación.

En nuestro caso elegimos como servidor de datos el software Firebird. Se trata de un RDBMS de código abierto, muy maduro y que se integra perfectamente con Delphi.

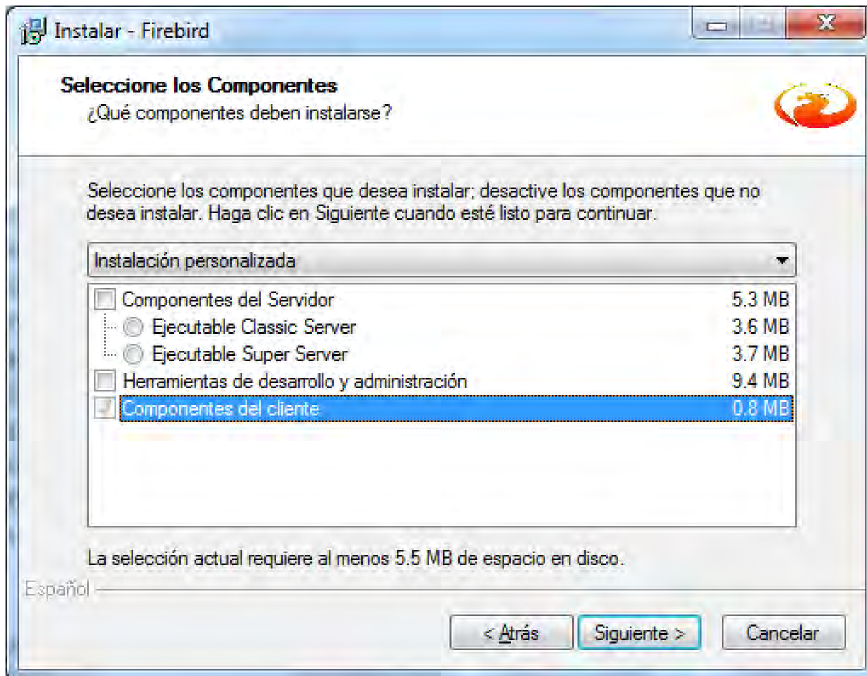
Instalación del RDBMS

Una vez obtenida de `http://www.firebirdsql.org` la última versión de Firebird para el sistema operativo con que trabaje el servidor: Win32, Win64, Linux o MacOS X, la instalación se reduce a seguir los pasos de un asistente aceptando la licencia, facilitando la ruta de instalación y seleccionando el tipo de instalación que se quiere realizar.

En la mayoría de los pasos los valores ofrecidos por defecto son los adecuados. En caso de duda lo más recomendable es examinar la documentación del RDBMS, dado que los parámetros podrían cambiar de unas versiones a otras.

El siguiente paso será instalar la parte cliente del RDBMS en cada ordenador de usuario. En el caso de Firebird en Windows podríamos limitarnos a incluir en la propia carpeta de nuestra aplicación el módulo `fbclient.dll`, no requiriéndose entradas en el registro ni elementos adicionales siempre que el sistema ya incluya una serie de dependencias que, por regla general, se instalan con las propias actualizaciones del sistema operativo.

No obstante, si queremos asegurar una instalación sin problemas lo recomendable es usar el propio instalador de Firebird, desmarcando en el paso Seleccione los Componentes todos los elementos menos Componentes del cliente, como puede apreciarse en la imagen inferior. Será una instalación rápida e incluirá los elementos adecuados según la versión del sistema.



> NOTA
>
> Además del parte servidor del RDBMS también necesitaremos alguna herramienta de administración en la máquina de desarrollo, como puede ser FlameRobin.

Creación de la base de datos

Disponiendo ya del RDBMS el paso siguiente será diseñar y crear la base de datos. Sin la base de datos el desarrollo de la aplicación no podrá completarse. En este sencillo ejemplo la base de datos contiene una sola tabla con la información de los ordenadores, tabla que podemos definir desde el propio entorno de Delphi, con el Explorador de datos; a través de una herramienta de administración como FlameRobin o directamente desde el intérprete SQL de Firebird[71].

Tras crear la base de datos vacía, y definir una conexión en el Explorador de datos, la opción New Table de éste da paso a un diseñador como el de la imagen inferior. En él indicaremos el nombre y tipo de cada columna, su longitud y estableceremos las restricciones `NOT NULL` (Nullable) y de clave primaria. En este caso no podrán quedar sin valor ni el identificador de ordenador ni el nombre, siendo el primero la clave primaria de la tabla. Esa columna contendrá sencillamente un número entero secuencial y único.

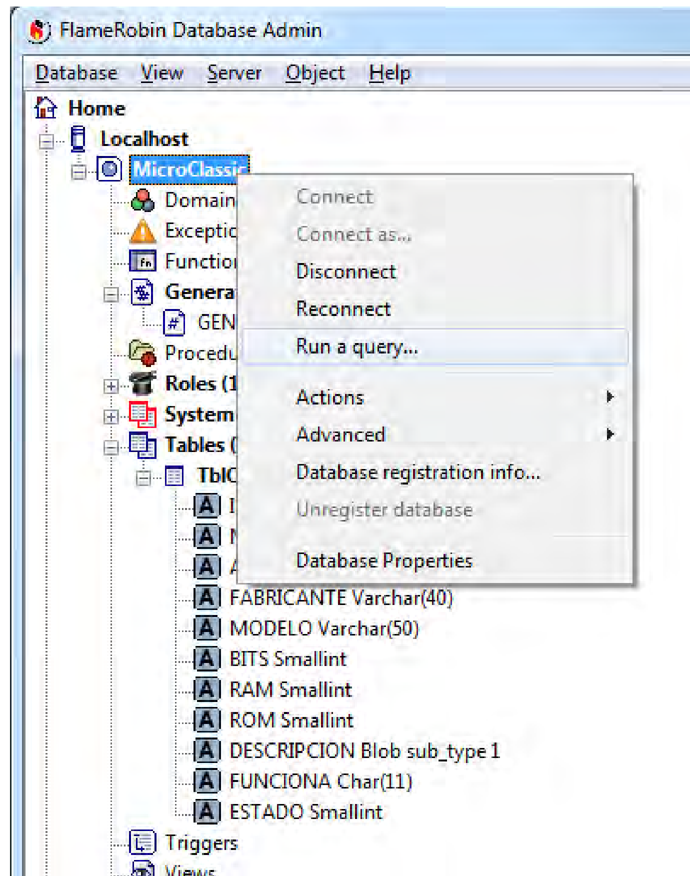
Principal | MicroClassicDB | Table Design:"TblOrdenador" | MicroClassic | InfodbExpress

Name	Data Type	Precision	Scale	Nullable	Primary Key
IDORDENADOR	NUMERIC	10	0	☐	☑
NOMBRE	VARCHAR	50	0	☐	☐
ANIO	SMALLINT	0	0	☑	☐
FABRICANTE	VARCHAR	40	0	☑	☐
MODELO	VARCHAR	50	0	☑	☐
BITS	SMALLINT	0	0	☑	☐
RAM	SMALLINT	0	0	☑	☐
ROM	SMALLINT	0	0	☑	☐
DESCRIPCION	BLOB	8	0	☑	☐
FUNCIONA	CHAR	11	0	☑	☐
ESTADO	SMALLINT	0	0	☑	☐
*				☐	☐

71 Tras la instalación de Firebird encontraremos en el menú Inicio o recurso equivalente de nuestro sistema operativo un elemento llamado Firebird ISQL Tool que da paso a una consola de texto que, de manera interactiva, permite introducir y ejecutar sentencias SQL.

La mayoría de herramientas de administración con que cuentan los RDBMS ofrecen una interfaz parecida a la anterior para el diseño de las tablas y consultas.

Podemos, no obstante, introducir las sentencias SQL para crear la tabla en un archivo de texto, guardarlo y después ejecutarlo como un guión, ya sea desde la consola ISQL de Firebird, con la opción Run a query de FlameRobin (véase la imagen inferior) o similar en otras herramientas de administración.



El contenido de ese archivo de texto sería, en nuestro caso concreto, una sentencia CREATE TABLE como la mostrada a continuación:

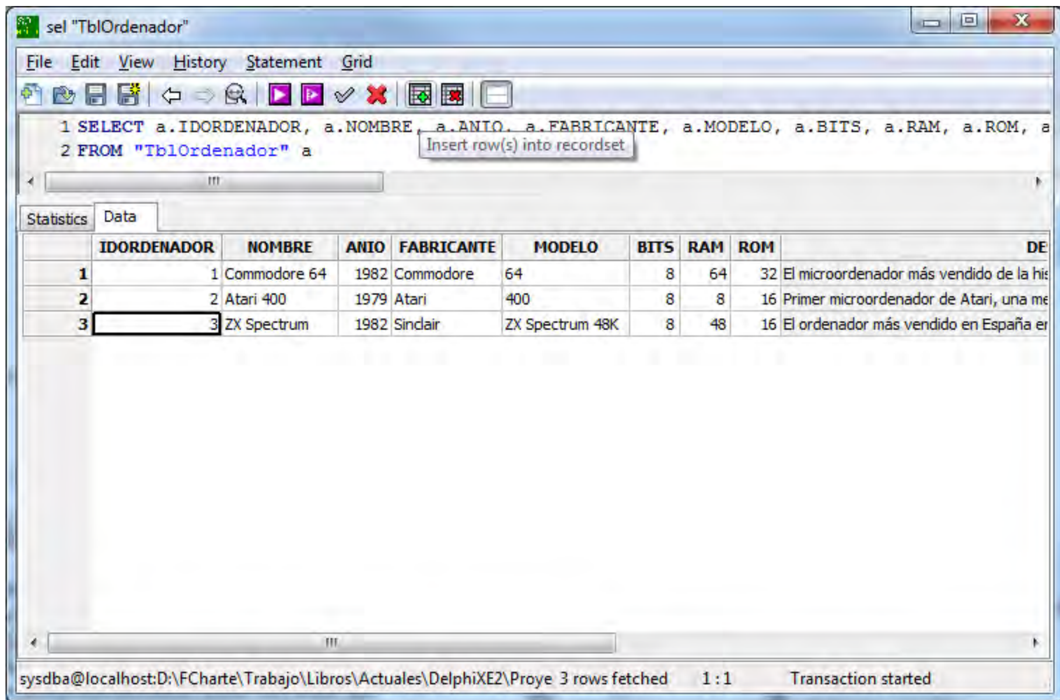
```
CREATE TABLE TblOrdenador
(
```

```
   IDOrdenador Numeric(10,0) NOT NULL,
   Nombre Varchar(50) NOT NULL,
   Anio Smallint,
   Fabricante Varchar(40),
   Modelo Varchar(50),
   Bits Smallint,
   RAM Smallint,
   ROM Smallint,
   Funciona Varchar(11),
   Estado Smallint,
   Descripcion Blob sub_type 1,
   PRIMARY KEY (IDOrdenador)
);
```

La ventaja de esta última opción es que si algo va mal con la tabla, por ejemplo el tipo de una columna no es el adecuado, podemos eliminarla con la sentencia `DROP TABLE TblOrdenador` y volver a crearla a partir del guión, tras introducir los cambios que convengan, sin repetir todos los pasos que requiere el diseño manual con las herramientas antes indicadas que sería mucho más lento.

Como puede verse, las columnas `IDOrdenador` y `Nombre` tienen la restricción `NOT NULL` por lo que nunca podrán quedar vacías. Además la primera actúa como clave primaria de la tabla, según indica la cláusula `PRIMARY KEY` del final. La columna `Descripcion` es de tipo `Blob` (*Binary Large Object*) subtipo 1, el adecuado para almacenar un texto y que pueda ser conectada a un componente `TMemo`. En cuanto a la columna `Funciona`, que se enlazará con un control `TCheckBox`, se ha definido como una secuencia variable de hasta 11 caracteres, de forma que el valor `Boolean` del citado control se convierta a cadena al almacenar y viceversa.

También podríamos introducir en el propio guión encargado de crear la tabla varias sentencias `INSERT` para agregar algunas filas, de forma que haya datos ya de partida que, durante el desarrollo de la aplicación, aparezcan en los controles de la interfaz de usuario a medida que vayan definiéndose los vínculos con la base de datos. Si lo preferimos podemos introducir esos datos iniciales de manera interactiva, ya sea desde el entorno de Delphi: con la opción Retrieve Data from Table, o bien usando la herramienta de administración que tengamos instalada en el sistema. En la figura de la página siguiente puede apreciarse la ventana abierta por la opción Select from de FlameRobin, con una barra de botones en la parte superior que nos permite agregar nuevas filas, modificar el contenido de las mostradas en la panel inferior, eliminarlas, etc.



Generadores de secuencias y su uso

La columna `IDOrdenador` que aparece en nuestra tabla no se corresponde con ninguno de los datos que teníamos originalmente en la aplicación `MicroClassic`. Se trata de una columna *artificial* que tiene como única finalidad asociar un identificador único a cada fila de datos, un valor que actuará como clave principal y permitirá diferenciar de manera inequívoca una fila del resto de las existentes en la tabla, lo cual es un requerimiento de diseño de bases de datos.

El problema que surge al definir una columna de este tipo es cómo establecer el número asociado a cada fila. Si se deja en manos del usuario final habrá que facilitarle al menos información sobre cuál es el último número utilizado, pero siempre se corre el riesgo de que introduzca uno ya en uso lo cual provocaría un fallo de duplicación en el momento de la inserción de la nueva fila.

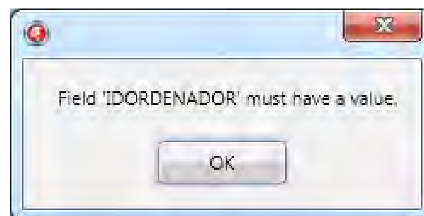
Una buena alternativa consiste en delegar en el propio RDBMS el trabajo de generar una secuencia de números enteros sin repeticiones. En el caso de Firebird podemos hacerlo con la sentencia `CREATE GENERATOR`, facilitando como único parámetro el nombre que deseamos dar al generador. Por ejemplo:

```
CREATE GENERATOR GEN_TblOrdenador_ID;
```

Incluso podríamos asegurarnos de que cada nueva fila de la tabla tome un valor de la secuencia, siempre que no se le haya asignado valor, creando un desencadenador (*trigger*) que se ejecute automáticamente antes de cada sentencia `INSERT`. En Firebird lo crearíamos con el siguiente guión SQL:

```
SET TERM ^ ;

CREATE TRIGGER TblOrdenador_BI FOR TblOrdenador
 ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
  IF (NEW.IDOrdenador IS NULL) Or (NEW.IDOrdenador <= 0)
THEN
    NEW.IDOrdenador = GEN_ID(GEN_TblOrdenador_ID, 1);
END^

SET TERM ; ^
```

El problema que plantea esta opción es que si las restricciones de la tabla son transferidas y comprobadas en el cliente, que es lo habitual, cuando el usuario intente agregar una fila dejando la columna `IDOrdenador` sin contenido recibirá un mensaje de error como respuesta (véase la imagen inferior), no llegando a enviarse el comando de actualización al RDBMS.

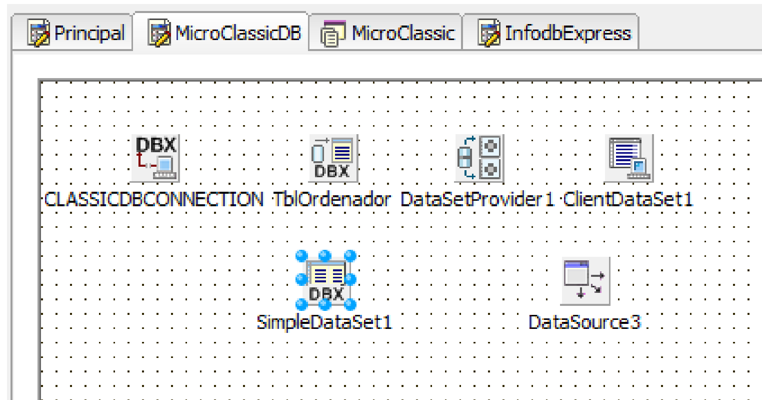


La solución pasa porque sea la aplicación cliente la que emplee el generador creado en el RDBMS para obtener el siguiente valor, asignándolo a la columna `IDOrdenador` de la nueva fila. Esto provocaría su aparición inmediata en el componente de la interfaz de usuario que estuviese vinculado a dicha columna a través de un LiveBinding.

Cambios en el módulo de datos

Nuestro proyecto cuenta con un módulo de datos, al que llamamos en su momento `MicroClassicDB`, en el que están alojados los componentes relacionados con el acceso a datos. Actualmente solo contiene un `TClientDataSet`, para el almacenamiento local en un archivo XML, y un `TDataSource` que facilita la conexión con la interfaz de usuario.

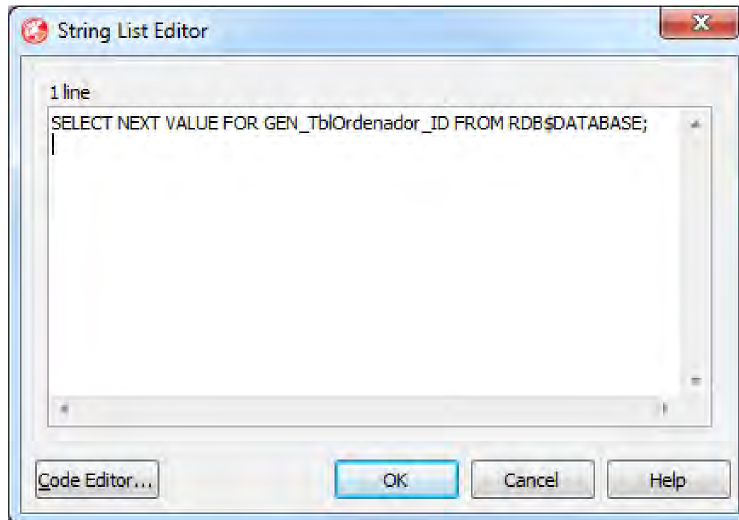
Para configurar la conexión a nuestra nueva base de datos tenemos, como ya sabemos, dos alternativas que quedan reflejadas en la imagen inferior: insertar un `TSQLConnection`, un `TSQLDataSet`, un `TDataSetProvider`, un `TClientDataSet` y realizar todas las conexiones necesarias o, en su lugar, insertar un `TSimpleDataSet` que se ocupe de todo. En cualquier caso tendremos que añadir también un `TDataSource`, enlazado con el `TClientDataSet` o el `TSimpleDataSet`, que será el que actúe como intermediario entre los datos y la interfaz de usuario.



> TRUCO
>
> Si optamos por la primera alternativa podemos ahorrarnos gran parte del trabajo tomando la tabla `TblOrdenador` desde el Explorador de datos y arrastrándola hasta el Módulo de datos.

Además de los anteriores también precisaremos un componente `TSQLQuery` (o un `TSQLDataSet`) que usaremos para solicitar a la base de datos el siguiente valor de la secuencia del generador `GEN_TblOrdenador_ID` que habíamos creado antes.

Tras agregar el TSQLQuery desplegamos su propiedad SQLConnection y lo enlazamos con TSQLConnection, la conexión a la base de datos es la misma. A continuación abrimos el editor de la propiedad SQL y escribimos la consulta que puede verse en la imagen inferior.



La sentencia SELECT NEXT VALUE FOR generador devuelve el valor del generador y lo incrementa. El único parámetro importante es el nombre del generador pero, como todas las cláusulas SELECT han de contar con la cláusula FROM, es necesario usar una tabla cualquiera que contenga una sola fila de datos. En Firebird hay varias tablas de sistema que cumplen con este requisito, siendo una de ellas RDB$DATABASE.

Con esta configuración el TSQLQuery ejecutaría la consulta, obteniendo el siguiente valor del generador, sin más que invocar al método Open para abrirlo. En ese momento encontraríamos el valor almacenado en la primera y única columna de la única fila resultante de la consulta, a la que podemos acceder mediante la propiedad Fields heredada de TDataSet. Ésta es una colección de objetos TField, correspondiente cada uno a una columna de resultados, indexada como una matriz. Podemos, por tanto, usar la referencia Fields[0] para obtener el TField que nos interesa y, por último, convertir su contenido al tipo Integer mediante el método AsInteger.

Hay que preguntarse cuál es el momento adecuado para obtener el valor del generador y asignarlo a la columna IDOrdenador. Habría que hacerlo en el

momento en que el usuario de la aplicación haga clic sobre el botón + del control TBindNavigator, pero éste se encuentra en el módulo correspondiente al formulario que actúa como interfaz, no en el módulo de datos. Alojar código de gestión de datos, que tendría que acceder al componente TSQLQuery, en un módulo de interfaz iría en contra de la cohesión de cada módulo e incrementaría el acoplamiento de éstos, lo cual nunca es recomendable.

Cuando se hace clic sobre el citado botón del TBindNavigator éste, que está conectado a través de un componente TBindScopeDB al TClientDataSet, pide a éste que agregue un nuevo registro en blanco. En ese momento el resto de los componentes, enlazados al mismo TBindScopeDB, se actualizan mostrando vacíos los controles de la interfaz e invitando al usuario que introduzca la nueva información. Cada vez que un TClientDataSet añade al conjunto de datos una nueva fila genera también un evento OnNewRecord. Éste sería el lugar adecuado para abrir la consulta, obtener el valor del generador, asignarlo a la columna IDOrdenador y cerrar la consulta. El código para llevar a cabo esa tarea sería el mostrado a continuación:

```
procedure TDataModule1.ClientDataSet1NewRecord(
                            DataSet: TDataSet);
begin
   with SQLQuery1 do begin
    Open;
    DataSet.FieldByName('IDORDENADOR').Value :=
                          Fields[0].AsInteger;
    Close;
   end;
end;
```

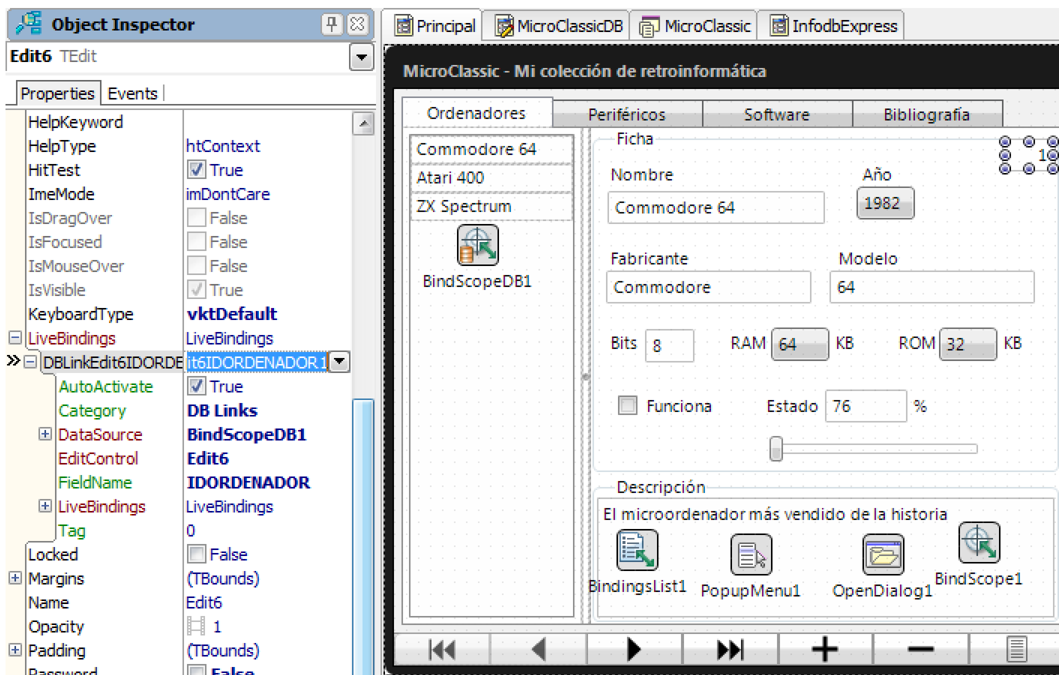
El método recibe como parámetro una referencia al derivado de TDataSet que ha generado el evento, en este caso podría ser el TClientDataSet o el TSimpleDataSet, dependiendo de la opción que eligiésemos anteriormente. En cualquier caso, la fila actual en ese conjunto de datos será la recién añadida y vacía. Para obtener el TField que corresponde a la columna IDOrdenador recurrimos al método FieldByName ya que, en principio, no sabemos el orden que se habrán agregado las columnas de la tabla y no deberíamos usar un índice que podría no ser el correcto.

Finalmente, en cuanto al módulo de datos afecta, agregaremos una llamada al método ApplyUpdates del TClientDataSet o TSimpleDataSet desde el método asociado al evento OnDestroy del propio módulo de datos. De esta forma nos aseguramos de que nunca queden cambios sin aplicar.

Cambios en la interfaz de usuario

Todos los elementos introducidos en el módulo de datos, que conectan la aplicación con la base de datos Firebird, no servirán de nada sino enlazamos la interfaz con el nuevo origen de datos. Para ello seleccionamos el componente BindScopeDB1, el TBindScopeDB que se agregó en el capítulo previo, desplegamos su propiedad DataSource y elegimos el TDataSource nuevo[72]. Con este simple cambio todos los controles que forman la interfaz ya mostrarán y permitirán editar la información almacenada en la base de datos que creábamos anteriormente en Firebird.

Lo siguiente será agregar a la interfaz un componente TEdit, podemos colocarlo en la parte superior derecha de la ficha (véase imagen inferior), definiendo un LiveBinding que lo enlace con la columna IDOrdenador. Si la tabla tiene datos veremos aparecer el valor de la primera fila.



72 También podríamos haber cambiado la propiedad DataSet del TDataSource que había originalmente en el módulo de datos, desvinculándolo del TClientDataSet usado con MyBase y enlazándolo con el correspondiente a Firebird. De esta forma no habría que cambiar nada en el TBindScopeDB.

ADVERTENCIA
Dado que el TEdit agregado a la interfaz servirá únicamente para mostrar el código asociado a cada ordenador, no para que el usuario lo introduzca o modifique, habría que dar el valor True a la propiedad ReadOnly. Teóricamente con esto debería bastar, en la documentación de FMX se indica que un TEdit con ReadOnly a True no permite la alteración del contenido por parte del usuario. Sin embargo esto no es cierto, a menos que se corrija en una posterior actualización del producto.
Hay múltiples formas de impedir los cambios en un TEdit aprovechando los distintos eventos con que cuenta. Una de ellas consiste en introducir en el método asociado al evento OnEnter una sentencia del tipo Self.SetFocused(Edit1), trasladando el foco al siguiente control de la interfaz. Al no poder tomar el foco, ni con el teclado ni con el ratón, será imposible alterar el contenido.

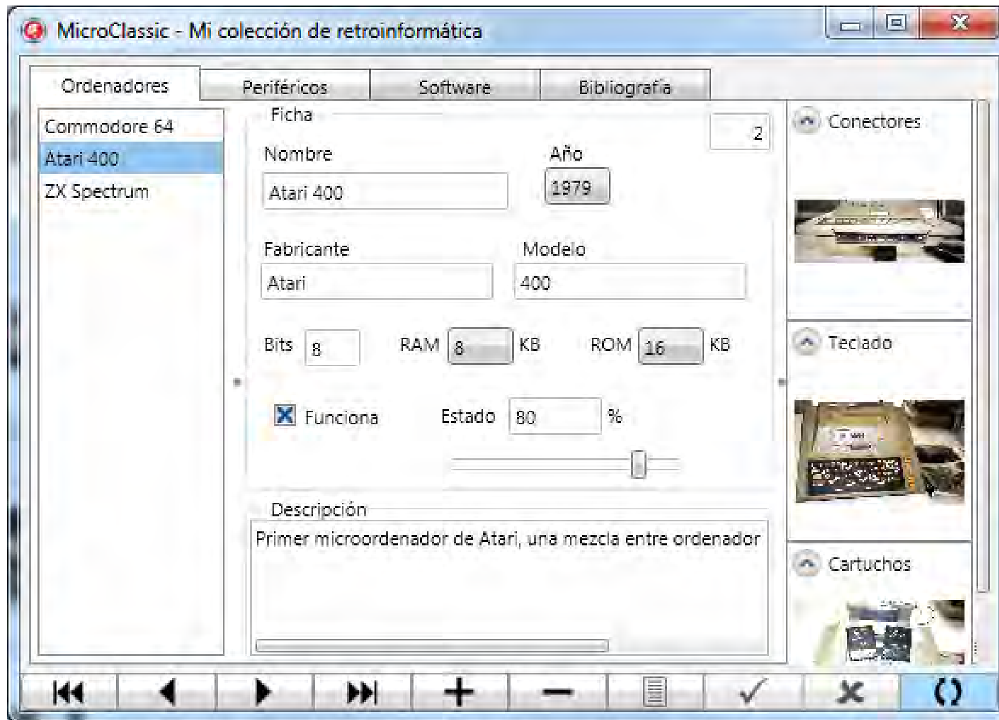
No necesitamos tocar nada más en la interfaz, tanto los controles que la forman como los LiveBinding que se definieron en su momento seguirán funcionando exactamente igual. Para ellos el cambio desde el TClientDataSet que almacena la información en un archivo XML a la nueva configuración en que se conecta con un RDBMS resulta totalmente transparente.

Al ejecutar el proyecto y navegar por las filas de datos veremos el identificador asociado a cada ficha, como se muestra en la imagen de la página siguiente. Si agregamos un nuevo registro el nuevo valor de la secuencia aparecerá en el TEdit y, en caso de haber introducido la llamada a SetFocused en su evento OnEnter, no podremos cambiarlo.

Mejoras a la aplicación

Aunque en su estado actual este proyecto funcionará perfectamente siempre que no haya más de un usuario concurrente, al ejecutar la aplicación desde varias máquinas y modificar los mismos datos comenzaremos a

encontrarnos con problemas. Para alcanzar una operativa correcta es necesario llevar a cabo las mejores que se detallan a continuación:



- Actualmente se llama al método `ApplyUpdates` únicamente desde el evento `OnDestroy` del módulo de datos, es decir, solo cuando se sale de la aplicación. Esto implica que todos los cambios efectuados por un usuario no aparecerán en la base de datos hasta que cierre el programa, tiempo durante el cual otras personas podrían estar modificando, eliminando y agregando filas. Es necesario enviar los cambios al RDBMS cuando se producen, aprovechando para ello los distintos eventos de `TClientDataSet`. También se podría comprobar periódicamente si hay cambios pendientes de aplicar y, en caso afirmativo, llamar al método `ApplyUpdates`, todo ello desde el propio módulo de datos.
- Para que los cambios realizados por otros usuarios en la base de datos se reflejen en la aplicación, de forma que sean visibles para los demás, es necesario actualizar el `TClientDataSet`. El componente

TBindNavigator cuenta con un botón, el último, que permite actualizar los datos a demanda del usuario. No obstante es recomendable actualizarlos también tras cada llamada a ApplyUpdates, para lo cual no hay más que usar el método Refresh[73] de TClientDataSet. También existe la posibilidad de actualizar únicamente el registro visible actualmente en la interfaz mediante el método RefreshRecord en lugar de Refresh.

- Por último habría que tratar los potenciales errores de actualización agregando el código apropiado al método asociado al evento OnReconcileError que se explicó en un punto previo. Puede optarse por un tratamiento genérico, como el del formulario VCL que se indicó antes, o bien sencillamente por informar al usuario de que se ha producido un fallo mostrándole los nuevos datos, cancelando la actualización con la acción raRefresh pero permitiéndole recuperar los datos que hubiese introducido antes del problema, obteniéndolos del parámetro DataSet y conservándolos.

Realizadas estas mejoras la aplicación estaría preparada para operar sin demasiados problemas en un entorno con múltiples usuarios.

Controladores dbExpress

A la hora de distribuir una aplicación que conecta con bases de datos no bastará con tener en cada ordenador el software cliente del RDBMS que corresponda, el programa también necesitará los módulos correspondientes al controlador dbExpress que proceda.

El primer requerimiento es agregar a la cláusula uses del módulo de datos una referencia al módulo Data.DBXFirebird. Esto es imprescindible para que la aplicación pueda cargar el controlador dbExpress de Firebird, alojado en una DLL externa llamada dbxfb.dll que hay que distribuir con el resto de archivos que formen la aplicación.

---

73 Es importante llamar a este método únicamente en caso de que ApplyUpdates no devuelva error alguno, ya que en caso contrario no será posible actualizar el estado del TClientDataSet y se producirá una excepción. ApplyUpdates devuelve un valor entero indicando el número de errores encontrados, si es mayor que 0 no deberíamos llamar a Refresh.

También será necesario incluir en el paquete a distribuir el módulo midas.dll, a menos que hayamos añadido a la cláusula uses una referencia a MidasLib (tal y como se explicó en un capítulo previo).

Lógicamente si usamos un RDBMS que no sea Firebird tanto la DLL a distribuir como el módulo a referenciar en la cláusula uses será otro. En http://docwiki.embarcadero.com/RADStudio/en/Deploying_dbExpress_Database_Applications se ofrece una tabla completa para cada RDBMS contemplado por dbExpress.

Conclusión

Como hemos podido comprobar a lo largo de este capítulo, dbExpress nos permite conectar nuestras aplicaciones a una base de datos con una gran sencillez, gracias tanto a los componentes ofrecidos, como TSimpleDataSet, como a las herramientas que integra el entorno de Delphi XE2 para facilitar su configuración.

dbExpress es el método preferente de acceso a datos en Delphi y prácticamente el único que avanza y seguirá evolucionando versión a versión del producto. En Delphi XE2 tenemos no solamente controladores para las últimas versiones de los RDBMS más usados, sino también controladores dbExpress para MacOS X y un nuevo controlador ODBC que nos abre las puertas a prácticamente cualquier depósito de información en la actualidad.

A continuación

Tras analizar y poner en práctica la estructura de una aplicación que almacena los datos localmente, en el capítulo anterior, y otra que se ajusta a la arquitectura cliente/servidor, en este capítulo, en el próximo conoceremos la tercera arquitectura posible: la de una aplicación distribuida.

En Delphi XE2 este tipo de aplicaciones se construyen con un conjunto de componentes conocidos genéricamente como DataSnap, en estrecha conexión con dbExpress, LiveBindings y el resto de lo aprendido hasta ahora.

Capítulo 7: DataSnap

Heredero de la tecnología MIDAS (*Multi-tier Distribution Application Services*) introducida en las primeras versiones de Delphi, DataSnap ha ido evolucionando a lo largo de los años hasta convertirse en una solución general para la construcción de aplicaciones distribuidas con Delphi. Esta arquitectura, evolución de la más clásica cliente/servidor, se compone habitualmente de tres pilares: un servidor de datos, un servidor de aplicaciones y una o más interfaces de usuario ligeras de acceso a los servicios.

En este capítulo conoceremos los elementos fundamentales de DataSnap, aprenderemos a conectar las distintas partes de que se compone un proyecto de este tipo y lo pondremos todo en práctica con una nueva versión de `MicroClassic`. En ella, de la interfaz de usuario desaparecerá prácticamente toda la lógica (el código) de gestión de negocio. Ésta pasará a implementarse en el servidor de aplicaciones, ejecutándose en una máquina independiente que actuará como intermediaria entre el servidor de datos y los ordenadores de cada uno de los clientes.

Componentes DataSnap

El elemento central de una aplicación distribuida desarrollada con Delphi es el servidor de aplicaciones o servidor DataSnap. Éste conectará opcionalmente[74] con un servidor de datos a través de dbExpress (u otro de los mecanismos de acceso a datos de Delphi) y responderá a las solicitudes efectuadas por los clientes. Estos conectarán con el servidor típicamente mediante TCP/IP o HTTP, si bien existen otras opciones heredadas de versiones previas como puede ser COM.

La conexión desde el servidor de aplicaciones hacia el de datos, asumiendo que esté basada en dbExpress que es el método preferente como hemos repetido en capítulos previos, se basa en exactamente los mismos componentes descritos en el capítulo anterior. Tendremos, por tanto, un `TSQLConnection` con los parámetros de la conexión y uno o más `TSQLDataSet` para acceder a las tablas, ejecutar consultas, procedimientos almacenados, etc. Cada uno de los conjuntos de datos tendrá asociado normalmente un `TDataSetProvider`.

> NOTA
>
> El servidor de aplicaciones raramente mantendrá un almacenamiento local de los datos, por lo que no se usará el componente `TClientDataSet` a menos que el propio servidor cuente con algún tipo de interfaz de administración o similar.

Al igual que un RDBMS operando en un servidor de datos ha de mantenerse constantemente a la escucha, normalmente en un puerto TCP/IP, para atender las solicitudes que reciba de sus clientes, el servidor de aplicaciones ha de hacer lo propio. Con este fin se usan los componentes de transporte DataSnap: `TDSTCPServerTransport` y `TDSHTTPService` (un derivado de `TDSHTTPServerTransport`). El primero facilitará las conexiones directas a través de TCP/IP, por ejemplo para clientes que son aplicaciones nativas Delphi ejecutándose sobre Windows o MacOS X, mientras que el segundo se

---

74 Es perfectamente posible tener un servidor de aplicaciones que no conecta con un RDBMS, ofreciendo otro tipo de servicios como pueden ser métodos que requieran una gran potencia de proceso disponible en el servidor pero no en los clientes. No obstante, la mayoría de los servidores de aplicaciones hacen uso de una base de datos.

ocupa de la conexiones HTTP/HTTPS habitualmente para los clientes basados en interfaces web y dispositivos móviles.

Los clientes conectarán con el servidor DataSnap, por los canales mencionados, para demandar algún servicio. Éste será implementado en el servidor como una clase cualquiera[75], cuyos métodos se expondrán hacia los clientes mediante un componente `TDSServerClass`. Un mismo servidor puede contar con varios componentes de este tipo, conectado cada uno de ellos a una clase que, por ejemplo, puede ofrecer servicios diferenciados del resto.

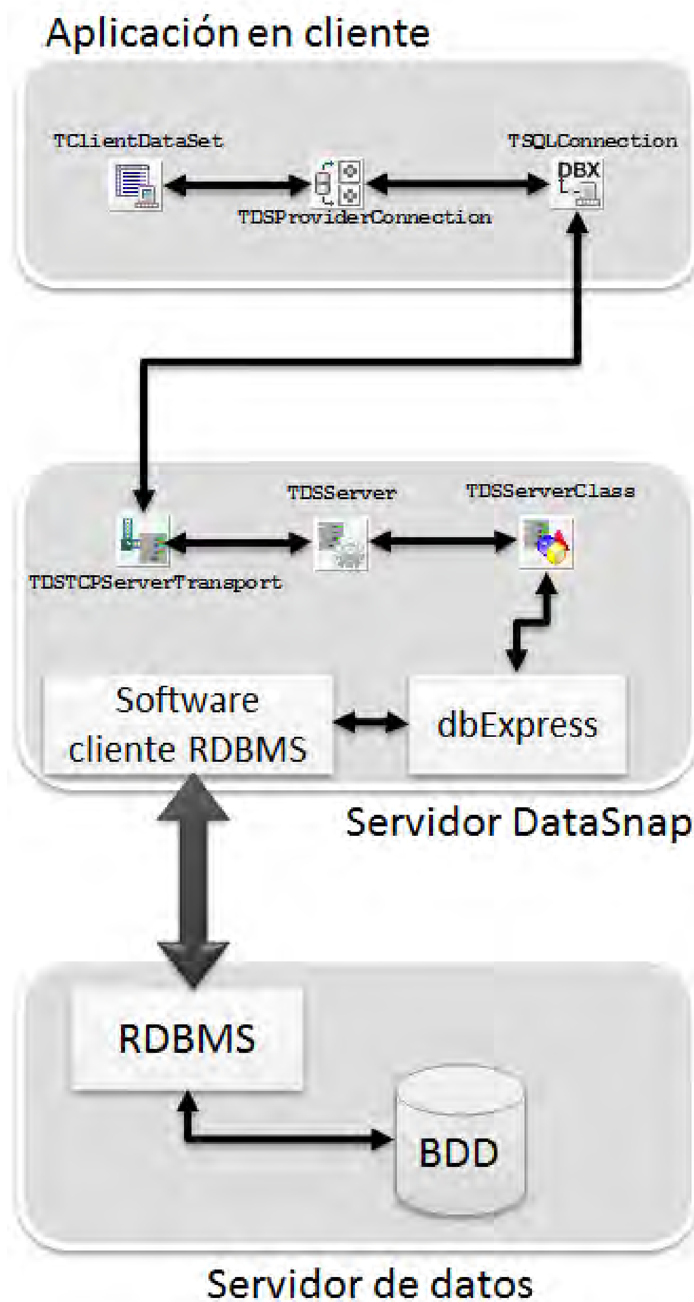
Todos los elementos del servidor: componentes que publican los servicios disponibles y aquellas que se encargan de facilitar la conexión y transporte de información en ambos sentidos, serán controlados por un componente `TDSServer`. Éste se encargará de crear a los anteriores en el momento adecuado, configurarlos y, cuando proceda, eliminarlos. Un servidor DataSnap contará con un único componente `TDSServer`.

En cuanto a las aplicaciones cliente que trabajan con el servidor DataSnap, conectarán y operarán con éste a través de dbExpress casi como si de un RDBMS se tratase: usando un componente `TSQLConnection` para configurar la conexión, un `TSQLDataSet` para ejecutar los servicios, etc. También existen algunos componentes específicos, como `TDSProviderConnection` que es una variante de `TDataSetProvider`, o `TSQLServerMethod` si simplemente se quiere ejecutar un método del servidor.

La figura de la página siguiente es una representación esquemática de los elementos citados: servidor de datos, servidor de aplicaciones y una aplicación cliente estándar (no de tipo web o móvil). En ésta el componente `TDSProviderConnection`, que permitiría trabajar con información de una base de datos a través del servidor DataSnap, podría ser sustituido por un `TSQLServerMethod` si únicamente se necesita ejecutar métodos ofrecidos por el servidor. La conexión al `TSQLConnection` se mantendría, al ser éste el componente encargado de comunicarse con el servidor DataSnap mediante TCP/IP.

---

75 Lo habitual es que se trate de una clase derivada de `TDSServerModule`, un tipo definido en `DSServer` y que facilita la exportación como servicios de los métodos de componentes alojados en un módulo de datos. Cuando no se precisa esta funcionalidad, sin embargo, es posible crear una clase a partir de `TComponent` e implementar manualmente los métodos que se deseen ofrecer como servicios.


Aplicación en cliente
TClientDataSet
TSQLConnection
DBX
TDSProviderConnection
TDSServer
TDSServerClass
TDSTCPServerTransport
Software cliente RDBMS
dbExpress
Servidor DataSnap
RDBMS
BDD
Servidor de datos

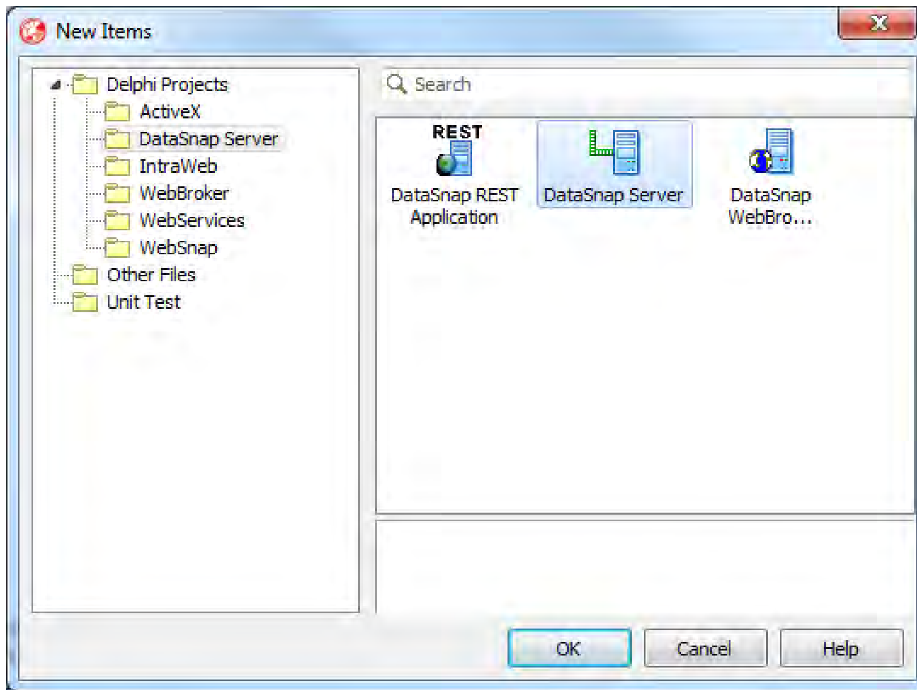
La comunicación entre servidor DataSnap y cliente, por ejemplo a través de TCP/IP, puede no ser directa en caso de que interese cifrar la información y/o comprimirla. Para ello se aplican filtros representados por objetos `TTransportFilter` y asociados al componente encargado de la comunicación.

Cada uno de los componentes citados ha de ser configurado adecuadamente asignando los valores pertinentes a sus propiedades: puerto de comunicación en que se ha de escuchar o por el que se intentará la comunicación, tiempo de vida de los objetos alojados en el servidor, etc. Esto nos resultará más cómodo en el cliente, como veremos después, una vez que tengamos el servidor DataSnap en funcionamiento, ya que éste aparecerá en el Explorador de datos como si de una conexión más se tratase. Para ayudarnos a configurar el servidor Delphi XE2 incluye múltiples asistentes.

> ADVERTENCIA
>
> Si bien el servidor de datos (RDBMS) puede ejecutarse en cualquier sistema operativo, al igual que las aplicaciones cliente que pueden ser nativas o de tipo web, el servidor de aplicaciones DataSnap únicamente puede ser ejecutado en Windows, ya sea en versión de 32 ó 64 bits. Es probable que en versiones posteriores de DataSnap se siga avanzando en el camino de la compatibilidad multiplataforma y podamos tener servidores de aplicaciones también en otros sistemas.

Un servidor DataSnap simple

Con el objetivo de familiarizarnos con los componentes que acaban de enumerarse vamos a construir un servidor DataSnap sencillo. Éste expondrá por una parte información procedente de la base de datos que creamos en el capítulo anterior, concretamente la única tabla con que cuenta dicha base de datos, y por otra ofrecerá algunos métodos que el cliente podrá ejecutar de manera remota.



Partimos creando un nuevo proyecto de tipo DataSnap Server. Lo encontraremos en la carpeta del mismo nombre del cuadro de diálogo New Items (véase imagen superior). Esto pondrá en marcha un asistente compuesto de múltiples pasos en los que va estableciéndose la configuración del servidor.

Modo de ejecución del servidor

El primer paso del mencionado asistente determina la forma en que se ejecutará la aplicación servidor, existiendo tres alternativas posibles que se corresponden con las opciones siguientes:

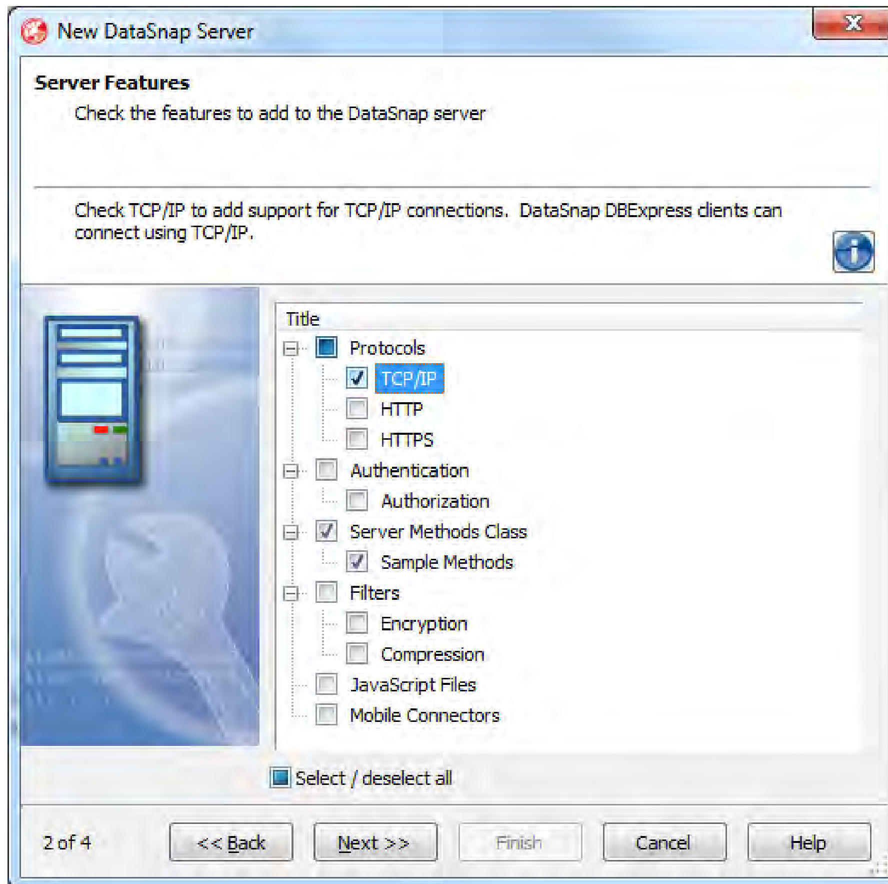
- VCL Forms Applications: El servidor será una aplicación que contará con interfaz gráfica, basada en la biblioteca VCL, lo cual facilitará la visualización de información de estado y acceso a opciones de configuración y administración en caso de existir. Es la opción seleccionada por defecto pero esto no implica que sea la más adecuada para un servidor de aplicaciones.

- Console Application: El servidor será una aplicación sin interfaz gráfica, siendo el único canal de comunicación con el usuario-administrador una consola de texto (la ventana Símbolo del sistema en Windows). Por regla general la consola será usada casi exclusivamente para que el servidor emita mensajes de estado, quedando la administración y configuración en manos de una herramienta externa o bien en la edición de un archivo de inicialización o similar.
- Service Application: Si se opta por una de las dos opciones previas el servidor, tenga o no interfaz de usuario, habrá de ser ejecutado a demanda como cualquier otro programa, tras haber iniciado sesión en el sistema. En general el software servidor, como pueden ser los RDBMS, servidores web, etc., se ejecutan como servicios. Esto permite que se pongan en marcha automáticamente (dependiendo de la configuración establecida) usando una cuenta determinada y sin necesidad de que se inicie sesión. De esta manera se garantiza el funcionamiento incluso en caso de que la máquina se reinicie y el administrador no tenga acceso inmediato a ella. Un servidor implementado como servicio no interactúa directamente con la pantalla y el teclado, por lo que su configuración y control precisan una herramienta adicional que puede ser específica, creada a medida para el servidor, o bien de tipo genérico como la consola Servicios de Windows.

En principio el desarrollo y prueba de un servidor de aplicaciones resultará más fácil si se dispone de una interfaz de usuario que, de forma interactiva, ofrezca información de estado y permita realizar cambios de configuración. Completado el proyecto, sin embargo, en la mayoría de los casos será preferible una implementación en forma de servicio, para lo cual bastaría con sustituir el formulario VCL por el módulo de configuración del servicio.

Características del servidor

Un clic en el botón Next del asistente nos llevará al paso siguiente, en el que se establecerán las características generales del servidor: protocolos de comunicación ante los que responderá, filtros a conectar para el cifrado y compresión de datos, mecanismo de autenticación de usuarios, etc. En la imagen de la página siguiente pueden verse las opciones existentes.



La primera rama de parámetros afecta a los protocolos de comunicación que el servidor contemplará, estando por defecto marcado TCP/IP[76] ya que es el usado con mayor frecuencia. Un cliente que se implemente como una aplicación Delphi nativa (no web) se comunicará de manera más eficiente a través de TCP/IP que usando HTTP o HTTPS. Estos dos protocolos, sin embargo, serán imprescindibles en caso de que el cliente esté basado en una interfaz web.

---

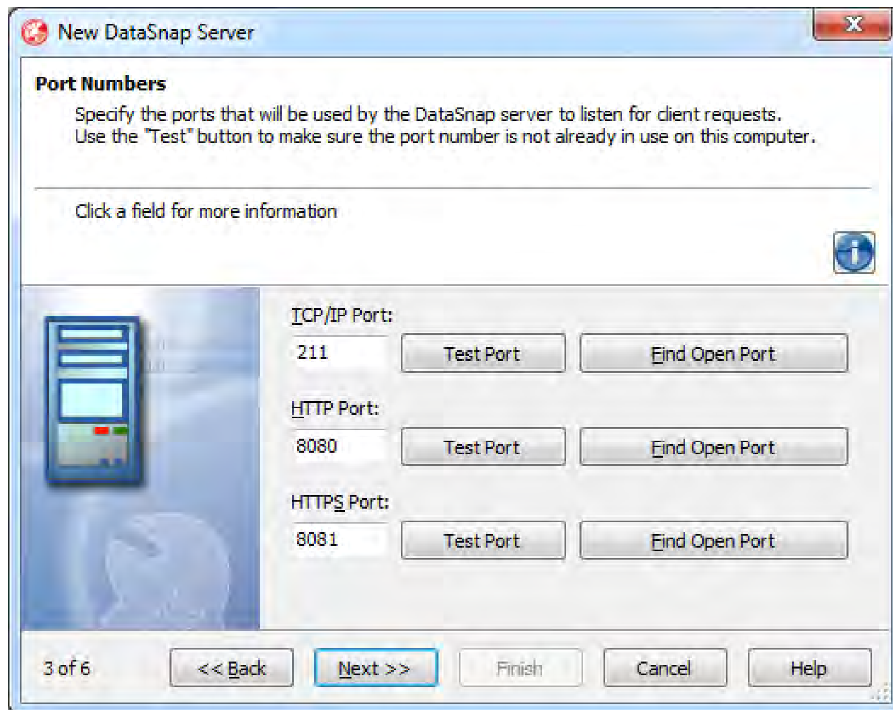
76 Aunque en la documentación de DataSnap se indica que debemos asegurarnos de tener siempre marcada la opción TCP/IP, lo cierto es que no hay ningún obstáculo para desarrollar un servidor que se comunique exclusivamente por HTTP/HTTPS y no de manera directa por TCP/IP.

Salvo que marquemos la opción Authorization el servidor no solicitará a los clientes credencial alguna para acceder a sus servicios, lo cual puede resultar apropiado si se opera en una red interna, no conectada a Internet y en consecuencia protegida contra accesos externos. Al marcar dicha opción se agregará al servidor un componente `TDSAuthenticationManager`, encargado de gestionar la identificación de los usuarios y sus perfiles, y cualquier intento de conexión al servidor requerirá una autenticación previa.

El resto de las opciones que aparecen en este paso del asistente agregan, en caso de marcarse, distintos módulos al proyecto: una clase con métodos que exponen servicios para los clientes, filtros asociados a los componentes de transporte y módulos en distintos lenguajes para facilitar el acceso al servidor desde aplicaciones web y aplicaciones para dispositivos móviles.

Configuración de puertos

El paso siguiente del asistente tendrá más o menos elementos dependiendo de los protocolos elegidos (véase la imagen inferior).



Por cada protocolo se propondrá un puerto, siempre el mismo para todos los proyectos, cuya disponibilidad puede comprobarse mediante el botón Test Port. El botón a la derecha del anterior permite buscar otro puerto abierto diferente.

En cualquier caso hemos de tener en cuenta que los puertos disponibles en la máquina de desarrollo no han de ser necesariamente los mismos que en la máquina que actuará como servidor de aplicaciones. Además hay que prever la posible necesidad de configurar software de cortafuegos, enrutadores y otros dispositivos de red, por lo que en la mayoría de los casos el puerto que ha de utilizarse surgirá por acuerdo entre el desarrollador y el administrador de red.

NOTA

Podemos utilizar los puertos por defecto durante el proceso de desarrollo y posteriormente, al instalar el servidor en la máquina definitiva, realizar los ajustes necesarios según las indicaciones del administrador de red.

Si entre los protocolos marcados estaba HTTPS, además de los puertos el asistente nos solicitará también los certificados digitales y claves asociadas para poder establecer una conexión segura.

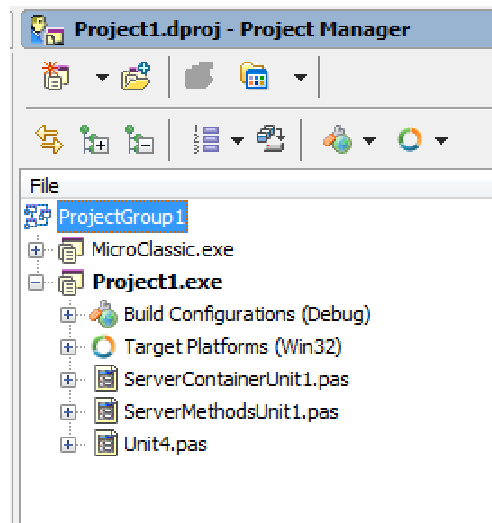
Otros parámetros

Si en el segundo paso del asistente se marcó la opción Server Methods Class el proyecto contará con un módulo en el que se definirá una clase con métodos accesibles para los clientes. Dicha clase puede estar derivada de `TComponent`, `TDataModule` o `TDSServerModule`, elección que tendremos que realizar en el siguiente paso del asistente. Un servidor que únicamente vaya a ofrecer a los clientes métodos a los que invocar puede derivar la clase directamente desde `TComponent`. Si se deriva de `TDataModule` además podrán utilizarse componentes no visuales desde esa clase. En la mayoría de los casos, no obstante, la clase base será `TDSServerModule` ya que permite introducir componentes dbExpress para ofrecer a los clientes acceso a datos obtenidos desde el RDBMS.

Finalmente, solo en caso de que haber marcado las dos últimas opciones, se solicitará el camino donde va a crearse el proyecto a fin de agregar al mismo los archivos con código JavaScript, hojas de estilo, etc.

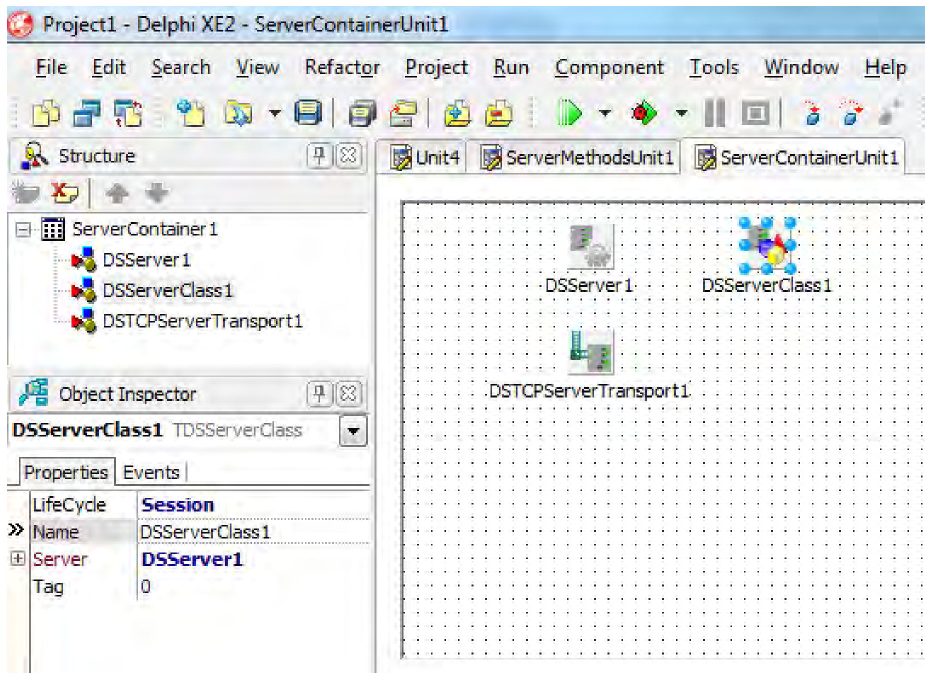
En nuestro caso crearemos un servidor con interfaz de usuario, aceptaremos en el segundo paso del asistente la configuración por defecto: se usará únicamente el protocolo TCP/IP y se agregará una clase con métodos de ejemplo. Ésta derivará de `TDSServerModule`.

Finalizado el asistente el proyecto debería contar con los elementos que pueden apreciarse en la imagen inferior: un formulario asociado a un módulo de código llamado `UnitN.pas`, un contenedor para los componentes del servidor llamado `ServerContainerUnit1.pas` y un módulo con la clase que expone los métodos accesibles para los clientes con el nombre `ServerMethodsUnit1.pas`.



En el diseñador aparecerá abierto el formulario que actuaría como interfaz de usuario del servidor y, aunque se trata de un formulario VCL y no FMX, los componentes disponibles y sus propiedades son similares, sino idénticas, a las que conocimos en capítulos previos.

Si abrimos el módulo `ServerContainerUnit1` veremos que es similar a un módulo de datos (véase la imagen de la página siguiente), un área cuadriculada que actúa como contenedor, y que tiene alojados tres componentes: un `TDSServer`, un `TSServerClass` y un `TDSTCPServerTransport`, estando los dos últimos conectados al primero mediante la propiedad `Server`. Lo primero que vamos a hacer es ocuparnos de las características y configuración de estos componentes.



Inicio del servidor y métodos administrativos

Comencemos con la configuración del componente `TDSServer` que, como se indicó anteriormente, es el encargado de controlar el funcionamiento del servidor y los demás componentes que forman parte de éste.

Debe distinguirse entre la ejecución del programa que actúa como servidor y la puesta en marcha de éste. Son acciones que pueden ir emparejadas, de hecho por defecto lo están ya que la propiedad `AutoStart` de `TDSServer` tiene inicialmente el valor `True`. Podemos, no obstante, desactivar esta característica y tomar el control del servidor mediante los métodos `Start` y `Stop`, lo cual nos permitiría ajustar su configuración antes de ponerlo en marcha. También se podría invocar a dichos métodos a demanda del administrador del servidor, agregando al formulario que actúa como interfaz de usuario un botón que iniciase/detuviese su funcionamiento.

Otra propiedad de interés en este componente es HideDSAdmin, de tipo Boolean como la anterior. El TDSServer usa internamente un objeto de la clase DSAdmin para ofrecer un conjunto de métodos con fines administrativos: obtener listas de clases y métodos disponibles en el servidor, obtener propiedades de conexión con bases de datos, etc. Todos esos métodos serán accesibles por parte de los clientes DataSnap a menos que asignemos el valor True a HideDSAdmin.

Para el desarrollo de nuestro servidor simple mantendremos la configuración por defecto, de manera que el servidor se pondrá en marcha automáticamente al iniciar la aplicación y podamos acceder a los métodos de administración.

Tiempo de vida de los objetos en el servidor

Un servidor DataSnap puede contar con una o más clases que expongan métodos accesibles para los clientes. Éstos los invocarán de forma remota, como veremos después, sin poder asumir ningún control sobre el proceso de creación y destrucción de objetos de dichas clases. Ésta es una responsabilidad que recae en el componente TDSServerClass.

Por cada clase con que vaya a contar el servidor cuyos métodos quieran exponerse en forma de servicios tendremos que agregar al módulo donde se encuentra el TDSServer un componente TDSServerClass, enlazando éste con el anterior mediante la propiedad Server. Además será necesario responder al evento OnGetClass del TDSServerClass, cuya finalidad es establecer de manera dinámica la clase que se desea exponer.

Si abrimos el código del módulo ServerContainerUnit1 que generó el asistente encontraremos el método siguiente:

```
procedure TServerContainer1.DSServerClass1GetClass(
  DSServerClass: TDSServerClass;
  var PersistentClass: TPersistentClass);
begin
  PersistentClass := ServerMethodsUnit1.TServerMethods1;
end;
```

El segundo parámetro se entrega por referencia no por valor, a fin de que pueda modificarse su contenido asignándole una referencia a la clase que nos interese, en este caso la alojada en el módulo ServerMethodsUnit1.

El componente TDSServerClass será el responsable de crear un objeto de esa clase cuando proceda, dependiendo del valor que se asigne a la propiedad LifeCycle que puede ser uno de estos tres:

- Server: En el servidor existirá un único objeto de la clase indicada, que se creará al iniciarse el servidor y no se destruirá hasta la detención de éste. Dicho objeto será, por tanto, compartido por todos los clientes que accedan a sus servicios, algo que hay que tener muy en cuenta en su diseño para evitar los problemas de reentrada[77].
- Session: Al llegar la primera solicitud por parte de un cliente se creará un nuevo objeto que se mantendrá vivo en el servidor hasta que la sesión concluya. Esto permite contar con un objeto individual por cliente pero con la capacidad de mantener información de estado entre las solicitudes, ya que el objeto no se destruye al final de la ejecución de cada método.
- Invocation: Con esta configuración se crearía un nuevo objeto para atender cada llamada a un método, por lo que no se mantendría información alguna entre llamadas. Es la forma en que opera el protocolo HTTP usado por los servidores web, si bien una aplicación web sí puede mantener información asociada a la sesión de un usuario a través de *cookies* y técnicas similares.

Por defecto el componente TDSServerClass se inicializa con el valor Session en la propiedad LifeCycle, al ser ésta la configuración más frecuente. Será la que mantengamos en nuestro primer servidor DataSnap.

NOTA

Al usar el ciclo de vida Invocation, especialmente si hay muchos clientes con gran demanda de servicios, la creación y destrucción continua de objetos puede convertirse en un problema para el rendimiento del servidor. Es posible, no obstante, usar los eventos OnCreateInstance y OnDestroyInstance de TDSServerClass para asumir esa tarea y, por ejemplo, recurrir a un *pool* de objetos con uso cíclico.

77 Este problema surge cuando estando en ejecución un método de la clase llega una solicitud de otro cliente que desea acceder al mismo método. Es la clase la responsable de que esto se realice de manera segura, enviando a cada cliente la respuesta que corresponda.

Configuración TCP/IP

El tercer componente que encontramos en el módulo ServerContainerUnit1 es de clase TDSTCPServerTransport. Su finalidad es gestionar la comunicación con los clientes, aceptando sus solicitudes y transfiriéndolas a la clase del servidor que deba atenderlas. Por cada conexión se creará un hilo independiente permitiendo así tratar con múltiples clientes de manera simultánea.

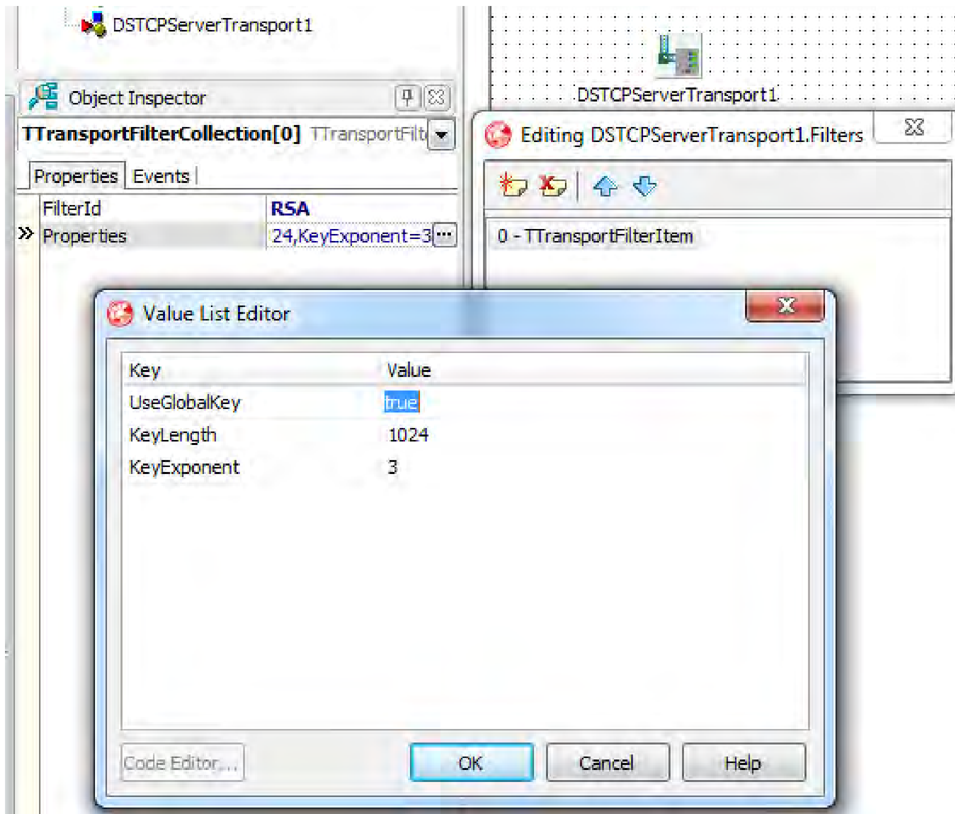
Cuando el asistente nos permitió seleccionar el puerto por el que el servidor quedaría a la escucha lo que hizo fue almacenar el número elegido en la propiedad Port del componente TDSTCPServerTransport. En caso de que necesitemos cambiar el puerto bastará con modificar el valor almacenado en dicha propiedad, sin más.

La propiedad Filters es la responsable de mantener los filtros asociados al canal de comunicación y sus respectivas configuraciones. El contenido inicial de esa propiedad dependerá de las opciones que hubiésemos marcado en el segundo paso del asistente. En cualquier caso siempre podemos personalizarlo mediante el editor asociado que, como es habitual, abriremos mediante el botón con tres puntos suspensivos dispuesto a la derecha del valor actual (en el Inspector de objetos).

El mencionado editor cuenta con una serie de botones que permiten agregar y eliminar filtros, así como establecer el orden en que han de aplicarse. Seleccionando un filtro cualquiera en el Inspector de objetos aparecerán las dos propiedades con que cuentan (véase la imagen de la página siguiente):

- FilterId: Determina el tipo del filtro. En la implementación actual contamos con dos filtros de cifrado: RSA y PC1, y un filtro de compresión llamado ZLibCompression.
- Properties: Almacena una colección de pares clave-valor, similar a la propiedad Params de un TSQLConnection, con las propiedades específicas del filtro indicado por FilterId. En la imagen de la página siguiente, por ejemplo, aparecen en primer plano las propiedades correspondientes al filtro RSA.

El resto de propiedades que afectarán a la comunicación entre el servidor y los clientes son BufferKBSize: fija el tamaño máximo de los paquetes TCP/IP y raramente será mayor que el valor por defecto de 32KB, y el terceto de propiedades KeepAliveXXX que mantienen los parámetros que afectan al tiempo de vida de la conexión.



En cuanto a la gestión de los hilos (*threads*) por parte de este componente, la configuración viene determinada por las propiedades `MaxThreads` y `PoolSize`. La primera indica el máximo número de hilos[78] que podrá usar el componente. El valor por defecto es `0` no imponiendo ningún límite en este sentido: se crearán tantos hilos como se necesiten para atender las solicitudes de los clientes.

Dado que la creación de nuevos hilos de ejecución tiene un cierto coste computacional, que podría afectar al rendimiento del servidor, es posible mantener un *pool* de hilos disponibles, preparados para ser lanzados en cualquier momento. El tamaño de ese paquete es lo que fija `PoolSize`.

---

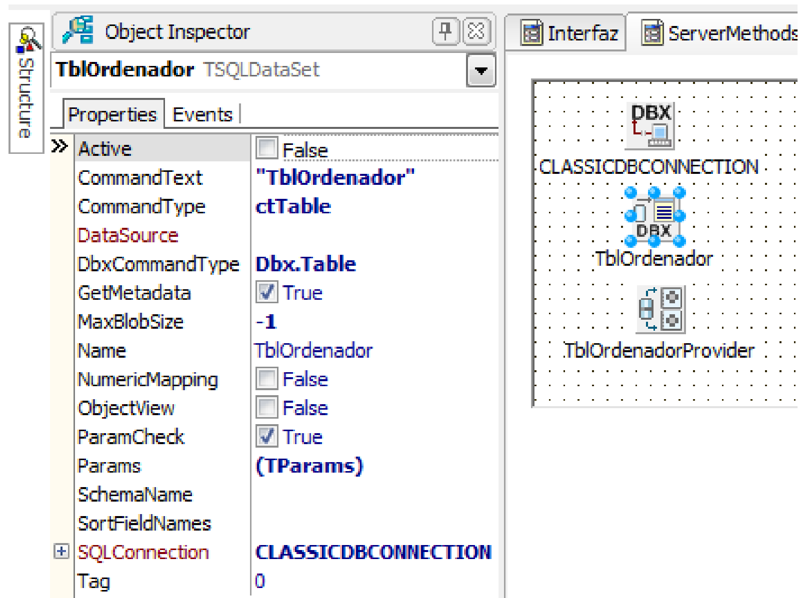
78 Si pretendemos ajustar este valor al inicializar el servidor DataSnap habremos de hacerlo antes de que se ponga en marcha, por lo que daremos el valor `False` a `AutoStart` y recurriremos al método `Start` una vez terminada la configuración.

Componentes de acceso a datos

El componente TDSServerClass apunta a TServerMethods1, definida en el módulo ServerMethodsUnit1, como clase que proveerá servicios a los clientes. Será en ella donde introduzcamos los componentes que permitirán a los usuarios acceder a la información sobre ordenadores existentes en nuestra base de datos.

¿Qué necesitamos para exponer el contenido de la tabla TblOrdenador y que el contenido de ésta pueda ser enviada a la aplicación cliente? Básicamente los mismos componentes dbExpress que usaríamos en una aplicación cliente/servidor:

- Un componente TSQLConnection configurado para acceder a la base de datos. Ésta puede residir en un servidor independiente o bien alojarse en la misma máquina que el servidor de aplicaciones.
- Un TSQLDataSet conectado al anterior y con la configuración que puede apreciarse en la imagen inferior para obtener la citada tabla.
- Un TDataSetProvider con la propiedad DataSet apuntando al TSQLDataSet y el resto de propiedades en configuración por defecto.



Podemos sencillamente tomar la tabla TblOrdenador desde el Explorador de datos, asumiendo que teníamos la conexión definida en capítulos previos, y arrastrarla hasta la superficie del diseñador para obtener el TSQLConnection y el TSQLDataSet completamente configurados, agregando después el TDataSetProvider y modificando su propiedad DataSet.

Métodos expuestos por el servidor

En su estado actual nuestro proyecto de servidor DataSnap expone como servicios para los clientes dos métodos implementados en la clase TServerMethods1: EchoString y ReverseString. Se trata de dos métodos de ejemplo agregados por el asistente y que se limitan a devolver como resultado la cadena que reciben como parámetro, ya sea sin cambiar o invirtiendo el orden los caracteres.

Si solamente necesitásemos exponer métodos simples, como los anteriores, la clase podría derivarse directamente de TComponent en lugar de tomar como base TDSServerModule que incorpora muchos más elementos, entre ellos la posibilidad de exponer componentes de acceso a datos.

Supongamos que queremos tener en el servidor algunos servicios básicos, no relacionados con el acceso a datos, como puede ser la obtención de la hora del servidor. Para implementar esta función basta con una clase derivada de TComponent que agregaríamos al proyecto en curso con los pasos indicados a continuación:

- Usando la opción Add>New Unit del menú contextual del proyecto añadimos un nuevo módulo de código. Éste se encontrará vacío en principio, sin más que la separación de las secciones de interfaz e implementación.
- Modificamos el código definiendo la clase TServidorSimple con un único método. La definición de la clase ha de estar delimitada por las directivas {$METHODINFO ON}[79] y {$METHODINFO OFF} como puede verse a continuación:

---

79 Esta directiva instruye al compilador de Delphi para que incluya la información necesaria que permita invocar posteriormente a los métodos de la clase. Esa información es una versión extendida de la RTTI (*Runtime Type Information*) habitual en Delphi.

```
unit ServidorSimple;
interface

uses System.SysUtils, System.Classes;

type
{$METHODINFO ON}
  TServidorSimple = class(TComponent)
  public
    function getHora: String;
  end;
{$METHODINFO OFF}

implementation

function TServidorSimple.getHora;
begin
  Result := DateTimeToStr(GetTime);
end;

end.
```

- Para que esta clase aparezca como un servicio accesible a los clientes tendremos que agregar al módulo `ServerContainerUnit1` un nuevo componente `TDSServerClass`. Modificaremos su propiedad `Server` para enlazarlo con el `TDSServer` y asociaremos a su evento `OnGetClass` un método con la siguiente sentencia:

```
  PersistentClass := ServidorSimple.TServidorSimple;
```

Con esta información se creará un objeto de la clase cada vez que se necesite, recuperando de él los datos sobre los métodos disponibles introducida por el compilador de Delphi. No necesitamos hacer nada más para que los clientes puedan invocar a este nuevo método.

> NOTA
>
> No es preciso que cada servicio ofrecido por el servidor DataSnap resida en una clase independiente. El método `getHora` podría haberse añadido perfectamente a la clase `TServerMethods1` que ya existía en el proyecto. La separación de los servicios en distintas clases se deberá principalmente a razones de diseño: mantener la cohesión de cada servicio y reducir el acoplamiento con otros, o bien por que se precisen tiempos de vida diferentes para los objetos.

Información sobre las conexiones activas

La interfaz con la que cuenta nuestro servidor DataSnap la usaremos únicamente para mostrar información sobre las conexiones efectuadas por los clientes. Sería muy sencillo, no obstante, agregar algunos elementos de administración, entre ellos el anteriormente citado botón que permitiese detener e iniciar el servidor a demanda del administrador.

Cada vez que un cliente conecta con el servidor el componente `TDSServer` genera un evento `OnConnect`. Análogamente, el cierre de esa conexión desencadena un evento `OnDisconnect`. Los métodos asociados a dichos eventos recibirán ambos un único parámetro de tipo `TDSConnectEventObject` con múltiples datos relativos a la conexión que acaba de abrirse o cerrarse.

Las conexiones se identifican de manera única asignándoles un código numérico que podemos encontrar en la propiedad `ChannelInfo.Id` del citado parámetro. La misma propiedad `ChannelInfo` cuenta con dos propiedades adicionales, llamadas `Info` y `ClientInfo`, con datos adicionales sobre el cliente como su dirección IP, puerto y aplicación que está usando.

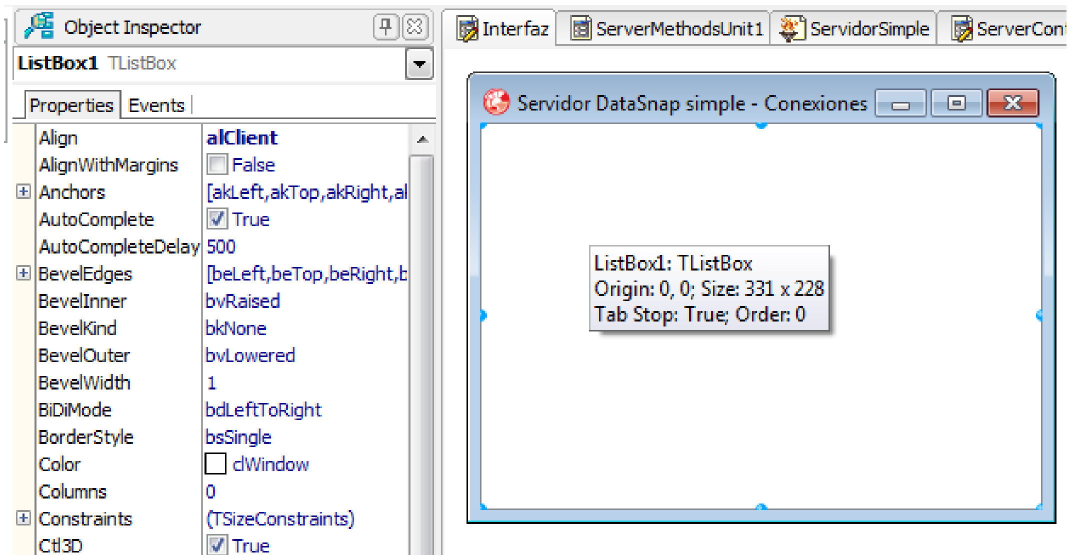
Otra de las propiedades de interés de `TDSConnectEventObject` es `ConnectProperties`. Ésta contiene una colección de pares clave-valor que corresponden a las propiedades de la conexión: protocolo de comunicación, dirección IP y puerto del servidor, nombre de la conexión y del controlador, filtros activos en el transporte, etc. Podemos recuperar la colección completa de propiedades mediante el método `GetLists` entregando como parámetros dos variables de tipo `TWideStringArray`.

Sirviéndonos de estas propiedades haremos que la interfaz de nuestro sencillo servidor muestre información sobre cada una de las conexiones realizadas. Comenzaremos por asociar al evento `OnConnect` del componente `TDSServer` el código necesario para obtener la lista de propiedades de conexión, con el método `ConnectProperties.GetLists`, así como el identificador único de la conexión. Estos datos serán entregados a un método, aun por implementar, del formulario que actúa como interfaz de usuario para que lo muestre al usuario.

Seleccionamos el componente `TDSServer`, elegimos el evento `OnConnect`, hacemos doble clic sobre su contenido para abrir el método asociado e introducimos en él las sentencias mostradas a continuación:

```
procedure TServerContainer1.DSServer1Connect(
  DSConnectEventObject: TDSConnectEventObject);
var
  props, valores: TWideStringArray;
begin
  DSConnectEventObject.ConnectProperties.GetLists(
                props,valores);
  VentanaADM.muestraConexion(
   DSConnectEventObject.ChannelInfo.Id, props,valores);
end;
```

`VentanaADM` será el nombre que demos al formulario que actúa como interfaz de usuario, formulario al que agregaremos, como puede comprobarse en la imagen inferior, un control `TListBox` que ocupará todo el espacio disponible.



Lo siguiente será agregar al formulario el método `muestraConexion` al que se llama desde el evento `OnConnect`. La implementación es la siguiente:

```
procedure TVentanaADM.muestraConexion(id: Integer;
    props: TWideStringArray; valores: TWideStringArray);
var
  I: integer;
begin
   ListBox1.Items.Add('Conexión con ID ' + IntToStr(id));
   for I := 0 to length(props)-1 do
     ListBox1.Items.Add(props[i] + ' = ' + valores[I]);
   ListBox1.Items.Add(StringOfChar('-', 100));
end;
```

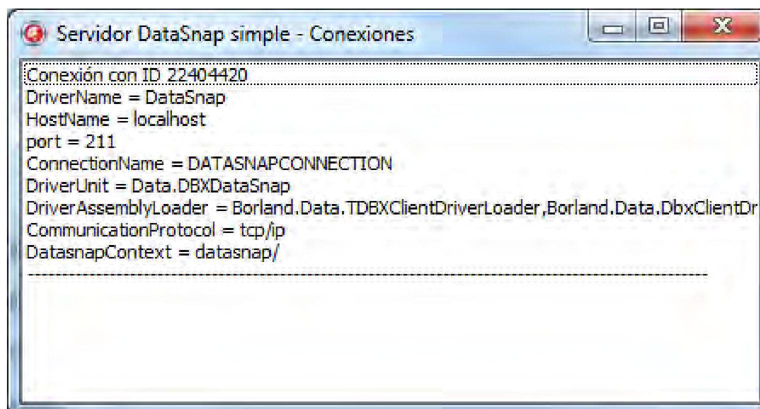
Añadimos a la lista un elemento informando sobre el identificador de la conexión, seguido por todos los elementos que vienen en las matrices de propiedades y valores. Finalmente se agrega un separador compuesto por una secuencia de cien guiones.

Con estos cambios en la interfaz completamos el desarrollo de nuestro servidor DataSnap que ya está preparado para aceptar conexiones de los clientes, cuyos datos mostrará en la lista, permitiéndoles invocar a varios métodos y acceder al contenido de la tabla de ordenadores.

Ejecución del servidor

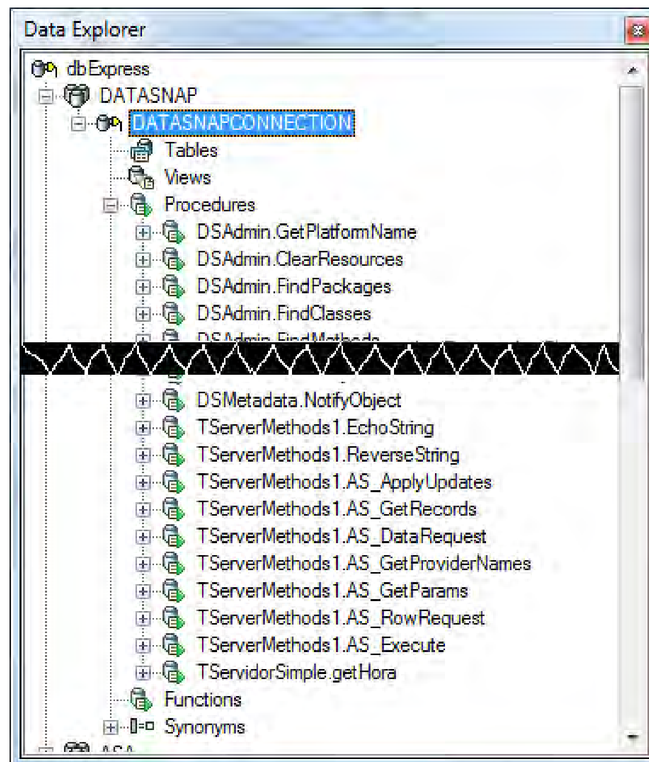
Incluso aunque todavía no contamos con un cliente preparado para conectar con este servidor DataSnap, sirviéndonos del Explorador de datos de Delphi podemos comprobar parcialmente su funcionamiento. Para ello lo primero que debemos hacer es ejecutar el servidor pero de manera que no active el modo de depuración en el entorno: abrimos el menú contextual del proyecto en el Gestor de proyectos y elegimos la opción Run Without Debugging.

Veremos aparecer la interfaz de la aplicación, con la lista inicialmente vacía. Dado que el componente `TDSServer` tenía la propiedad `AutoStart` a `True` automáticamente se habrá puesto en marcha el servidor y estará esperando solicitudes de los clientes. Podemos comprobarlo abriendo el Explorador de datos, desplegando la rama DATASNAP, a continuación la conexión DATASNAPCONNECTION y finalmente la rama Procedures. En la ventana de nuestro servidor aparecerá la lista de propiedades (véase la imagen inferior) de la conexión establecida por el entorno de Delphi.



> ADVERTENCIA
>
> La conexión DataSnap predefinida en el Explorador de datos, llamada `DATASNAPCONNECTION`, nos servirá siempre que nuestros servidor contemple la comunicación por TCP/IP a través del puerto 211 que es la configuración por defecto. Si hemos cambiado algún parámetro, usando un protocolo diferente u otro puerto, habremos de modificar las propiedades de la conexión para poder acceder al servidor.

En el Explorador de datos aparecerá una extensa lista de procedimientos expuestos por el servidor, la mayor parte de ellos con el prefijo `DSAdmin` que identifica a la clase en la que se implementan los métodos de administración del servidor. A continuación de éstos encontraremos los de nuestras clase `TServerMethods1` y `TServidorSimple` como puede apreciarse en la imagen inferior.

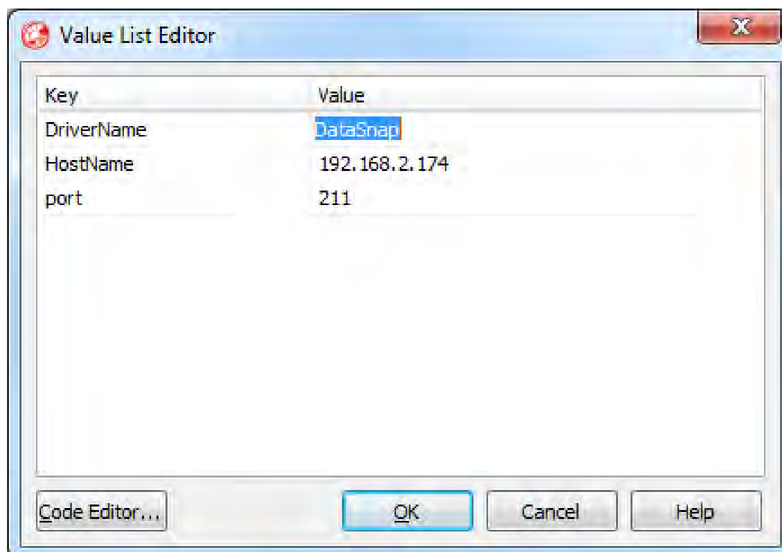


Abriendo el nodo correspondiente a cualquiera de los métodos obtendremos información sobre los parámetros que precisan y el valor que devuelven. Es toda la información que se necesita para desarrollar un cliente que acceda a los servicios expuestos por el servidor DataSnap.

Desarrollo de un cliente básico

Nuestro cliente será una aplicación FMX con un formulario desde el que podrá accederse a los servicios del servidor, entre ellos la información de la tabla de ordenadores.

Lo primero que necesitamos es un componente que permita conectar con el servidor y ese componente es un `TSQLConnection`. Tras agregarlo al formulario desplegamos la lista asociada a la propiedad `ConnectionName` y seleccionamos el elemento `DATASNAPCONNECTION`. De esta forma se leerá toda la configuración de dicha conexión. También podemos, como método alternativo, elegir `DataSnap` para la propiedad `DriverName`, abrir el editor de la propiedad `Params` e introducir la dirección IP o nombre de la máquina donde está el servidor y el puerto, tal y como se ha hecho en la imagen inferior.



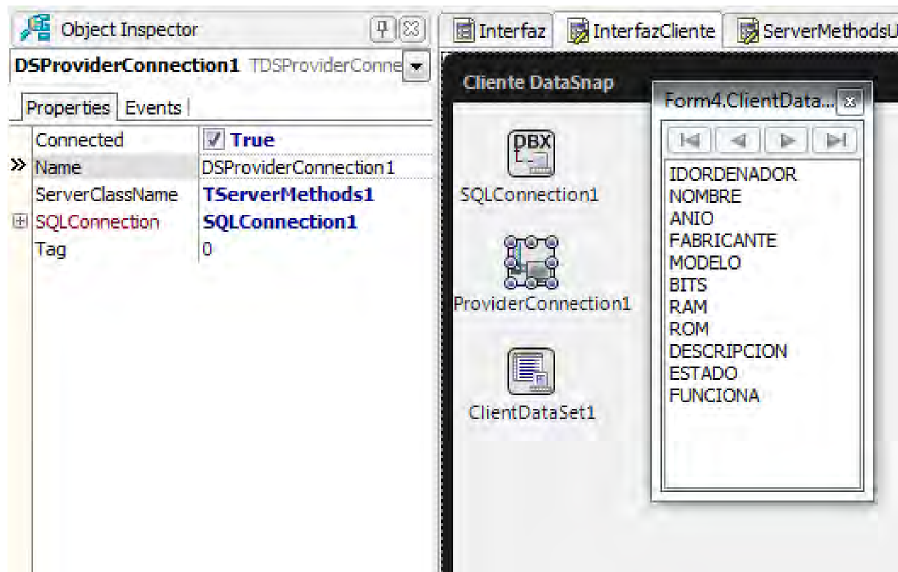
Pondremos a `False` la propiedad `LoginPrompt` ya que, por el momento, el servidor no requiere una identificación para el acceso a sus servicios.

Asumiendo que el servidor sigue en ejecución, bastará con dar el valor `True` a la propiedad `Connected` del `TSQLConnection` para conectar con él. Veremos aparecer en la interfaz del servidor un nuevo paquete de información. Si el servidor no estuviese ejecutándose la conexión fallaría y `Connected` no llegaría a tomar el valor `True`.

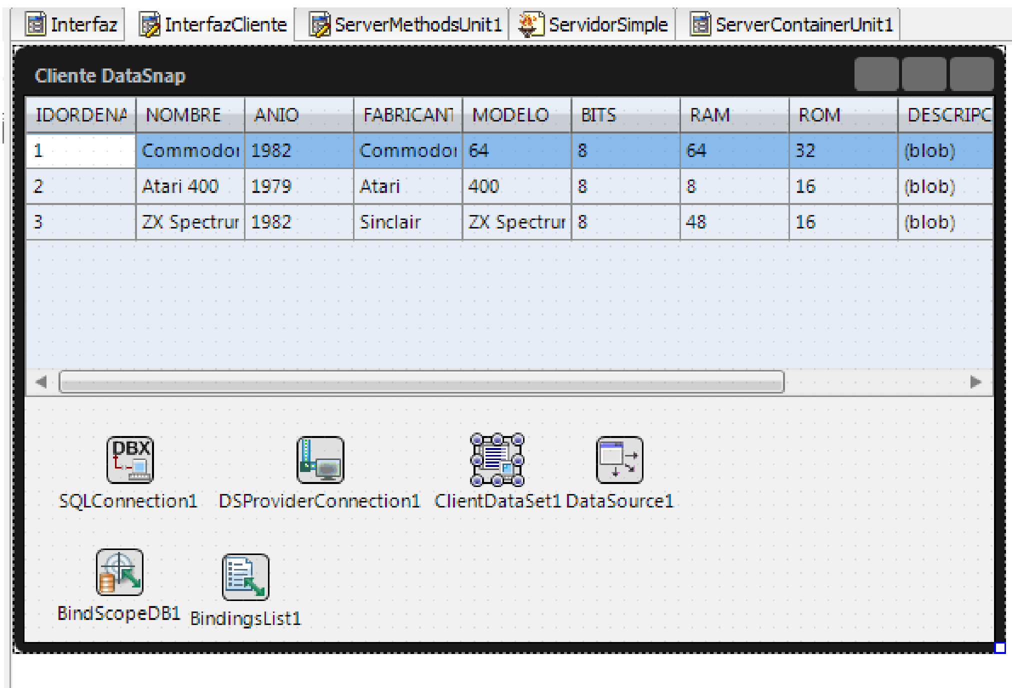
Para que el cliente pueda acceder al proveedor remoto, a fin de obtener la información de la tabla de ordenadores, agregaremos al formulario un componente `TDSProviderConnection`. Las propiedades clave de éste son `SQLConnection`, que le enlazarán con el componente de conexión anterior, y `ServerClassName` a la que debemos asignar el nombre de la clase del servidor que ofrece los datos. En este caso, asumiendo que no lo hemos cambiado, el nombre es `TServerMethods1`.

El siguiente elemento a introducir será un `TClientDataSet` del que también modificaremos dos propiedades: `RemoteServer` le enlazará con el `TDSProviderConnection` y `ProviderName` apuntará al `TDataSetProvider` del servidor (basta con abrir la lista asociada a estas propiedades y elegir el único valor disponible).

Podemos comprobar si todo va bien abriendo el editor de columnas del `TClientDataSet` y usando la opción Add All Fields. Debería aparecer la lista completa de columnas, como en la imagen inferior, obtenida desde el servidor.



Completaremos el acceso a la tabla de ordenadores agregando al formulario un `TDataSource`, que conectaremos con el `TClientDataSet`, y un `TStringGrid`. Usando la opción Link to DB DataSource crearemos un LiveBinding para vincular la cuadrícula con los datos. En cuanto demos el valor `True` a la propiedad `Connected` del `TClientDataSet` veremos aparecer en la cuadrícula (imagen inferior) los datos de la tabla. Éstos son obtenidos desde el RDBMS mediante una conexión dbExpress en el servidor DataSnap, no directamente por el cliente que puede estar ejecutándose en otra máquina y no precisa controladores ni software cliente de Firebird.



Para que los cambios que pudiesen efectuarse sobre los datos, editando directamente en el `TStringGrid`, sean transferidos al servidor DataSnap y desde éste al RDBMS llamaremos al método `ApplyUpdates` del `TClientDataSet` cuando se cierre el formulario (evento `OnClose`).

> NOTA
>
> Opcionalmente podríamos añadir un `TBindNavigator` para permitir otras operaciones, como el borrado e inserción de filas.

Además de para acceder a los datos sobre ordenadores también queremos usar la aplicación cliente para ejecutar algunos de los métodos expuestos por el servidor. Concretamente queremos obtener la hora del servidor, por ejemplo cada vez que se haga clic sobre un botón, y también invertir una cadena de caracteres cualquiera.

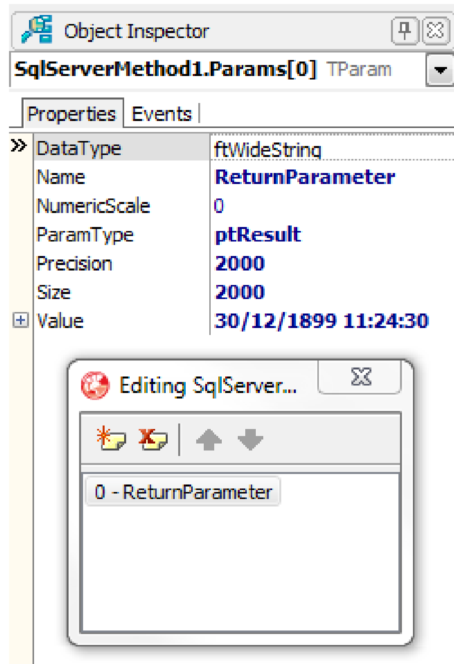
Para invocar a un método del servidor podemos usar el componente `TSqlServerMethod`. Éste es similar a los derivados de `TDataSet` pero no almacena un conjunto de filas y columnas, sino que recurre a la propiedad `Params` para almacenar los parámetros de entrada como de salida del método a ejecutar. El nombre de éste habrá que indicarlo en la propiedad `ServerMethodName` una vez se haya conectado este componente con el `TSQLConnection`.

En nuestro caso, para ejecutar el método que devuelve la hora del servidor, realizamos sobre el cliente los cambios indicados a continuación:

- Introducimos en el formulario un `TSqlServerMethod` y lo enlazamos con el `TSQLConnection` que ya había.
- Desplegamos la lista asociada a la propiedad `ServerMethodName` y elegimos `TServidorSimple.getHora` que es el método que queremos ejecutar.
- Hacemos clic en el botón asociado a la propiedad `Params` del `TSqlServerMethod` para abrir el editor de parámetros. Veremos que hay uno llamado `ReturnParameter`. Al seleccionarlos podremos ver en el Inspector de objetos, como se aprecia en la imagen de la página siguiente, el tipo del parámetro y su valor actual entre otros datos.
- Agregamos al formulario un `TButton`, hacemos doble clic sobre él para abrir el método asociado al evento `OnClick` e introducimos el código siguiente:

```
procedure TForm4.Button1Click(Sender: TObject);
begin
  SqlServerMethod1.ExecuteMethod;
  Button1.Text := SqlServerMethod1.Params[0].Value;
end;
```

- Finalmente generamos el método correspondiente al evento `FormCreate` del formulario y desde él invocamos al método anterior para que se muestre la hora del servidor en cuanto se abra la ventana.



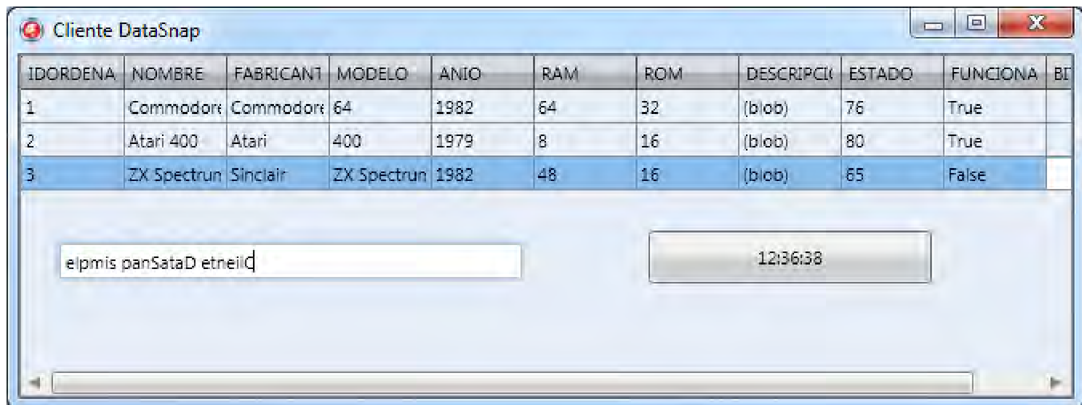
El procedimiento a seguir para invocar al método del servidor que invierte la cadena de caracteres recibida como parámetro será similar, pero teniendo en cuenta que en este caso existe un parámetro de entrada y otro de salida (el resultado). Los pasos a seguir:

- Añadimos un nuevo `TSqlServerMethod` y lo conectamos con el `TSQLConnection`.
- Abrimos la lista de la propiedad `ServerMethodName` y elegimos el elemento `TServerMethods1.ReverseString`.
- Si consultamos la propiedad `Params` comprobaremos que el primer parámetro (índice 0) es el de entrada y el segundo el de salida.
- Colocamos en el formulario un componente `TEdit` para permitir la introducción de una cadena de caracteres cualquiera.
- Localizamos en la página de eventos del Inspector de objetos el evento `OnKeyUp` del `TEdit`, hacemos doble clic sobre él e introducimos el código mostrado a continuación.

```
procedure TForm4.Edit1KeyUp(Sender: TObject;
   var Key: Word; var KeyChar: Char; Shift: TShiftState);
begin
  if Key = 13 then
   with SqlServerMethod2 do begin
      Params[0].Value := Edit1.Text;
      ExecuteMethod;
      Edit1.Text := Params[1].Value;
  end;
end;
```

El parámetro `Key` trae el código de la tecla que acaba de soltarse. En caso de que ese código sea `13` indicará que ha sido la tecla Intro, señal que usamos para tomar el texto del `TEdit` y enviarlo como parámetro al método del servidor, tomando el resultado devuelto por éste para volver a mostrarlo en el mismo `TEdit`.

Ya podemos ejecutar nuestro cliente que, siempre que pueda conectar con el servidor, mostrará una apariencia similar a la de la figura inferior. La cuadrícula no solamente expone el contenido actual de la tabla, sino que permite modificar los datos existentes. El botón refleja la hora en el servidor DataSnap, que puede actualizarse con un simple clic, y el recuadro de texto contiene una cadena de texto invertida.

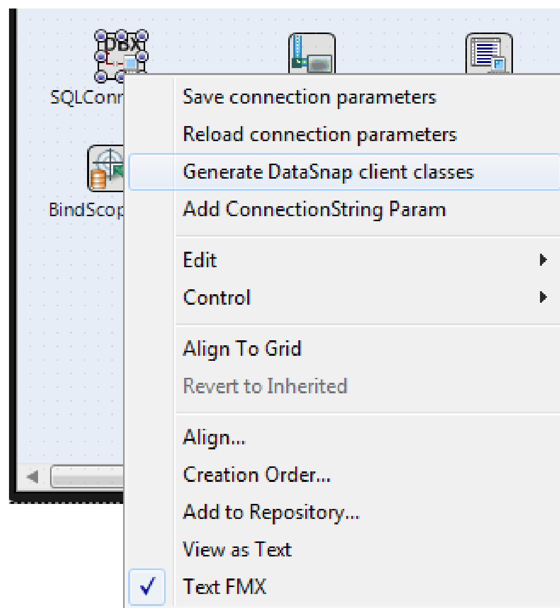


> NOTA
> En la aplicación cliente se deberían controlar las posibles excepciones, especialmente las relacionadas con la imposibilidad de conectar con el servidor.

Proxys de acceso a los servicios

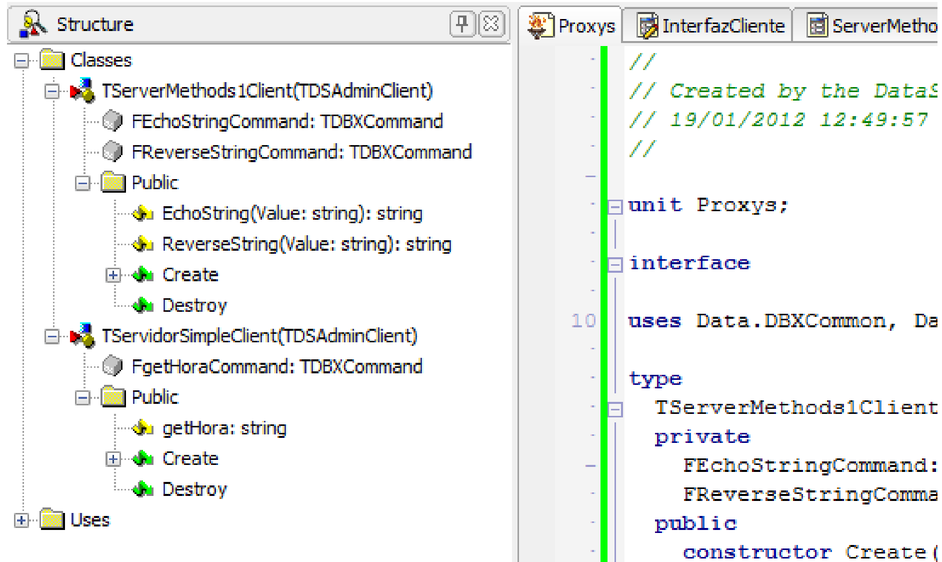
En el cliente que hemos desarrollado hemos tenido que introducir dos componentes `TSqlServerMethod`, asignando a los elementos de su propiedad `Params` los parámetros de entrada y recogiendo de ellos los resultados. Si en lugar de usar dos métodos del servidor fuesen decenas, no cabe duda de que este procedimiento requeriría un trabajo bastante tedioso por nuestra parte.

Una alternativa a la invocación mediante componentes `TSqlServerMethod` consiste en generar por cada clase con métodos expuestos por el servidor una clase en el cliente que actúe como *proxy*[80], facilitando el acceso a todos los métodos existentes que aparecerían como si fuesen locales. Podemos generar automáticamente esos *proxys* mediante la opción Generate DataSnap client classes del componente `TSQLConnection` (véase la imagen inferior).



80 En el contexto en que nos encontramos, tratando sobre aplicaciones distribuidas, un *proxy* es un objeto que actúa en representación de otro que opera de manera remota y que mantiene su misma *apariencia:* conjunto de miembros, de forma que es posible usarlo sin conocer más detalles sobre su funcionamiento.

Al ejecutar esa opción comprobaremos cómo se agrega al proyecto un nuevo módulo de código, en el que se define por cada clase del servidor que expone servicios una clase derivada de TDSAdminClient. En nuestro caso, como puede verse en la ventana Structure (imagen inferior), son dos clases: TServerMethods1Client y TServidorSimpleClient. La primera tiene los métodos de ejemplo EchoString y ReverseString, mientras que la segunda expone el método getHora que devolvía la hora del servidor.



En cada clase existirá un atributo de tipo TDBXCommand por cada método, dos constructores públicos, un destructor y todos los métodos de la clase original con el mismo nombre, parámetros de entrada y de salida. La versión *proxy* de nuestra clase TServidorSimple es la siguiente:

```
TServidorSimpleClient = class(TDSAdminClient)
 private
   FgetHoraCommand: TDBXCommand;
 public
   constructor Create(ADBXConnection: TDBXConnection);
overload;
   constructor Create(ADBXConnection: TDBXConnection;
       AInstanceOwner: Boolean); overload;
   destructor Destroy; override;
   function getHora: string;
end;
```

Para usar el método getHora implementado en el servidor DataSnap, por tanto, no tendríamos más que crear un objeto de la clase TServidorSimpleClient e invocar al método del mismo nombre. Es en la implementación de éste donde se esconde la *magia*:

```
function TServidorSimpleClient.getHora: string;
begin
  if FgetHoraCommand = nil then
  begin
    FgetHoraCommand := FDBXConnection.CreateCommand;
    FgetHoraCommand.CommandType :=
           TDBXCommandTypes.DSServerMethod;
    FgetHoraCommand.Text := 'TServidorSimple.getHora';
    FgetHoraCommand.Prepare;
  end;
  FgetHoraCommand.ExecuteUpdate;
  Result :=
     FgetHoraCommand.Parameters[0].Value.GetWideString;
end;
```

Este método crea un objeto TDBXCommand, funcionalmente equivalente al componente TSqlServerMethod pero sin la sobrecarga de éste[81], y lo configura para acceder al método del servidor. La conexión con éste se habrá facilitado con anterioridad al constructor de la clase TServidorSimpleClient. El objeto se crea una sola vez y a partir de ese momento se usa tantas veces sea necesario para llamar al método remoto, recuperando el resultado obtenido y devolviéndolo como valor de retorno de la función.

Una vez que hayamos añadido a la interfaz de nuestro cliente una referencia al módulo que contiene los *proxys*, con la opción File>Use Unit o agregando dicho módulo a la cláusula uses del primero, podremos cambiar el código asociado al evento OnClick del botón que ahora quedaría así:

```
procedure TForm4.Button1Click(Sender: TObject);
begin
  with TServidorSimpleClient.Create(Conn1.DBXConnection)
do
    Button1.Text := getHora;
end;
```

La forma de acceder al método es más simple y resulta más natural. Por supuesto podemos eliminar el componente TSqlServerMethod que usábamos anteriormente para realizar esa misma llamada.

---

81 Al no ser un componente derivado de TComponent no puede ser usado en un diseñador, pero a cambio requiere menos recursos para funcionar.

Ciclo de vida de objetos en el servidor

Uno de los aspectos más importantes durante el diseño de un servidor de aplicaciones es la gestión del tiempo de vida de los objetos alojados en él. Usando DataSnap el servidor puede contar con múltiples clases, asociada cada una de ellas con un componente `TDSServerClass` y con una configuración individual de ciclo de vida regida por la propiedad `LifeCycle` mencionada anteriormente.

Podemos experimentar las diferentes configuraciones posibles con nuestro sencillo servidor, introduciendo unos pequeños cambios en la implementación de la clase `TServidorSimple`. El código de ésta quedará como puede verse a continuación:

```
type
{$METHODINFO ON}
  TServidorSimple = class(TComponent)
  private
    Fhora: String;
  public
    constructor Create(AOwner: TComponent); override;
    function getHora: String;
  end;
{$METHODINFO OFF}

implementation

constructor TServidorSimple.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  Fhora := TimeToStr(GetTime);
end;

function TServidorSimple.getHora;
begin
  Result := Fhora;
end;
```

La clase cuenta ahora con un constructor que redefine al heredado de la clase base que es `TComponent`. Esto nos permite, tras delegar en el constructor heredado el tratamiento del parámetro recibido, obtener la hora actual del servidor y dejarla almacenada en una variable interna. El método `getHora` se ha modificado para que devuelva dicha variable.

Configuración por defecto

El valor que toma la propiedad `LifeCycle` por defecto es `Session` y, en consecuencia, se creará un objeto de la clase asociado a cada sesión activa en el servidor. Habitualmente una sesión corresponde a la conexión por parte de un cliente, por lo que podríamos decir que se crean objetos individuales para cada uno de ellos. Esos objetos se mantienen en memoria en el servidor hasta que la sesión finaliza.

Tras haber efectuado los cambios indicados antes, podemos ejecutar el servidor y, a continuación, iniciar varias instancias del cliente[82]. Comprobaremos que para cada uno de ellos la hora es diferente, lo que prueba que hay un objeto `TServidorSimple` independiente atendiendo a cada cliente.

El hecho de que la hora se obtenga en el constructor de ese objeto queda patente al hacer clic sobre el botón. La hora no se actualiza, el método `getHora` está devolviendo el contenido de un atributo y éste toma su contenido cuando se crea el objeto: al establecerse la conexión con el servidor. El constructor de la clase, por lo tanto, sería el lugar adecuado para introducir cualquier tarea que haya que efectuar una sola vez por cliente conectado al servidor, especialmente si se trata de una tarea con ciertos requerimientos de procesamiento.

> NOTA
>
> Dado que el cliente que hemos desarrollado mantiene abierta la conexión con el servidor durante toda su ejecución, a fin de mostrar los datos en la cuadrícula y facilitar su edición, el objeto `TServidorSimple` no se destruirá hasta cerrar la ventana. Sería diferente si el cliente abriese la conexión cada vez que necesitase hacer uso de un servicio y la cerrase a continuación.

---

82 Lo más fácil es usar varias veces la opción Run Without Debugging del menú contextual asociado al proyecto, teniendo varias copias del cliente en ejecución.

Creación de un objeto por llamada

Podemos recuperar el funcionamiento original del botón en el cliente introduciendo un cambio en el servidor: asignar el valor `Invocation` a la propiedad `LifeCycle` del segundo `TDSServerClass`. Con esto conseguiremos que el objeto de clase `TServidorSimple` se cree con cada llamada al método `getHora`, destruyéndose de manera inmediata una vez que se ha devuelto el resultado.

Este ciclo de vida tiene sentido únicamente para objetos cuyo estado no se necesita preservar entre llamadas sucesivas de una misma sesión de trabajo con un cliente, como es el caso de `TServidorSimple`. No es recomendable utilizarlo si la creación del objeto es costosa en tiempo y/o recursos, por ejemplo si ha de establecer una conexión con una base de datos, cargar en memoria una cantidad importante de información o realizar un procesamiento de preparación lento.

Haz el cambio indicado y a continuación ejecuta el servidor y una o varias instancias del cliente, comprobando que, en efecto, la hora se actualiza con cada clic en el botón.

Creación de un único objeto global

El tercer valor que puede tomar la propiedad `LifeCycle` es `Server` y, como se indicó antes, conlleva la creación de un único objeto de la clase afectada en el momento en que se pone en marcha el servidor, objeto que es compartido por todas las conexiones. Es equivalente, por tanto, a la implementación del patrón *singleton* en dicha clase.

Usar un único objeto en el servidor para todos los clientes tiene sentido en casos muy concretos, cuando se necesita que compartan un mismo estado o información que no reside necesariamente en una base de datos. Hay que poner especial cuidado en el diseño de estos servicios compartidos, de manera que el acceso simultáneo por parte de varios clientes no genere problemas.

En el caso de nuestro servidor, al dar el valor `Server` a `LifeCycle`, ejecutar el servidor y a continuación varias instancias del cliente comprobaremos que todos ellos muestran la misma hora, no actualizándose en ningún caso ya que solamente hay un objeto `TServidorSimple` creado desde un principio.

MicroClassic distribuido

Apoyándonos en lo que hemos aprendido sobre DataSnap a lo largo de este capítulo nos proponemos ahora crear una versión distribuida del proyecto `MicroClassic`. La aplicación constará de tres capas: un servidor de datos que es Firebird funcionando en una máquina, un servidor de aplicaciones DataSnap operando en una segunda máquina y una interfaz de usuario que se ejecutaría en los ordenadores de cada uno de los usuarios de la aplicación. Todos ellos compartirían no ya una misma base de datos, sino también un mismo proveedor de servicios que centraliza el acceso a la información.

El proyecto original tendrá por tanto que ser dividido en dos proyectos: el servidor de aplicaciones y la interfaz cliente. En el primero se alojarán los componentes de acceso a la base de datos y la lógica asociada con ésta, por ejemplo el método que obtiene el siguiente identificador de la secuencia generada en el RDBMS. El segundo será una versión más simple del proyecto original al prescindir de la conexión con la base de datos. Ésta será sustituida por la conexión con el servidor.

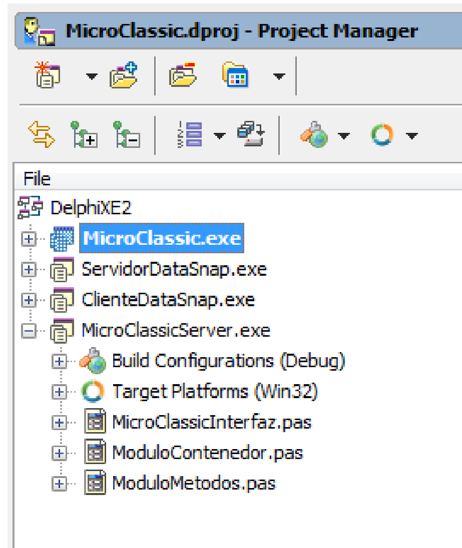
MicroClassicServer

El servidor se iniciará como un proyecto DataSnap aceptando los valores por defecto propuestos por el asistente con una excepción: desactivaremos la generación de los métodos de ejemplo ya que no nos interesa tener las funciones de eco e inversión de una cadena.

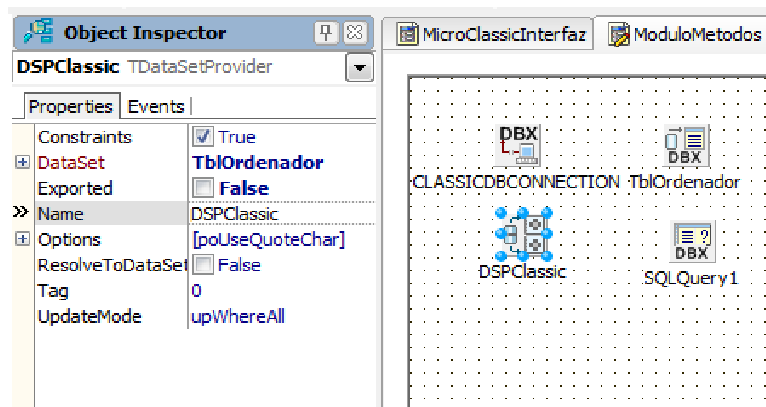
Los módulos del proyecto serán los que pueden verse en la imagen de la página siguiente, correspondiente al Gestor de proyectos, con el siguiente contenido:

- `MicroClassicInterfaz`: Es el módulo asociado al formulario VCL que actuará como interfaz de administración del servidor. Insertaremos en dicho formulario un componente `TListBox` que servirá solamente para mostrar información.
- `ModuloContenedor`: En él se encuentran los componentes `TDSServer`, `TDSServerClass` y `TDSTCPServerTransport` con su configuración por defecto.

- `ModuloMetodos`: Se trata del módulo que aloja la clase derivada de `TDSServerModule`, a la que llamaremos `TMCServidor` y que ofrecerá a los clientes tanto el acceso a los datos como los servicios adicionales.



En este último módulo introduciremos los elementos que pueden verse en la imagen inferior: un `TSQLConnection`, `TSQLDataSet`, un `TDataSetProvider` y un `TSQLQuery`. Este último tendrá la misma consulta que usábamos en el capítulo previo para obtener el siguiente valor a usar como identificador de ordenador.



Usaremos el evento BeforeConnect del TSQLConnection para agregar a la lista que actúa como interfaz de administración un mensaje y, opcionalmente, establecer las credenciales de acceso a la base de datos en caso de que no se hubiesen introducido ya en la propiedad Params:

```
procedure TMCServidor.CLASSICDBCONNECTIONBeforeConnect(
         Sender: TObject);
begin
  with CLASSICDBCONNECTION do begin
    Params.Values['User_Name'] := 'sysdba';
    Params.Values['Password'] := 'masterkey';
  end;
   MicroClassicSvr.LBMensajes.Items.Add(
      'Establecida conexión con la base de datos');
end;
```

Aquí MicroClassicSvr es el nombre que hemos dado al formulario VCL que actúa como interfaz y LBMensajes el nombre del control TListBox.

Además agregaremos un método para que los clientes puedan obtener el siguiente valor del generador del RDBMS y usarlo como identificador de un nuevo ordenador. El código de implementación sería el siguiente:

```
function TMCServidor.getNextID;
begin
   with SQLQuery1 do begin
    Open;
    Result := Fields[0].AsInteger;
    Close;
   end;
   MicroClassicSvr.LBMensajes.Items.Add(
        'Enviado nuevo ID de ordenador: ' +
         IntToStr(Result));
end;
```

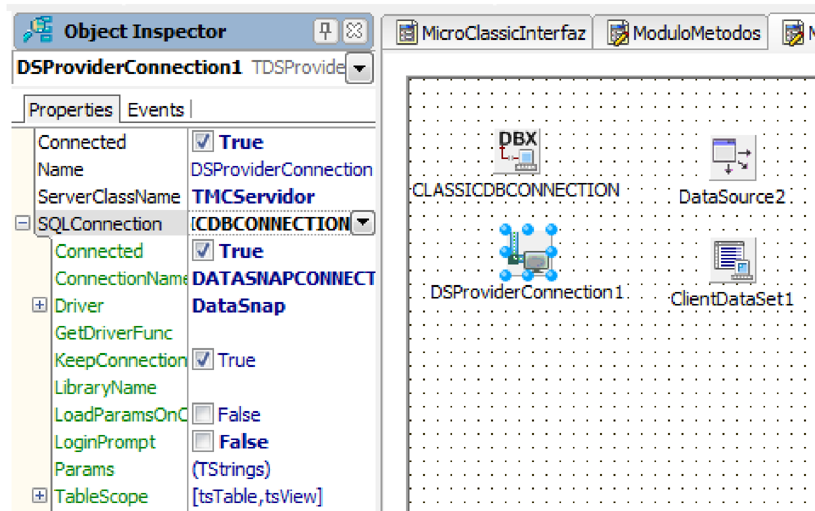
Como puede verse también se agrega un mensaje a la lista de la interfaz de administración, simplemente a efectos de comprobación durante las pruebas del nuevo proyecto.

En el módulo contenedor seleccionaremos el componente TDSServerClass y modificaremos su propiedad LifeCycle, asignándole el valor Server. De esta manera no existirá una conexión entre el servidor de aplicaciones y el de datos por cada cliente conectado, sino una única conexión para todos. Esto reduce drásticamente los recursos en uso por parte del RDBMS respecto a la implementación cliente/servidor del capítulo previo, en el que era necesario mantener abierta una conexión por cada cliente. Dado que cada operación sobre el RDBMS se efectúa en una transacción independiente, este uso compartido no conllevará ningún problema.

MicroClassicClient

La preparación de nuestro cliente resultará muy sencilla gracias a que en el diseño de la aplicación original separamos, en módulos independientes, la interfaz de usuario de los componentes relacionados con el acceso a datos. La interfaz en sí no sufrirá ningún cambio, sirve exactamente como está. Será en el módulo de datos donde introduzcamos las modificaciones necesarias.

Comenzaremos por seleccionar en el módulo de datos el componente `TSQLConnection` y cambiar sus propiedades `Driver` y `ConnectionName` para conectar con el servidor de aplicaciones[83]. Eliminaremos el `TSQLDataSet` y `TSQLQuery` y sustituiremos el `TDataSetProvider` por un `TDSProviderConnection`. Como se aprecia en la imagen inferior, la propiedad `ServerClass` de éste tomará el valor `TMCServidor`, apuntando así a la clase del servidor que expone los datos. Si damos el valor `True` a la propiedad `Connected` y ésta cambia su estado significará que todo va bien y se ha podido conectar con el servidor DataSnap sin problemas.



83 Es necesario tener el servidor ejecutándose, de forma que los componentes puedan conectar con él para obtener la información que precisan sobre elementos disponibles a fin de generar la clase *proxy* y también que el Inspector de objetos pueda introducir en las listas de ciertas propiedades los valores válidos.

A continuación seleccionamos el TClientDataSet, que antes estaba enlazado al TDataSetProvider, y cambiamos sus propiedades RemoteServer y ProviderName. La primera apuntará al TDSProviderConnection recién añadido, mientras que la segunda contendrá el nombre del TDataSetProvider que introdujimos en el servidor DataSnap (no hay más que desplegar la lista asociada y elegir el único valor posible).

El siguiente paso será seleccionar el TSQLConnection y, mediante la opción Generate DataSnap client classes, generar el módulo con la clase que actuará como *proxy* y facilitará el acceso al método getNextID. Dado que la clase en que se implementó dicho método se llamaba TMCServidor el *proxy* recibirá el nombre TMCServidorClient.

Ahora tendremos que cambiar el método asociado al evento OnNewRecord del TClientDataSet, a fin de que obtenga el siguiente identificador generado por el RDBMS a través del servidor DataSnap:

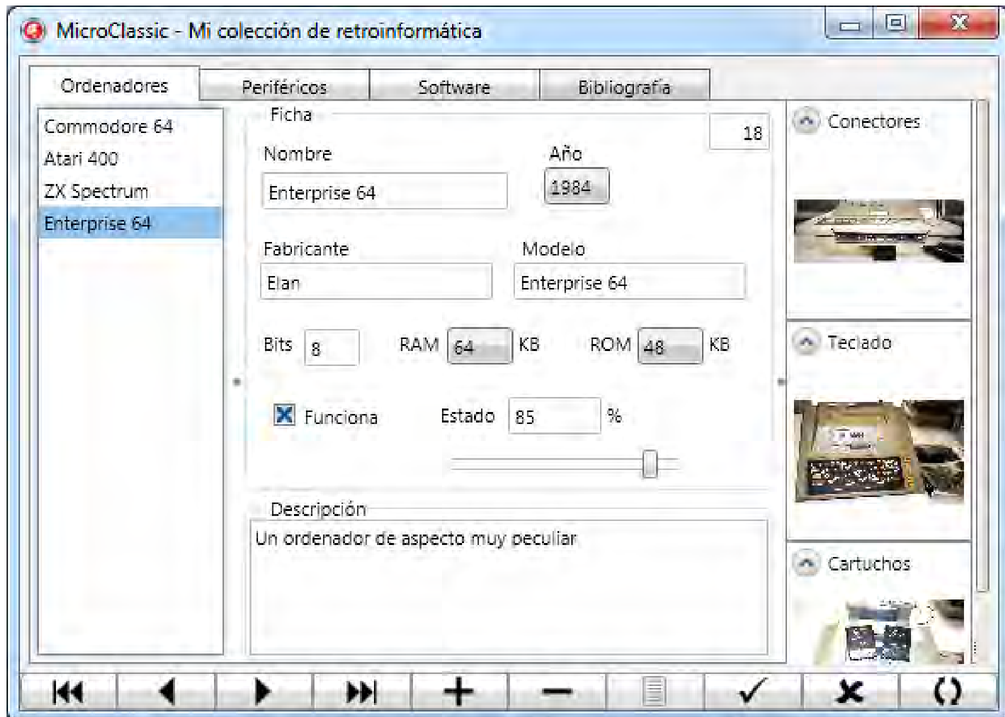
```
procedure TDataModule1.ClientDataSet1NewRecord(
         DataSet: TDataSet);
begin
 with TMCServidorClient.Create(
         CLASSICDBCONNECTION.DBXConnection) do
   DataSet.FieldByName('IDORDENADOR').Value := getNextID;
end;
```

Por último será necesario invocar al método ApplyUpdates del TClientDataSet, tanto en el momento en que se destruya el módulo de datos como cada vez que se produzca un cambio en los datos: adición de nuevo registro, modificación del actual o eliminación de una ficha. El paquete de cambios viajará hasta el servidor DataSnap y éste se encargará de hacerlo llegar al servidor de datos, comunicando en sentido inverso los problemas que pudieran detectarse.

> NOTA
>
> Si queremos que los clientes puedan ver los cambios que realizan otros usuarios además habrá que *refrescar* el TClientDataSet, invocando a su método Refresh, ya sea periódicamente o tras cada llamada a ApplyUpdates.

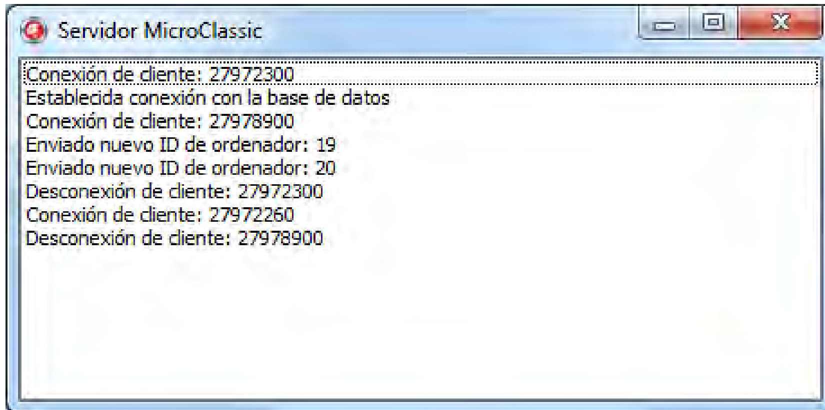
Una vez puesto en marcha el servidor, podemos ejecutar varias instancias del cliente, ya sea en distintas máquinas o en la misma, para comprobar su

funcionamiento. La interfaz del cliente no mostrará diferencia alguna respecto a la versión previa tal y como se aprecia en la imagen inferior. La información que está mostrando ha viajado por una conexión de red desde el RDBMS al servidor DataSnap y desde ésta hasta el cliente, mientras que los cambios que podamos efectuar viajarán en sentido inverso hasta llegar de nuevo al RDBMS.



La ejecución de varios clientes, incluso si se hace en la misma máquina, nos permitirá verificar que el servidor de aplicaciones, manteniendo una única copia de la clase `TMCServidor` y una sola conexión con la base de datos, es capaz de atender solicitudes procedentes de distintos puntos.

En la imagen de la página siguiente, correspondiente a la interfaz de administración del servidor, pueden verse dos conexiones activas y cómo al solicitar ambos clientes la adición de una nueva ficha se facilita a cada uno un identificador obtenido del RDBMS. Que los cambios efectuados por uno sean visibles para el otro dependerá de la política de actualización seguida: envío de cambios al servidor y solicitud de refresco de los datos locales.



Configuración de seguridad

En su configuración actual nuestro servidor `MicroClassicServer` permitiría que cualquiera accediese a sus servicios, siempre que conociese la dirección IP y el puerto en que se encuentra en funcionamiento. Habitualmente los servidores de aplicaciones requieren de los clientes una identificación, basada en un nombre de usuario y una contraseña, que opcionalmente puede llevar asociada la pertenencia a uno o más perfiles. Cada perfil tendría permitido o denegado el acceso a determinados servicios.

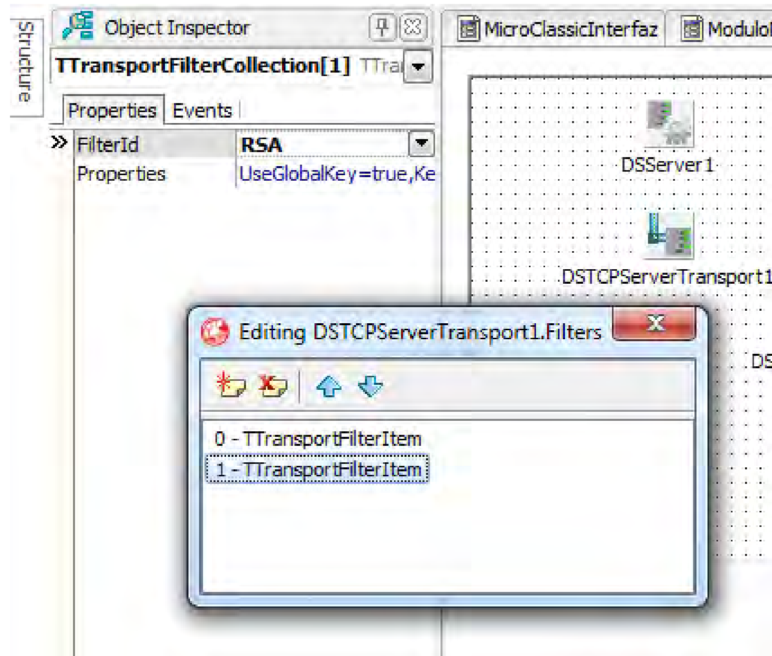
Un problema adicional de seguridad viene derivado del hecho de que la transmisión de los datos entre servidor y clientes, incluyendo el nombre de usuario y contraseña, se efectúan sin cifrar. Cualquiera que intercepte la comunicación podría no solamente ver los datos sino, una vez obtenidas las credenciales, hacerse pasar por un usuario y manipular la información.

Podemos mitigar estos problemas de acceso al servidor y a los datos que se transmiten entre él y los clientes cambiando la configuración del componente `TDSTCPServerTransport`. Por una parte agregaremos los filtros necesarios para cifrar la comunicación y por otra le asociaremos un componente adicional que será el encargado de gestionar la identificación de los clientes y la autorización de acceso a los servicios.

Filtros activos en el transporte

Es el componente encargado del transporte de la información entre servidor y clientes, en este caso un TDSTCPServerTransport, el encargado de gestionar los filtros por los que pasará el flujo de datos. Inicialmente la propiedad Filters de dicho componente, asumiendo que al generar el proyecto con el asistente no marcamos la opción de cifrado ni compresión, estará vacía.

Para agregar un filtro no tenemos más que abrir el editor asociado (en primer plano en la figura inferior) y hacer clic en el primero de los botones. Éste añade un TTransportFilterItem que a continuación, mediante el Inspector de objetos, habremos de personalizar. En nuestro caso añadiremos dos filtros, estableciendo la propiedad FilterId del primero a ZLibCompression y la del segundo a RSA.



Introducida esta modificación, cada vez que un cliente intente conectar con el servidor se producirá una negociación para establecer la configuración de los filtros, por ejemplo la clave asociada al algoritmo de cifrado. Dicha negociación tendrá éxito siempre que el cliente disponga de los elementos

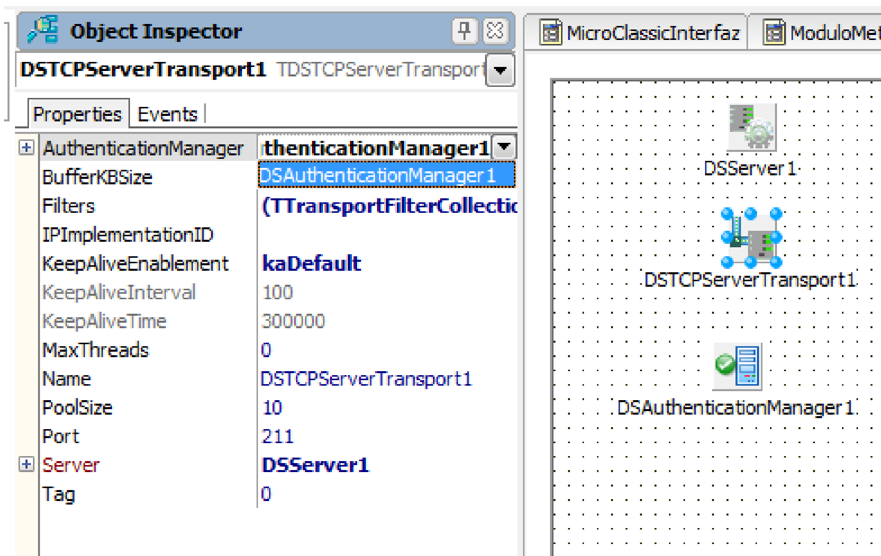
necesarios para cifrar/descifrar o comprimir/descomprimir. Nos aseguramos de que esto es así agregando a la cláusula `uses` del módulo de datos de `MicroClassicClient` los módulos `Data.DBXCompressionFilter` y `Data.DBXRSAFilter`:

```
uses
  Data.DBXCompressionFilter, Data.DBXRSAFilter;
```

No es necesario ningún cambio más. Aunque al ejecutar el servidor y los clientes no apreciaremos diferencia alguna en su comportamiento, si analizásemos el tráfico de red entre ambos antes y después de activar los filtros en el primer caso podríamos ver los datos, especialmente las cadenas de caracteres, mientras que en el segundo éstos ya no serían interpretables.

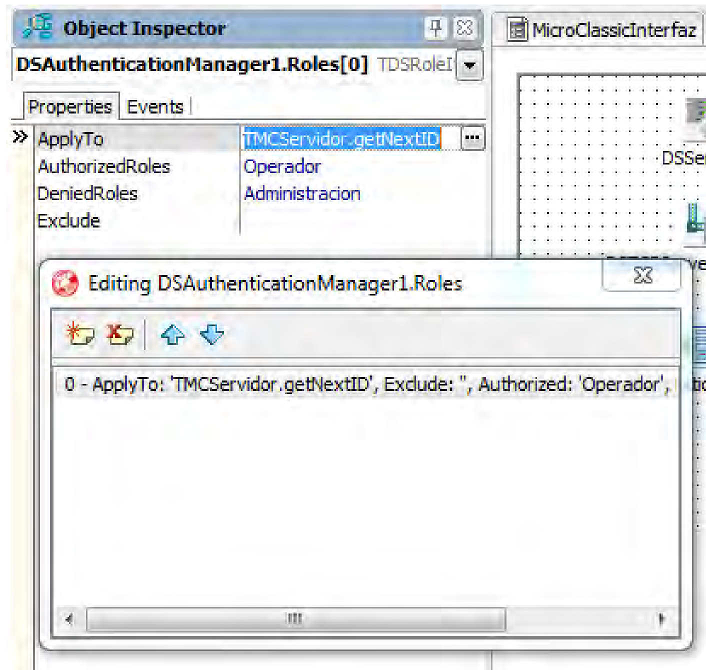
Identificación de usuarios

El segundo objetivo es limitar el acceso al servidor a aquellos usuarios que se identifiquen adecuadamente, para lo cual agregaremos al módulo que contiene el `TDSTCPServerTransport` un componente `TDSAuthenticationManager`, enlazando ambos mediante la propiedad `AuthenticationManager` del primero tal y como se muestra en la imagen inferior.



Un componente TDSAuthenticationManager sirve para gestionar tanto la identificación de los usuarios como la autorización de acceso a los servicios que ofrece el servidor. Cuenta con una propiedad llamada Roles que es una colección de objetos TDSRoleItem, cada uno de los cuales representa un conjunto de reglas mediante las que se autoriza o deniega la ejecución de ciertos servicios.

Al abrir el editor de la citada propiedad encontramos una lista vacía. Agregamos un nuevo elemento y recurrimos al Inspector de objetos para configurarlo, editando sus propiedades ApplyTo, AuthorizedRoles, DeniedRoles y Exclude tal y como puede verse en la imagen inferior. Todas ellas son listas de cadenas y cuentan con un editor específico.



La propiedad ApplyTo puede hacer referencia a una clase o un miembro de una clase, por ejemplo TMCServidor.getNextID. Mediante las propiedades AuthorizedRoles y DeniedRoles se indica qué perfiles pueden acceder a dicho método y a cuáles se les denegará su ejecución. En este ejemplo suponemos que el operador de la aplicación puede agregar nuevas fichas de ordenador, pero no un hipotético administrador que solamente generará informes.

A partir de este momento cada vez que un cliente conecte con el servidor éste le solicitará que se identifique, tras lo cual el componente TDSAuthenticationManager generará un evento OnUserAuthenticate. Éste recibirá, entre otros parámetros, el nombre de usuario y la contraseña, así como un argumento de salida llamado valid al que se ha de asignar True o False para permitir o denegar la conexión, respectivamente. Una forma de responder a este evento podría ser la siguiente:

```
procedure TServerContainer2.DSAuthManagUserAuthenticate(
  Sender: TObject;
  const Protocol, Context, User, Password: string;
  var valid: Boolean; UserRoles: TStrings);
begin
  MicroClassicSvr.LBMensajes.Items.Add(
     'Solicitud: ' + User + ', ' + Password);

  valid := (User = 'francisco') and (Password = 'charte');
end;
```

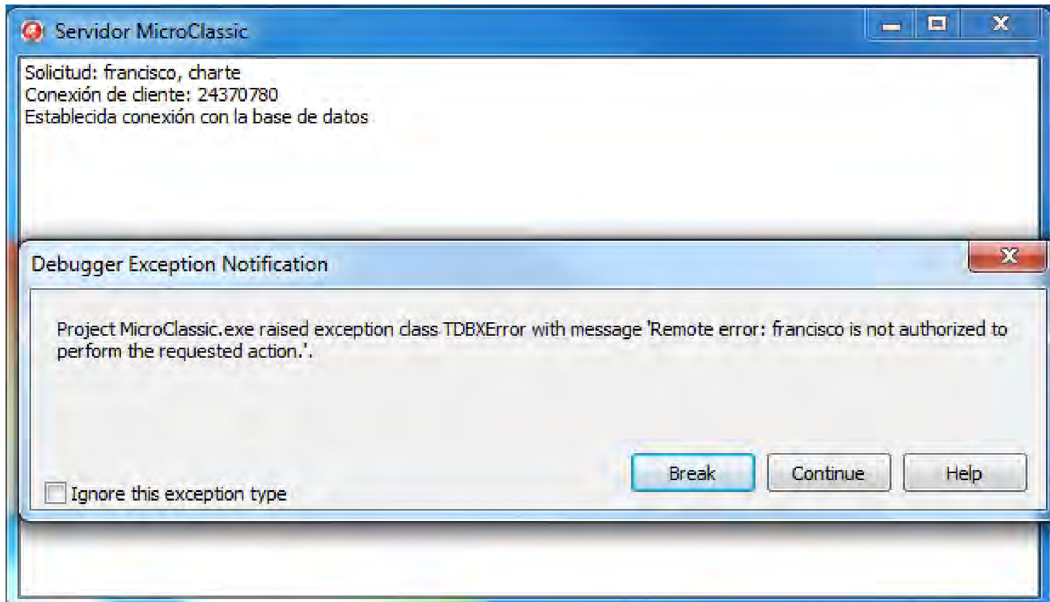
Obviamente introducir directamente en el código las credenciales que han de utilizarse no es la mejor opción, esto es sencillamente un ejemplo. En la práctica habría más de un usuario y el proceso de identificación requeriría una consulta sobre el RDBMS, en el que existiría una tabla con los usuarios que pueden conectar con el servidor. De esta manera se podría permitir el acceso a nuevos usuarios (o cancelar el de cuentas que expiran) sin más que agregar o eliminar una fila de una tabla.

Si tras introducir estos cambios en el servidor lo ejecutamos e intentamos acceder desde la aplicación cliente obtendremos un mensaje de error, ya que ésta no está aportando las credenciales necesarias. Podemos introducirlas directamente en la propiedad Params del componente TSQLConnection, agregando dos parámetros con los nombres DSAuthenticationUser y DSAuthenticationPassword, o bien establecerlos mediante código aprovechando el evento BeforeConnection, tal y como se muestra en el siguiente fragmento de código:

```
  with DATASNAPCONNECTION do begin
    Params.Values['DSAuthenticationUser'] := 'francisco';
    Params.Values['DSAuthenticationPassword'] := 'charte';
  end;
```

En caso de que al configurar el TDSAuthenticationManager hubiésemos definido la regla mostrada en la figura de la página anterior, limitando el acceso al método getNextID a los usuarios que pertenezcan al perfil Operador, el programa nos permitirá navegar por los datos y modificarlos,

pero no añadir nuevos ordenadores. En el mismo momento en que hagamos clic sobre el botón + del TBindNavigator aparecerá un mensaje de error[84] como el de la imagen inferior.



El problema es que por el momento no hemos asociado perfil alguno a los usuarios que conectan con el servidor de aplicaciones. Para ello tenemos básicamente dos alternativas: responder al evento OnUserAuthorize del componente TDSAuthenticationManager o bien indicar los perfiles asociados a un usuario que inicia sesión en el servidor usando el argumento UserRoles del evento OnUserAuthenticate.

Para el ejemplo que poníamos antes de identificación del usuario bastaría con agregar esta línea de código al final del método:

```
UserRoles.Append('Operador');
```

Dado que ahora el usuario pertenece al perfil al que se permite ejecutar el método getNextID se le permitirá agregar nuevas filas.

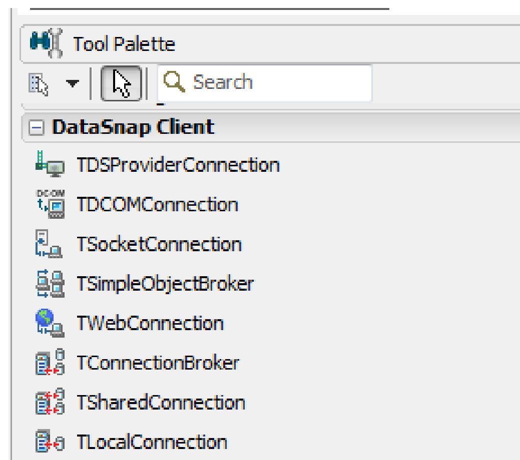
84 Para evitar que el usuario final real de la aplicación se enfrentase a este mensaje de error habría que controlar la excepción que desencadena el intento de ejecución del método, al igual que la conexión inicial con el servidor, ofreciendo un tratamiento más amigable.

Conclusión

Como hemos podido ver en la práctica en este capítulo, Delphi nos ofrece con DataSnap un método potente, flexible y bastante sencillo para la construcción de aplicaciones distribuidas. En éstas el pilar fundamental es el servidor de aplicaciones, un programa que puede ejecutarse como cualquier otro o bien como un servicio de Windows atendiendo las solicitudes efectuadas por los clientes que le llegan a través de una conexión de red.

Introducir mecanismos de seguridad, como el cifrado de la información transmitida entre servidor y clientes, la identificación de los usuarios que conectan con el servidor y el control de acceso a los servicios basado en perfiles, es una tarea también sencilla como hemos podido comprobar en el último punto.

Aunque nos hemos centrado en el uso del protocolo TCP para comunicar servidor y cliente, ya que éste es una aplicación nativa y dicho protocolo resulta más eficiente, agregar la posibilidad de usar un canal por HTTP o HTTPS para otro tipo de clientes es una tarea prácticamente trivial. Aunque en la página DataSnap Client de la Paleta de herramientas (véase la figura inferior) podemos encontrar otras alternativas para la comunicación entre servidor y clientes: `TDCOMConnection`, `TLocalConnection`, etc., en su mayor parte éstos son métodos heredados de versiones previas de Delphi que se mantienen en la versión XE2 únicamente por razones de compatibilidad.



A continuación

Con este capítulo concluye la parte del libro dedicada a las bases de datos y su uso en aplicaciones monolíticas (una sola capa), cliente/servidor (dos capas) y distribuidas (tres o más capas). La siguiente sección, de menor extensión, estará dedicada al desarrollo de aplicaciones web con Delphi.

En el capítulo siguiente tomaremos contacto con las posibilidades que ofrece Delphi para la creación de aplicaciones web, fundamentos que servirán como base en el capítulo posterior para desarrollar una interfaz web que facilite el acceso a los servicios ofrecidos por un servidor DataSnap.

Apartado III: Aplicaciones web

En los últimos años la WWW (*World Wide Web*) se ha convertido en una plataforma más para el despliegue de aplicaciones hacia los clientes, con importantes ventajas como por ejemplo la independencia del hardware y el software que empleen los usuarios: solamente se precisa un navegador web.

Actualmente existen infinidad de herramientas e incluso lenguajes de programación diseñados a medida para el desarrollo de aplicaciones web. En los dos capítulos que componen esta tercera parte del libro conoceremos lo que ofrece Delphi en este campo, partiendo de que ni el lenguaje Delphi ni el entorno son específicos para este tipo de proyectos.

- **Capítulo 8: Aplicaciones IntraWeb**
- **Capítulo 9: Interfaces web para aplicaciones DataSnap**

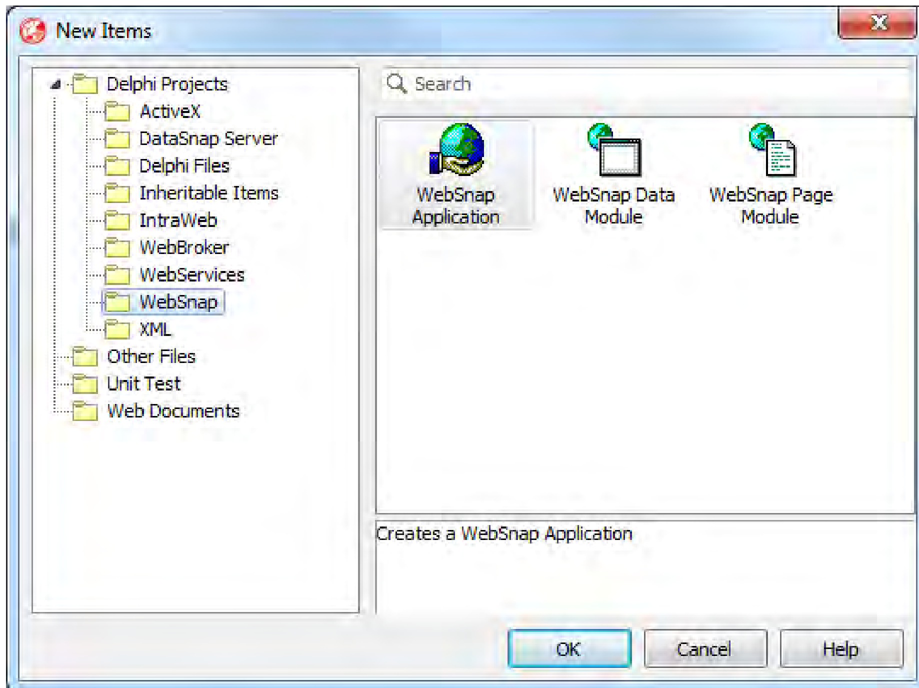
Capítulo 8: Aplicaciones IntraWeb

Delphi XE2 cuenta con un gran conjunto de componentes, plantillas y asistentes que facilitan el desarrollo de aplicaciones con capacidad de comunicarse a través de redes basadas en la pila de protocolos TCP/IP, como es el caso de Internet. Un ejemplo de ello son los componentes Indy, con los que podríamos crear prácticamente cualquier tipo de servidor o cliente.

De todo ese gran abanico de posibilidades centraremos nuestra atención en un tipo específico de proyecto: las aplicaciones que, ejecutándose en un servidor, pueden ser usadas desde una navegador web cualquiera, generalmente de forma remota por parte de usuarios que pueden encontrarse virtualmente en cualquier parte del mundo.

Tipos de proyectos web en Delphi

Si echamos un vistazo a la ventana New Items de Delphi XE2 encontraremos múltiples tipos de proyectos relacionados con la web. Se agrupan en las categorías IntraWeb, WebBroker y WebSnap.



Estos tres tipos de proyecto web han sido heredados de versiones previas de Delphi. Algunos como WebBroker[85] surgieron hace más de una década y apenas han evolucionado en los últimos años. Otros a pesar de ser más recientes, como en el caso de WebSnap que apareció en Delphi 6, carecen ya de soporte por parte de Embarcadero[86], incluyéndose en la versión XE2 únicamente por razones de compatibilidad con versiones anteriores.

---

85 WebBroker fue una de las novedades más importantes introducidas por Delphi 3, versión lanzada en 1997.

86 En http://docwiki.embarcadero.com/RADStudio/en/Using_WebSnap_Index (documentación electrónica de Delphi XE2) el fabricante desaconseja su uso al ser una tecnología obsoleta.

De las tres opciones mencionadas IntraWeb es la de uso más recomendable, primero porque sigue actualizándose y cuenta con el soporte de la empresa que desarrolla estos componentes, que es Atozed Software, y segundo porque el procedimiento de desarrollo de aplicaciones web con IntraWeb se asemeja al habitual en Delphi para el resto de proyectos: existe una superficie de trabajo sobre la que se colocan componentes y controles visuales. Con WebBroker y WebSnap la metodología es muy distinta ya que la interfaz se diseña editando páginas HTML/CSS y es necesario responder a solicitudes HTTP.

Por las razones expuestas este capítulo describe el procedimiento a seguir para la creación de aplicaciones web usando IntraWeb, al ser éste el método preferente en Delphi XE2 y, al parecer, el único que seguirá actualizándose y estando presente en versiones futuras del producto, en detrimento de los citados WebBroker y WebSnap.

En cualquier caso ha de tenerse en cuenta que ninguna de las tres opciones mencionadas están preparadas para el desarrollo multiplataforma, al tener dependencias de la biblioteca de clases VCL. Podemos crear una aplicación web que se integre con diversos servidores web, como IIS (*Internet Information Server*) o el popular Apache, pero siempre funcionando sobre Windows.

> NOTA
>
> Aquellos que prefieran desarrollar sus aplicaciones web con el lenguaje más empleado para tal fin, PHP, tienen a su disposición el producto RadPHP XE2. El entorno es el mismo de Delphi y la forma de desarrollar muy parecida, si bien los componentes disponibles difieren y el lenguaje base es PHP. Una aplicación RadPHP puede conectar con un servidor DataSnap creado con Delphi y consumir sus servicios, siendo posiblemente ésta la vía más efectiva para ofrecer una interfaz de usuario web para cualquier servidor DataSnap. RadPHP también contempla el desarrollo de aplicaciones para dispositivos móviles y para Facebook.
>
> Puedes encontrar más información sobre este producto en `http://www.danysoft.com/general/embarcadero-radphp-xe2.html`.

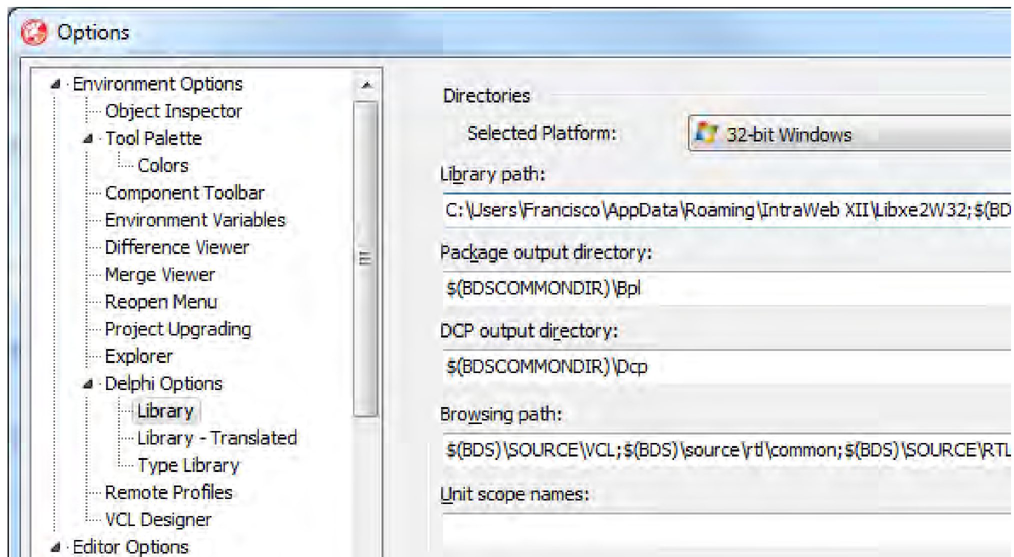
Actualización de IntraWeb

Es recomendable que lo primero que hagamos, tras haber instalado Delphi XE2 y los correspondientes *Update*, sea actualizar la propia biblioteca IntraWeb. Las diferencias entre la versión que se instala con Delphi XE2, incluso tras el *Update 3*, y la versión más actual de IntraWeb que ofrece Atozed son importantes, existiendo un conjunto considerable de nuevas clases y cambios tanto en las denominaciones de algunos miembros como en las listas de parámetros de métodos y eventos.

La actualización de IntraWeb la obtendremos directamente de la web del fabricante: www.atozed.com/IntraWeb/Download/Download.EN.aspx (véase imagen inferior). Para su instalación precisaremos una clave de activación que, al ser propietarios de una licencia de Delphi XE2, podemos obtener sin coste. El enlace a seguir en la misma página web es Bundled Key Request. El proceso de registro requiere la introducción de una dirección de correo y una contraseña y, una vez indicada la versión y edición de Delphi, recibiremos por correo electrónico dicha clave.

Antes de ejecutar la utilidad de instalación es importante que eliminemos de la carpeta en la que está instalado Delphi XE2 todos los módulos DCU y BPL correspondientes a la versión actualmente instalada de IntraWeb. En www.atozed.com/IntraWeb/FAQ/InstallingIntraWeb.EN.aspx se indican detalladamente los pasos a seguir.

Completados estos pasos previos, no tenemos más que instalar IntraWeb, introducir la clave de activación que obtuvimos y reiniciar Delphi. En la ventana que aparece momentáneamente mientras se abre el entorno debe aparecer un icono adicional haciendo referencia a IntraWeb XII. Si no es así debemos abrir la ventana de opciones de Delphi, con la opción Tools>Options, y verificar que en el apartado Library Path de la sección Delphi Options>Library aparece la ruta donde se ha instalado IntraWeb.



ADVERTENCIA

En el resto del capítulo se asume que se ha instalado la actualización 12.1.20 de IntraWeb y se hace referencia a elementos que no están disponibles en la versión de los componentes que incluye Delphi XE2, algo que debe tener en cuenta el lector puesto que las diferencias son, como se ha indicado antes, notables.

Estructura de una aplicación IntraWeb

Toda aplicación IntraWeb se compone de una serie de elementos básicos, entre los que destacan el controlador del servidor, el módulo de sesión de usuario y los módulos asociados a elementos de interfaz de usuario como son los formularios y paneles. Como veremos en un instante, Delphi XE2 dispone de un asistente que se encarga de generar un proyecto básico con todos esos elementos ya preparados.

Las aplicaciones IntraWeb pueden ejecutarse en dos modos diferentes: como un programa cualquiera, con su interfaz de usuario, o bien como una extensión ISAPI (en forma de servicio). Es un esquema similar al de los servidores DataSnap que conocimos en el capítulo previo, si bien en este caso existe un servicio que es el servidor web (normalmente IIS) y la aplicación IntraWeb funciona como un complemento de dicho servidor.

Inicialización de una aplicación IntraWeb

En un proyecto IntraWeb ha de existir un módulo principal[87] que se encargue de poner en marcha la ejecución, con independencia de que el servidor vaya a funcionar como una aplicación estándar o bien como un servicio. Los elementos fundamentales de dicho módulo serán dos:

- Una referencia al módulo `IWStart` en su cláusula `uses`. En `IWStart` se encuentra la definición de la clase `TIWStart`, derivada de `TObject`, como único elemento público[88].
- La llamada al método de clase `Execute` de `TIWStart`. Éste toma como argumento un valor de tipo `Boolean` con el que indicaremos si el servidor se ejecutará como una aplicación estándar con interfaz de usuario: `True`, o bien como un servicio: `False`.

---

87 El archivo de código asociado al proyecto y con encabezado `program` en lugar de `unit`.

88 El módulo `IWStart` en la versión de IntraWeb que incluye Delphi XE2 sustituye a los módulos `IWMain` e `IWInitService` de versiones previas, unificando el proceso de puesta en marcha de la aplicación.

Lo único que diferencia a un proyecto IntraWeb que se ejecuta como aplicación estándar de otro que lo hace como servicio es, por tanto, el parámetro que se entrega al método `Execute` de la clase `TIWStart`. Esto nos permite usar el primer modo durante el desarrollo y depuración y cambiar al segundo al distribuir el proyecto sin más que sustituir el valor `True` por `False`.

La clase `TIWStart` se ocupa exclusivamente de lanzar la ejecución de la aplicación IntraWeb. Para un correcto funcionamiento ésta necesita saber qué clase va a actuar como controlador del servidor y cuál será el formulario por defecto.

El controlador del servidor

La mayor parte de los parámetros de funcionamiento de una aplicación IntraWeb los gestionará el controlador del servidor: un objeto de una clase descendiente de `TIWServerControllerBase`. En un proyecto IntraWeb solamente puede existir una clase actuando como controlador y ésta se *registrará* a sí misma mediante una llamada al procedimiento `SetServerControllerClass` de `TIWServerController`[89].

No es necesario que creemos explícitamente un objeto de la clase que va a actuar como controlador del servidor. Con la operación de registro completada, será el propio motor de IntraWeb el que cree un único objeto a partir de ella.

El controlador cuenta con un conjunto de propiedades, heredadas de `TIWServerController`, que determinan la configuración de la aplicación web, entre ellas:

- `AppName`, `Description`, `DisplayName`: Identifican a la aplicación estableciendo su nombre y facilitando una descripción. El contenido de `AppName` servirá para dar nombre al servicio Windows, si la aplicación se instala como tal, nombre que ha de ser único entre los servicios existentes en el sistema. Asimismo servirá como base para la denominación de las *cookies* creadas en el navegador de los clientes que accedan a la aplicación. `Description` facilitará la

89 Esta clase también está derivada de `TIWServerControllerBase` y, aparte de los miembros heredados, expone como único elemento de interés el procedimiento de clase `SetServerController`.

descripción del servicio para mostrar en la consola de administración de servicios de Windows. Finalmente DisplayName suele utilizarse en caso de que la aplicación tenga que mostrar algún cuadro de diálogo, usando su contenido como título.

- Port, SSLOptions, SessionTimeout: Controlan parámetros de comunicación entre cliente y servidor como el puerto HTTP, la configuración de seguridad SSL y el tiempo de expiración de sesión.
- StyleSheet: Establece una hoja de estilos CSS a aplicar a todos los formularios de la aplicación. Esta propiedad es un objeto TIWFileReference y puede contener la ruta de un archivo o un URL.
- AllowMultipleSessionsPerUser, AuthBeforeNewSession: Por defecto una aplicación IntraWeb asocia una sesión a cada usuario, pero dando el valor True a la primera propiedad se permitirá el inicio de varias sesiones. La segunda controla la solicitud de autorización de acceso antes de crear cada nueva sesión.
- CacheDir, FilesDir, InternalFilesDir, TemplateDir: Almacenan la ruta de carpetas que serán utilizadas para el almacenamiento de datos en caché, búsqueda de archivos y plantillas.
- GUIActive: La aplicación puede saber si está ejecutándose como servicio o en modo estándar consultando esta propiedad. En caso de que sea False indicará que está ejecutándose como un servicio.

Asimismo el controlador expone una serie de eventos que podemos aprovechar para intervenir en acciones como la creación o destrucción de sesiones, la respuesta a la pulsación del botón de retroceso del navegador por parte del usuario, etc. Algunos de esos eventos son:

- OnNewSession, OnCloseSession: Notifican el inicio de una nueva sesión y el cierre de una sesión existente. En ambos casos van acompañados de un parámetro de tipo TIWApplication que representa la nueva sesión o sesión que va a destruirse, y que puede ser usada, por ejemplo, para el almacenamiento de datos asociados a la misma.
- OnBrowserCheck: Inmediatamente después de haber creado una sesión se produce este evento que, además del parámetro TIWApplication que identifica la sesión, recibe también un objeto de tipo TBrowser. A partir de la información obtenida del navegador

del cliente se crea un objeto, de una clase derivada de TBrowser, con el objetivo de que el contenido se adapte lo mejor posible. Dado que este parámetro se recibe por referencia es posible modificar su contenido, lo cual abre las puertas a procedimientos propios de detección del navegador y la aportación de implementaciones a medida de clases descendientes de TBrowser.

- OnSessionTag, OnUserTag: Se generan cuando al analizar el código que será enviado al navegador se encuentran marcas que se ajustan al patrón {%Session.XXX%} y {%User.XXX%}, respectivamente.
- OnBeforeRender, OnAfterRender: Tienen lugar antes y después de componer la página de respuesta a una solicitud. El primero recibe los parámetros AForm y VNewForm, ambos objetos de tipo TIWBaseForm y que pueden ser usados para procesar el contenido original y aportar una versión modificada. El segundo evento únicamente recibe el parámetro AForm con el formulario que finalmente recibe el cliente.

Los servicios que ofrece el controlador de una aplicación IntraWeb pueden necesitarse desde cualquier punto del proyecto, por ello contamos con varios métodos para hacer referencia al objeto que actúa como controlador. El más inmediato consiste en usar el método IWServerController del propio controlador, para lo cual será necesario agregar una referencia al módulo en que se encuentra definida la clase derivada de TIWServerControllerBase. Un método alternativo es añadir una referencia al módulo IWGlobals de IntraWeb, lo cual nos permite usar las variables globales gServerController y gSC, de manera indistinta, para recuperar la referencia al controlador.

Formularios

Es función del controlador, entre otras tareas, atender las solicitudes que lleguen desde los clientes y devolver la respuesta adecuada. Cada solicitud tendrá asociado un URL que identificará el contenido o función requeridos. En nuestro proyecto habrá una correspondencia entre URLs y formularios IntraWeb, existiendo tantos de éstos como páginas dinámicas diferentes compongan la aplicación[90].

90 El proyecto puede contener asimismo páginas estáticas y otros archivos de contenido.

Cada formulario será definido como una nueva clase descendiente de `TIWAppForm`, de la que heredarán la funcionalidad genérica. Al igual que los formularios VCL, éstos también cuentan con los eventos `OnCreate` y `OnDestroy` que pueden ser aprovechados para tareas de inicialización y liberación de recursos, respectivamente.

Los formularios tienen acceso a la sesión del usuario activo a través de la propiedad `WebApplication`. Ésta mantiene la referencia al objeto `TIWApplication` que representa la sesión y que era recibido como parámetro en algunos de los eventos del controlador del servidor. Esto permitirá al formulario, por ejemplo, saber qué URL ha desencadenado su apertura, información que se almacena en la propiedad `ReferringURL` de la sesión.

De los múltiples formularios que pueden existir en una aplicación IntraWeb, cada uno de ellos alojado en un módulo independiente como clase derivada de `TIWAppForm`, es necesario establecer cuál será el que se abra por defecto al acceder al servidor. Para ello ese formulario principal ha de invocar al método de clase `SetAsMainForm` que hereda de `TIWAppForm`. El procedimiento es similar al registro del controlador del servidor y suele llevarse a cabo en la sección de inicialización del módulo correspondiente.

> NOTA
>
> En cualquier momento es posible recuperar una referencia al tipo del formulario principal de la aplicación (su clase) mediante la variable global `gMainFormClass`. Al igual que `gSC` y `gServerController`, anteriormente mencionadas, ésta se encuentra definida en el módulo `IWGlobal` que será necesario agregar a la cláusula `uses`.

Sesión de usuario

Cada vez que un cliente conecta con una aplicación IntraWeb se genera una nueva sesión, representada por un objeto `TIWApplication` que es recibido como argumento en algunos de los eventos y que también puede obtenerse mediante la propiedad `WebApplication` de `TIWAppForm`.
`TIWApplication` cuenta, entre otras, con una propiedad llamada `Data` cuya finalidad es almacenar información asociada a la sesión de cada usuario.

Aunque podríamos usar la propiedad Data para guardar virtualmente cualquier dato, ya que su tipo es TObject, lo habitual es que se utilice un objeto de una clase derivada de TIWUserSessionBase. Ésta es similar a un módulo de datos, de hecho cuenta durante la fase de diseño con una superficie que actúa como contenedor y permite alojar componentes. Al tratarse de una clase que definiríamos nosotros mismos, normalmente en un módulo a propósito para ello, podemos declarar todos los atributos que sean necesarios para almacenar datos relacionados con la sesión de trabajo de cada usuario que utilice la aplicación.

Aprovechando el evento OnNewSession del controlador, citado anteriormente, no habría más que usar una sentencia como la mostrada en el siguiente fragmento de código para crear el objeto que contendrá la información de sesión y guardarlo en la propiedad Data:

```
procedure TIWSC.IWServerControllerBaseNewSession(
  ASession: TIWApplication; var VMainForm: TIWBaseForm);
begin
  ASession.Data := TIWUserSession.Create(nil);
end;
```

De hecho el asistente que incorpora Delphi para crear aplicaciones DataSnap ya incluye el módulo con la clase derivada de TIWUserSessionBase y el código asociado al evento OnNewSession, aparte de una función llamada UserSession que facilita el acceso a los datos de sesión desde cualquier punto de la aplicación, sin necesidad de leer la propiedad Data y convertirla al tipo correspondiente.

Clases de navegadores

Entre las ventajas que ofrece el diseño de aplicaciones con IntraWeb, respecto a otras opciones heredadas, se encuentra su capacidad para adaptar el contenido enviado a los clientes en función del navegador que estén usando. Para ello el controlador analiza la información que se recibe sobre el navegador en la propia solicitud y, según una serie de reglas, procede a crear un objeto de la clase de navegador apropiada.

Todas las clases de navegador están derivadas de TBrowser, tipo definido en IW.Browser.Browser, y cada una de ellas se aloja en un módulo propio. Todos esos módulos tienen el prefijo IW.Browser, mientras que el sufijo

denota al navegador concreto: `Android`, `Firefox`, `Chrome`, `InternetExplorer`, `Safari`, `SafariMobile` u `Other`, tal y como puede apreciarse en la imagen inferior. Los nombres de las clases coinciden con el de esos sufijos pero precedidos de la letra `T`, como es habitual.

```
procedure TIWServerController.IWServerControllerBaseBrowserCheck(
  aSession: TIWApplication; var rBrowser: TBrowser);
begin
  if rBrowser is TOther then
      rBrowser := IW.Browser.
```

```
unit    SafariMobile;
unit    Android;
unit    Firefox;
unit    Chrome;
unit    Safari;
unit    Other;
unit    Browser;
unit    InternetExplorer;
```

Al crear un objeto de una de esas clases, a través del habitual constructor `Create`, es preciso facilitar como parámetro un valor en punto flotante indicando la versión del navegador a la que es necesario ajustarse. Por ejemplo:

```
rBrowser := TInternetExplorer.Create(6.0);
```

Mediante la propiedad `IsSupported` de `TBrowser`, heredada por las demás clases, podemos saber si IntraWeb está preparado o no para adaptarse a dicho navegador. La propiedad `MinSupportedVersion`, por su parte, indica cuál es la versión más antigua para la que puede generarse contenido. Esto nos permite realizar comprobaciones como la siguiente:

```
with rBrowser do
  if (not IsSupported) then
    if ClassNameIs('TChrome') then
      rBrowser := TChrome.Create(MinSupportedVersion)
    else if ClassNameIs('TFirefox') then
      rBrowser := TFirefox.Create(MinSupportedVersion)
    // Comprobar otros navegadores...
    else
      // Navegador por defecto
      rBrowser := TInternetExplorer.Create(6.0);
```

> ADVERTENCIA
> Para poder usar el tipo `TBrowser`, por ejemplo al responder al evento `OnBrowserCheck`, es necesario agregar manualmente una referencia al módulo `IW.Browser.Browser` en la cláusula `uses` de nuestro módulo de código. Asimismo la creación de un objeto de la clase `TChrome`, `TFirefox` o cualquier otra de las derivadas de la anterior implica insertar la referencia al módulo correspondiente: `IW.Browser.Chrome`, `IW.Browser.Firefox`, etc.

Componentes IntraWeb

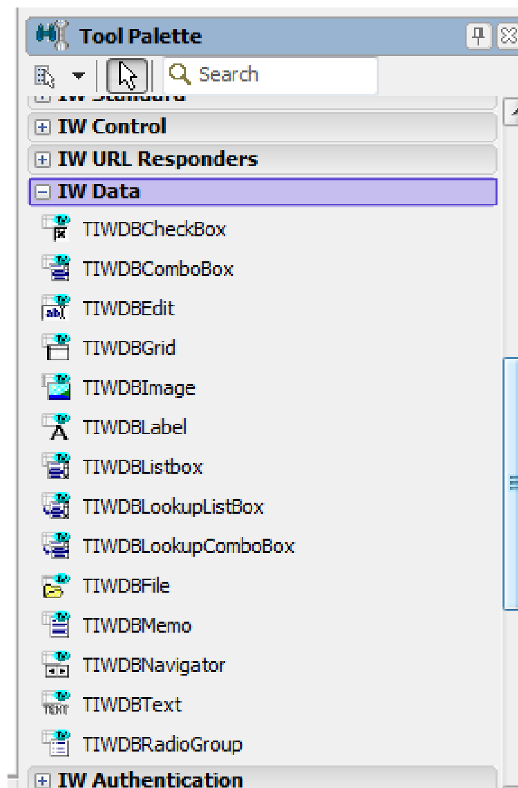
Ya sabemos que un proyecto de aplicación IntraWeb se compone de varios módulos de apoyo: el controlador del servidor, la clase de almacenamiento de datos de sesión y el código de puesta en marcha del programa, y uno o varios módulos que actuarán como interfaz de usuario: los formularios.

Al igual que ocurre en una aplicación VCL o FMX, los formularios IntraWeb están preparados para contener componentes visuales. Sin embargo al ejecutar la aplicación esos controles lo que hacen es generar código HTML, marcas que el navegador interpretará y convertirá en el elemento que corresponda. La apariencia de los controles visuales en el navegador no ha de ser necesariamente idéntica a la que veremos durante la fase de diseño.

Podemos agrupar los componentes IntraWeb en tres categorías diferentes según su función:

- **Controles visuales**: Se encuentran en la página IW Standard de la Paleta de herramientas. Una buena parte de ellos replican la funcionalidad de los controles VCL, como son los botones: `TIWButton`, recuadros de texto: `TIWEdit`, listas: `TIWList` o menús: `TIWMenu`. Otros aportan elementos específicos de una aplicación web como son los *applets* Java: `TIWApplet`, contenidos en Flash: `TIWFlash`, QuickTime: `TIWQuickTime` o Silverlight: `TIWSilverlight`, etc.

- **Controles conectados a datos**: Al igual que la VCL, IntraWeb cuenta con un grupo de controles específicos para operar sobre información procedente de una base de datos. Los encontramos en la página `IW Data` (véase la imagen inferior) con el prefijo `TIWDB`, caracterizándose por la presencia de las propiedades `DataSource` y `DataField`[91]. Los más usuales son `TIWDBEdit`, `TIWDBCheckBox`, `TIWDBListBox` y `TIWDBRadioGroup`. El componente `TIWDBNavigator` es el encargado de facilitar la navegación por las filas de un conjunto de datos.



- **Componentes de control**: Se trata de componentes no visuales que ofrecen mecanismos de control tales como la autenticación de los usuarios, la vinculación de formularios a URLs, gestión de

91 Esta propiedades fueron descritas al principio del quinto capítulo, dedicado a LiveBindings.

notificaciones en aplicaciones AJAX, etc. Se encuentran repartidos por las páginas IW Control, IW Authentication e IW URL Responders de la Paleta de herramientas.

En total son varias decenas de componentes, cada uno de ellos con un conjunto de propiedades, métodos y eventos igualmente importante. Sobre ninguno de estos elementos encontraremos información en la documentación electrónica de Delphi XE2, únicamente un enlace a la web de la empresa Atozed que es la responsable del producto.

IntraWeb en la práctica

Conocidos los fundamentos sobre IntraWeb, la estructura general de una aplicación de este tipo y los tipos de componentes que es posible usar en ella, nos proponemos ahora desarrollar un sencillo proyecto que nos permita poner en práctica lo explicado y completar así la visión global del proceso a seguir para crear una aplicación web en Delphi XE2.

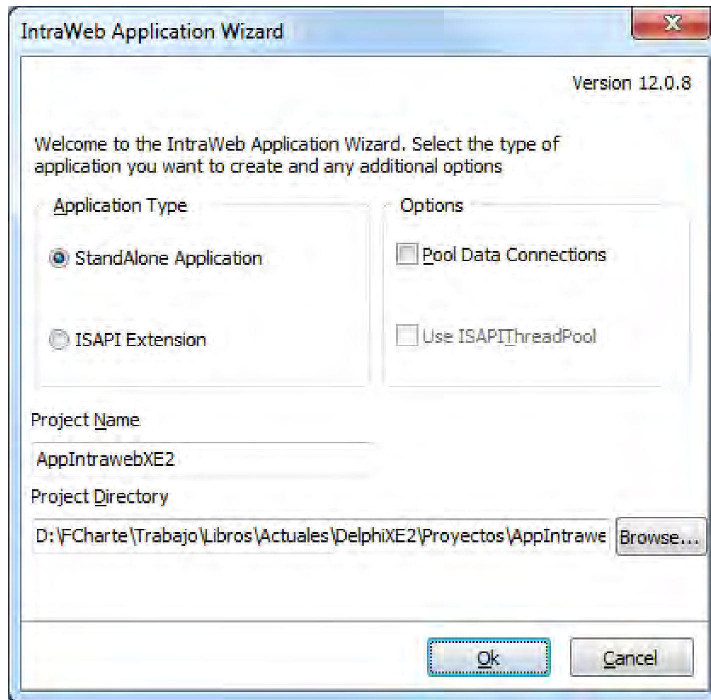
Partimos como es lógico creando un nuevo proyecto seleccionando del grupo IntraWeb de la ventana New Items el elemento IntraWeb Application Wizard. Se pone en marcha el asistente que consta de un único paso correspondiente a la imagen que puede verse en la página siguiente.

Elegiremos como tipo de aplicación StandAlone Application, ya que nos resultará en principio más fácil para depurar la aplicación y no precisaremos un servidor web como IIS. En este modo la aplicación incluye su propio servidor.

Hemos de especificar la ruta en la que se almacenarán los módulos del proyecto generados por el asistente, así como asignar un nombre para el proyecto. En el apartado Options indicaremos, en caso de que la aplicación vaya a utilizar conexiones a datos, si queremos agregar los elementos necesarios para contar con un *pool* de conexiones[92].

---

92 Una aplicación web con conexión a datos y que atienda a un gran número de usuarios estará continuamente abriendo y cerrando conexiones, una tarea que puede consumir un cantidad de tiempo y recursos importante. Mantener un *pool* de conexiones preparadas, con un correcto dimensionamiento de su tamaño, ayudará a mitigar este problema.



Dependiendo de las opciones marcadas el proyecto constará inicialmente de tres o cuatro módulos de código, aparte del módulo de proyecto, con la siguiente denominación:

- `ServerController.pas`: Contiene la definición de la clase que actuará como controlador del servidor, derivada de `TIWServerControllerBase`, así como el código correspondiente a los métodos `IWServerController` y `UserSession` explicados anteriormente. También estarán implementados varios de los eventos, a fin de asociar como datos de cada nueva sesión un objeto `TIWUserSession` y, si procede, gestionar el *pool* de conexiones.
- `UserSessionUnit.pas`: Es el módulo en el que se define la clase `TIWUserSession`, derivada de `TIWUserSessionBase` y cuya finalidad es facilitar la conservación de información durante el tiempo que dura una sesión de trabajo de un usuario. El diseñador nos permite introducir en este módulo componentes no visuales como si de un módulo de datos se tratase.

- Unit1.pas: En principio el proyecto contará con un formulario, una clase derivada de TIWForm, definida en este módulo e inicialmente vacío. Como si de una interfaz FMX o VCL se tratase, usaremos el diseñador para introducir en el formulario los componentes que necesitemos y personalizarlos aspecto y comportamiento a través de sus propiedades y eventos.
- DataModuleUnit.pas: En caso de que hubiésemos marcado la opción Pool Data Connections en el asistente, el proyecto también contará con este módulo. En él se define una clase derivada de TDataModule, a la que se agregarán los componentes de conexión a datos, gestionada desde el controlador del servidor.

El objetivo de esta aplicación será mantener una lista de ordenadores, únicamente su nombre y estado de funcionamiento, independiente para cada usuario que acceda. La información estará, por tanto, asociada a la sesión de cada cliente. Además queremos que cualquiera de ellos pueda saber qué conexiones se han producido con el servidor, indicando el navegador y dirección IP de origen.

Partiendo de los módulos en el estado en que los genera el asistente, veamos cuáles son los pasos a seguir para alcanzar el objetivo que nos hemos propuesto.

Información de sesión

Comenzaremos por introducir en el módulo UserSessionUnit los elementos que necesitamos para mantener la información asociada a cada sesión. El primero de ellos será una sencilla clase con un constructor y dos atributos asociados a los datos de cada ordenador:

```
type
  TOrdenador = class
  public
    nombre:   String;
    funciona: boolean;

    constructor Create(nombre: String;
                       funciona: boolean);
  end;

implementation
```

```
constructor TOrdenador.Create(nombre: String;
                              funciona: boolean);
begin
  self.nombre := nombre;
  self.funciona := funciona;
end;
```

TOrdenador es sencillamente un nuevo tipo de dato, tipo que usaremos para una matriz definida en la clase TIWUserSession que, a la postre, es de la que se creará un objeto asociado a cada sesión. La definición de dicha clase quedará así:

```
  TIWUserSession = class(TIWUserSessionBase)
  private
    ordenadores: array[0..100] of TOrdenador;
    ultimo: integer;
  public
    constructor Create(aOwner: TComponent);
    function getOrdenador(i: integer): TOrdenador;
    procedure addOrdenador(o: TOrdenador);
  end;
```

Cada sesión tendrá asociada una matriz de objetos TOrdenador y un entero que indicará cuál es el último elemento usado en la misma. Redefinimos el constructor para inicializar ese atributo y definimos también dos métodos que facilitarán la recuperación y adición de objetos.

NOTA

Para este sencillo ejemplo se ha asumido que cada usuario no va a agregar más de 100 elementos a la lista. En la práctica el sistema lógicamente debería tomar en consideración esa posibilidad, ajustando el tamaño de la matriz dinámicamente o bien recurriendo a otros medios para almacenar y recuperar los objetos.

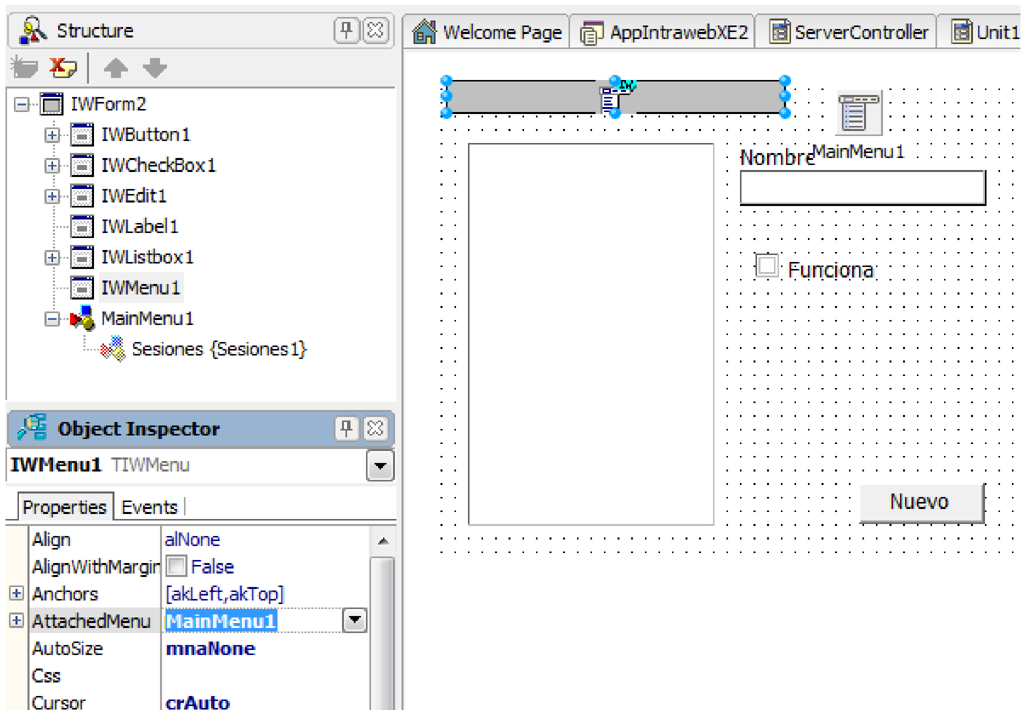
```
constructor TIWUserSession.Create;
begin
  inherited Create(nil);
  ultimo := 0;
end;

function TIWUserSession.getOrdenador(i: integer):
TOrdenador;
begin
  result := ordenadores[i];
end;
```

```
procedure TIWUserSession.addOrdenador(o: TOrdenador);
begin
  ordenadores[ultimo] := o;
  inc(ultimo);
end;
```

Esto es todo lo que necesitamos para guardar los datos de cada ordenador añadido y recuperarlos cuando se necesiten, todo ello de manera independiente para cada usuario (sesión).

La página principal

Vamos a ocuparnos ahora del módulo `Unit1`, en el que se encuentra la clase derivada de `TIWForm` que actuará como página principal de la aplicación. Usando los elementos habituales de diseño introduciremos en el formulario los elementos que pueden apreciarse en la imagen inferior, especialmente en la ventana Structure.



Los controles `TIWEdit` y `TIWCheckBox` servirán para que el usuario introduzca el nombre y estado de funcionamiento de cada ordenador, datos que se convertirán en un objeto `TOrdenador` asociado a la sesión en cuanto se haga clic sobre el `TIWButton`. En ese momento el nombre del ordenador también será agregado al `TIWListBox`. El código asociado al evento `OnClick` del botón es el siguiente:

```
procedure TIWForm2.IWButton1Click(Sender: TObject);
begin
  IWListbox1.Items.Add(IWEdit1.Text);
  IWListbox1.ItemIndex := IWListbox1.Items.Count-1;

  UserSession.addOrdenador(
    TOrdenador.Create(IWEdit1.Text, IWCheckBox1.Checked));

  if IWListbox1.Items.Count = 1 then
      WebApplication.ShowMessage(
        'Tu clase de navegador es ' +
        WebApplication.Browser.ClassName);
end;
```

Tras agregar el nombre del nuevo ordenador a la lista, y activar ese elemento como activo, creamos el objeto `TOrdenador` y lo añadimos a la información de sesión. La última sentencia es únicamente para demostrar cómo una aplicación web puede mostrar un mensaje de aviso con el método `ShowMessage` de la clase `TIWApplication`, en este caso indicando al usuario el navegador que está utilizando. El mensaje aparecerá una sola vez: al agregar el primer ordenador.

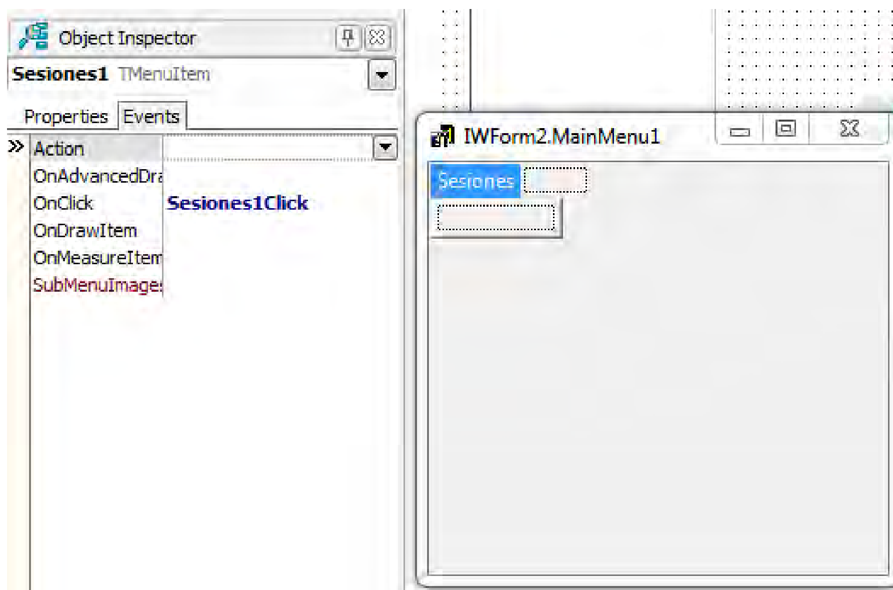
La lista no solamente mostrará el nombre de los ordenadores añadidos hasta el momento, sino que también servirá para seleccionar uno de ellos y ver la información asociada. Para ello responderemos al evento `OnChange` de la manera siguiente:

```
procedure TIWForm2.IWListbox1Change(Sender: TObject);
begin
  with UserSession.getOrdenador(IWListbox1.ItemIndex) do
  begin
    IWEdit1.Text := nombre;
    IWCheckBox1.Checked := funciona;
  end;
end;
```

Recuperamos el objeto `TOrdenador` que corresponde al elemento seleccionado en la lista, leyendo de sus atributos el nombre y el estado de funcionamiento para mostrarlos en el `TIWEdit` y `TIWCheckBox`, respectivamente.

Queremos que la página principal de la aplicación cuente con un menú de opciones, si bien contendrá una única opción nos servirá para aprender a usar este tipo de elemento una aplicación web. Aunque entre los componentes IntraWeb hay uno llamado TIWMenu éste no dispone de propiedades que permitan definir las opciones ni eventos para responder cada vez que sean usadas.

Para diseñar un menú tendremos que recurrir al habitual componente TMainMenu de la VCL, usando el editor asociado a fin de crear los TMenuItem que se necesiten. En la imagen inferior se muestra la estructura de nuestro menú, conteniendo una sola opción con el título Sesiones. El método asociado al evento OnClick de ésta ejecutará el método mostrado a continuación.




```
procedure TIWForm2.Sesiones1Click(Sender: TObject);
begin
  IWForm3.Show;
end;
```

La variable IWForm3 hace referencia a un segundo formulario IntraWeb que tendremos que agregar al proyecto, añadiendo al actual una referencia al módulo correspondiente en la cláusula uses a fin de poder mostrarlo. Los métodos Show y Hide, que conocemos de VCL y FMX, tienen la misma finalidad en una aplicación IntraWeb.

La conexión entre el TMainMenu y el TIWMenu la estableceremos a través de la propiedad AttachedMenu de este último. Como muchas otras, ésta ofrece una lista desplegable en la que aparecen los menús disponibles en el formulario, en este caso solamente uno.

La lista de conexiones

A diferencia de la lista de ordenadores, que será almacenada como información asociada a la sesión y, por tanto, cada usuario tendrá la suya, la lista de conexiones será compartida entre todos los usuarios. Dicha lista aparecerá en un formulario que hemos de agregar al proyecto, con la definición de una nueva clase derivada de TIWAppForm como puede comprobarse a continuación:

```
type
  TIWForm3 = class(TIWAppForm)
    IWListbox1: TIWListbox;
    IWButton1: TIWButton;
    procedure IWButton1Click(Sender: TObject);
  end;

...

var
  IWForm3: TIWForm3;
```

El formulario contendrá una lista para mostrar las conexiones y un botón cuya finalidad será cerrar la página, volviendo al formulario principal de la aplicación. La disposición no es importante. El método asociado al evento OnClick se limitará a llamar al método Hide del propio formulario.

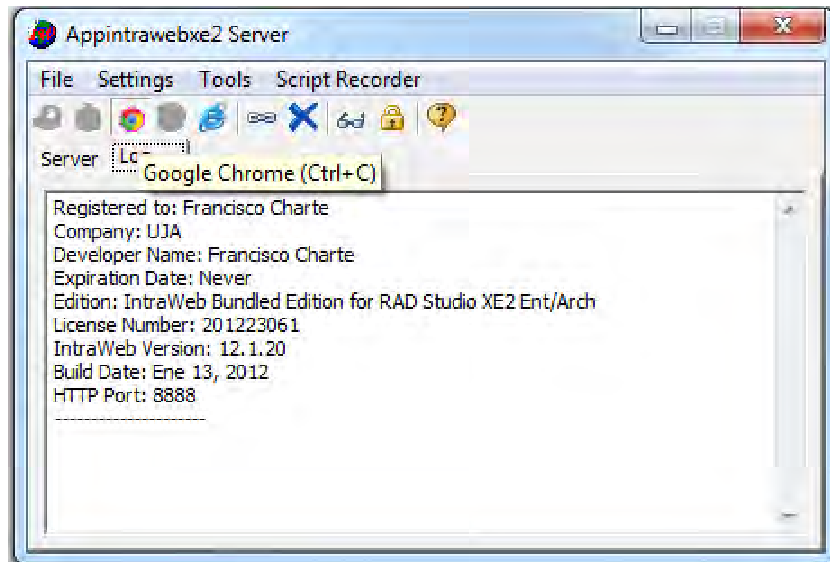
La variable IWForm3 está declarada en el mismo módulo donde se define la clase, pero queremos que solamente se cree un objeto de esta clase para toda la aplicación. Es una operación que debe delegarse en el controlador del servidor. También es éste el que cuenta con el evento OnNewSession que nos permite saber cuándo tiene lugar una nueva conexión, insertando en la lista la clase de navegador y dirección IP del cliente. El código a agregar al método asociado a dicho evento será el siguiente:

```
  if IWForm3 = nil then
      IWForm3 := TIWForm3.Create(gSC);

  IWForm3.IWListbox1.Items.Add('Conexión con ' +
    ASession.Browser.ClassName + ' desde ' + ASession.IP);
```

Usamos el parámetro `ASession` que se recibe como parámetro, y que representa la nueva sesión, para obtener el contenido de las propiedades `Browser` e `IP`. La primera es una referencia a un objeto de una clase derivada de `TBrowser`, mientras que la segunda es una cadena de caracteres con la dirección IP.

Ejecución del proyecto

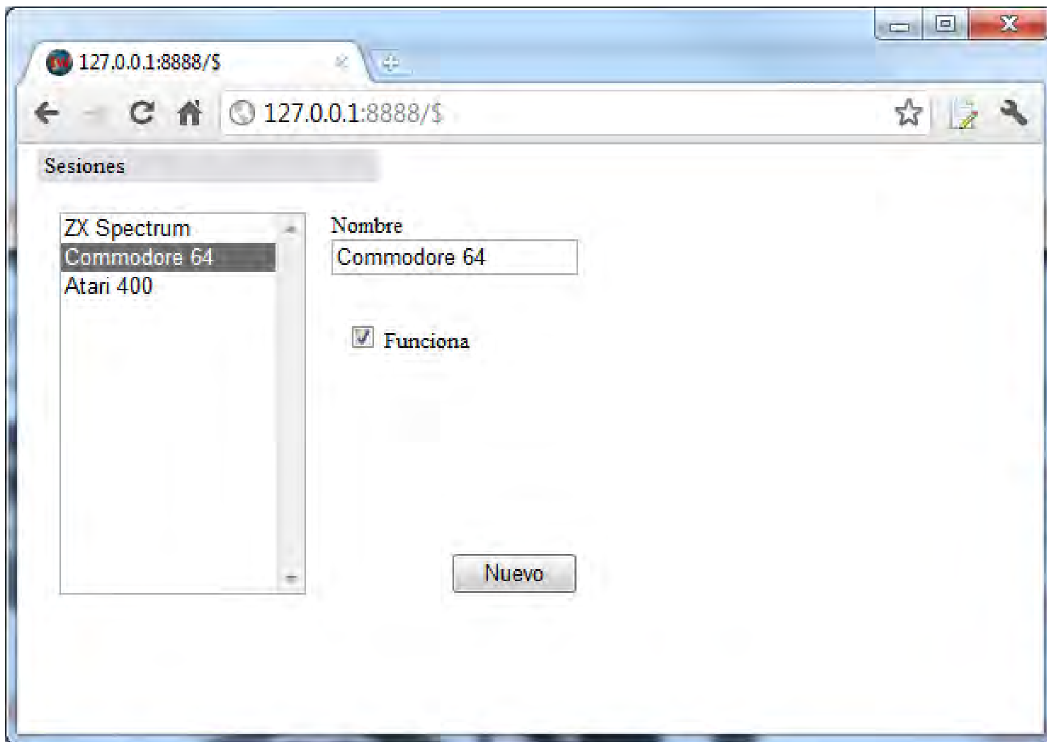
Realizados todos los cambios descritos en los puntos previos estamos en disposición de ejecutar el proyecto IntraWeb. Al hacerlo veremos aparecer la ventana del servidor (véase la imagen inferior), con información sobre la versión de IntraWeb en uso y una página con parámetros de control de la aplicación.



Bajo el menú de opciones podemos ver una barra de botones de los cuales los cinco primeros (comenzando por la izquierda) corresponden a otros tantos navegadores. Únicamente estarán activos los de los navegadores que, en efecto, se encuentren instalados en el sistema. Al hacer clic sobre cualquiera de ellos se abrirá una nueva página en la que aparecerá el formulario principal de la aplicación. También podemos usar el botón inmediatamente posterior para pegar en el portapapeles el enlace a la

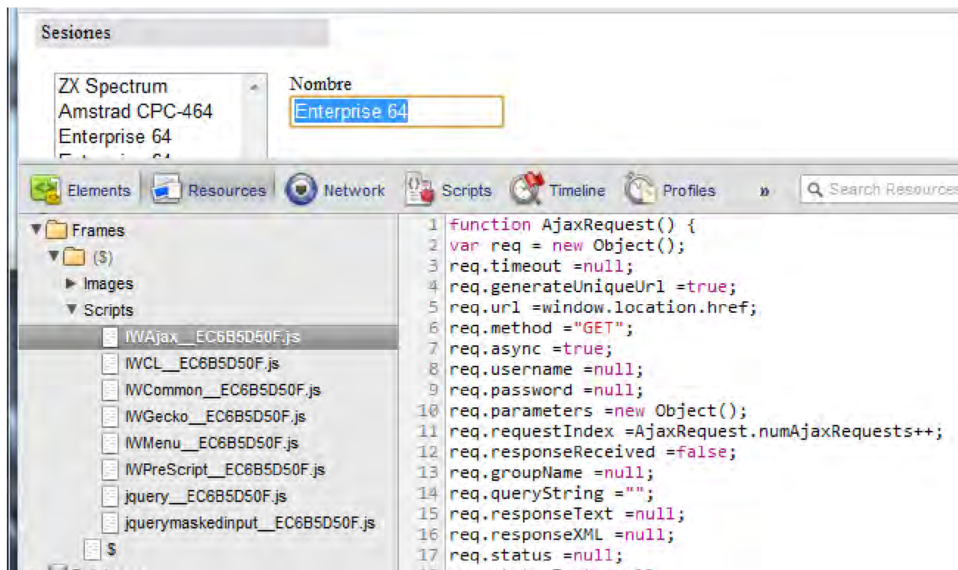
aplicación, lo cual nos permite hacer pruebas con otros navegadores que pudieran no haber sido reconocidos.

Una vez se haya lanzado la ejecución en el navegador podremos comenzar a introducir datos, viendo cómo los nombres de los ordenadores van apareciendo en la lista. Al cambiar el elemento activo en ésta se actualizará la información mostrada en los otros dos controles, tal y como puede apreciarse en la imagen inferior.



La página HTML alojada en el navegador contiene los elementos visibles en la lista, pero los datos de cada objeto `TOrdenador`, que permiten volver a mostrar el nombre y estado de funcionamiento en el formulario cuando se cambia el elemento seleccionado, están almacenados en el servidor web como un paquete de información asociado a la sesión de cada usuario. Lo que ocurre es que cada clic en la `TIWListBox` se traduce en una solicitud enviada al servidor y, a la postre, la ejecución del método asociado al evento `OnChange`. Los cambios realizados allí se traducen en una nueva respuesta enviada al servidor.

Si examinamos el código de la página, ya sea con la opción Ver código fuente que ofrecen la mayoría de los navegadores o con herramientas específicas para desarrolladores como las que ofrece Chrome y que aparece en la imagen siguiente, comprobaremos que en el formulario HTML hay definidos múltiples campos ocultos que se usan para mantener el estado de la página. Asimismo se hace referencia a varios módulos JavaScript (véase la imagen inferior) que son los que almacenan el código que se ejecuta en el cliente, por ejemplo preparando las solicitudes AJAX[93] y procesando las respuestas recibidas a fin de actualizar la interfaz web.

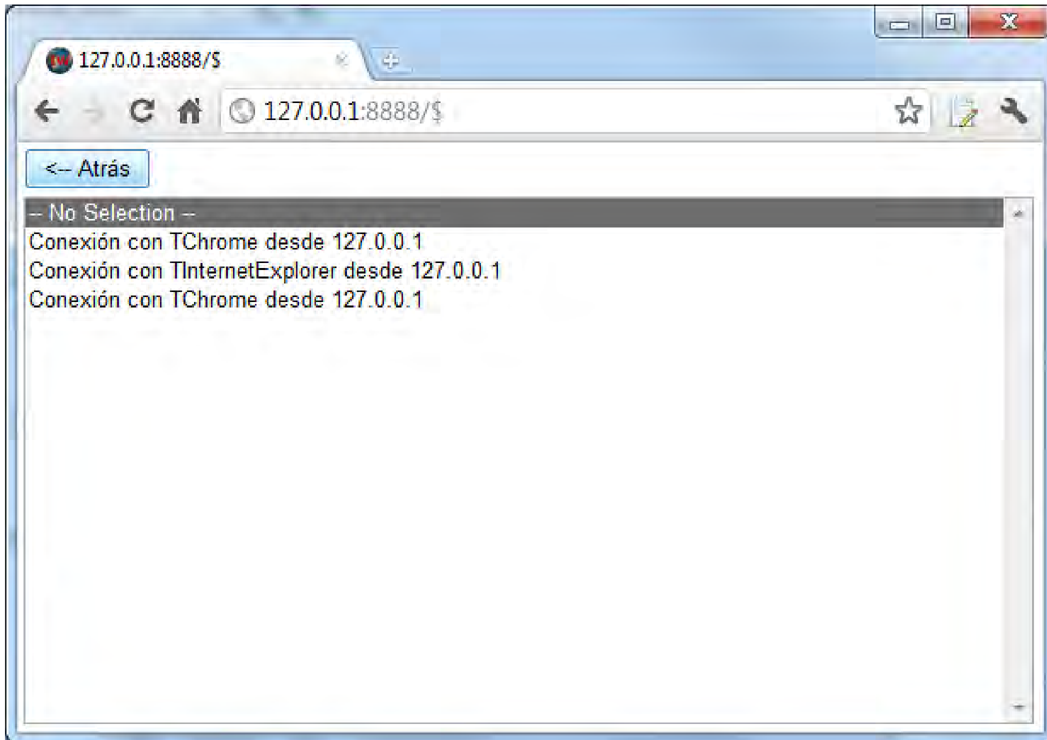


El menú de opciones, al contener una única opción, aparece en la página como un enlace en la parte superior izquierda que desencadena la opción asociada. Si hubiésemos definido varias opciones con subopciones veríamos cómo éstas se despliegan como ocurre habitualmente en un menú cualquiera.

Suponiendo que hubiésemos abierto varias ventanas de acceso a la aplicación, para simular la presencia de múltiples usuarios, en la lista de conexiones (véase imagen de la página siguiente) obtendremos el nombre

93 *Asynchronous Javascript And XML*. Es una conocida técnica que se emplea desde hace años en el desarrollo de aplicaciones web a fin de actualizar partes de la interfaz de usuario sin necesidad de solicitar al servidor la página completa a cada paso.

del navegador y la dirección IP de cada uno de ellos. Un clic en el botón que habíamos colocado en esta página, y que invocaba al método Hi de del propio formulario, nos devolverá a la página principal de la aplicación.



Asociar formularios y URLs

En principio la única vía con que contamos para acceder al formulario que muestra las conexiones activas, o cualquier otro que pudiera existir en el proyecto, es a través de la página principal. Ésta ha de invocar al método Show del formulario que deseemos abrir, a menos que asociemos un URL que haga posible el acceso directo.

Las asociaciones entre formularios y contenidos se gestionan mediante la clase TIWURLMap, definida en el módulo IWURLMap que será necesario añadir a la cláusula uses del formulario afectado por la asociación. Dicha

clase cuenta con un método llamado Add al que es necesario facilitar tres parámetros:

- La ruta del URL relativo[94] que se asociará al formulario.
- El archivo del URL a asociar.
- El tipo del formulario asociado.

Podríamos, por ejemplo, asociar el formulario que muestra la lista de conexiones con el URL `http://myapp/conexiones/index.html` agregando la siguiente sentencia a la sección de inicialización del propio formulario:

```
initialization

  TIWURLMap.Add('/conexiones/', 'index.html', TIWForm3);
```

> NOTA
>
> En realidad la asociación funcionará con cualquier URL en el que coincida la ruta relativa indicada, sin que importe el nombre del archivo. Podría usarse tanto `http://myApp/conexiones` como esta raíz seguida de cualquier nombre de archivo.

Identificación de usuarios

En su configuración actual cualquiera que conozca la dirección IP y nombre de la aplicación, el URL completo, podría acceder a ella sin ninguna limitación. Habrá casos en que interese que esto sea así, para servicios puestos a disposición pública, y otros en los que se necesite establecer un control sobre los usuarios.

La página IW Authentication de la Paleta de herramientas contiene tres componentes IntraWeb relacionados con la identificación de usuarios y la autorización de acceso a los contenidos:

- `TIWAutherINI`: La identificación se realizará comparando las credenciales facilitadas por el usuario con las registradas en un archivo INI. Éste habrá de encontrarse en la misma carpeta que la

94 La base siempre será el URL de la página de inicio o principal de la aplicación.

aplicación y su nombre debe ser #Auth.ini. En él existirá una sección por usuario, usando el nombre de éste como identificador. En el interior de cada sección una única pareja clave-valor con la sintaxis Password=contraseña.

- TIWAutherList: El funcionamiento es similar al del componente anterior, pero con la diferencia de que la lista de usuarios no estará almacenada en un archivo INI sino que se introducirá en la propiedad List. Esto permite eliminar dependencias de archivos externos pero, al tiempo, es un mecanismo menos flexible a la hora de facilitar el registro o eliminación de usuarios.
- TIWAutherEvent: A diferencia de los dos anteriores la comprobación de las credenciales por parte de este componente no se realiza automáticamente, sino que se delega en el programador. Para ello se genera un evento OnCheck que entrega dos parámetros de tipo String: el nombre de usuario y la contraseña facilitado. El método asociado es una función que ha de devolver True si la identificación es satisfactoria o False en caso contrario. Esto permite realizar la autenticación de los usuarios, por ejemplo, contra información almacenada en una base de datos o algún otro recurso.

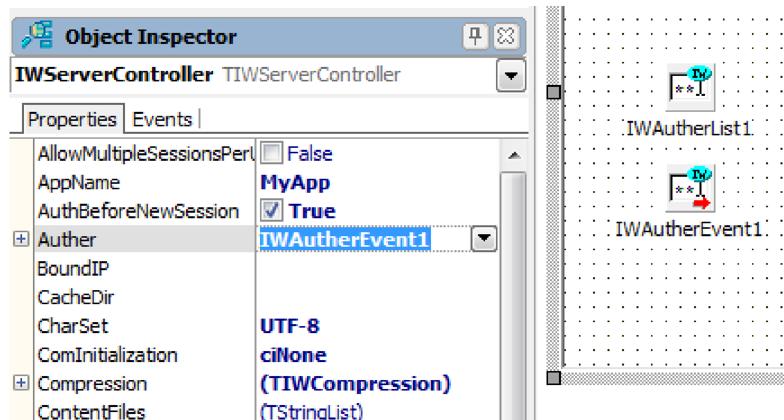
Los tres componentes cuentan con una propiedad llamada AutherPolicy que establece la política de acceso a la aplicación y que puede tomar dos valores: apRestrictAll y apRestrictNone, siendo el primero el tomado por defecto. En modo apRestrictAll se restringe el acceso a cualquier parte de la aplicación para usuarios que no estén identificados.

NOTA

Es posible establecer una política de acceso global, por ejemplo de acceso restringido, pero manteniendo algunos elementos de la aplicación accesibles para usuarios no identificados. Para ello es necesario redefinir el método RequestAuth heredado de TIWForm, devolviendo True o False según se autorice el acceso o no. Dicho método recibe como parámetro la información sobre la solicitud en un objeto THttpRequest.

Además los tres componentes generan un evento OnAuthenticate una vez completado el proceso de identificación, indicando con un parámetro de tipo Boolean si ésta ha sido satisfactoria o no.

Una vez que hayamos decidido qué método usar para la identificación, configurando uno de los tres componentes anteriores[95], habrá que establecer una conexión con el controlador del servidor. Para ello recurriremos a la propiedad `Auther` de éste tal y como se muestra en la imagen inferior. También habrá que activar la propiedad `AuthBeforeNewSession`, de forma que cada vez que un usuario vaya a iniciar una sesión se le soliciten las credenciales de acceso.



ADVERTENCIA

La versión de los componentes IntraWeb que incluye Delphi XE2 tiene desactivada la funcionalidad que acaba de describirse y que permitiría identificar a los usuarios de la aplicación. Para estas características es necesario actualizar a una edición superior, contacta con Danysoft. En http://www.atozed.com/IntraWeb/FeatureMatrix.EN.aspx se indica cuál es la edición incluida con Delphi XE2 y sus limitaciones, así como las características de las edición Ultimate. También es posible probar esta funcionalidad por tiempo limitado, eliminando la información de licencia de IntraWeb para activar el modo de evaluación.

---

95 El componente que corresponda habrá que insertarlo en el módulo del controlador de servidor.

Conclusión

Delphi XE2 incorpora varias bibliotecas que permiten desarrollar aplicaciones web, todas ellas heredadas de versiones previas del producto, entre las cuales destaca IntraWeb por su metodología de trabajo, prácticamente idéntica a la que emplearíamos para una aplicación VCL o FMX, como por el hecho de ser la única alternativa que sigue evolucionando.

En este capítulo hemos conocido la estructura general de una aplicación IntraWeb y algunos de sus componentes. También hemos desarrollado un proyecto sencillo con algunas características habituales de una aplicación web, como es el mantenimiento de datos asociados a cada sesión y el acceso a información compartida.

A partir de aquí lo único que necesitaríamos sería un conocimiento detallado de cada uno de los componentes disponibles, especialmente los relacionados con el diseño de la interfaz. Es una información que encontramos en la documentación del fabricante.

A continuación

En el capítulo siguiente, segundo de esta parte del libro, complementaremos lo aprendido en éste con el objetivo de poder desarrollar interfaces web que faciliten el acceso a servidores DataSnap. Esto dará lugar a aplicaciones distribuidas con una capa adicional, ya que el servidor web puede estar ejecutándose en una máquina distinta al servidor de aplicaciones.

Como comprobaremos, no hay más que combinar lo aprendido en este capítulo sobre IntraWeb con lo que conocimos en los capítulos dedicados a estudiar DataSnap y dbExpress.

Capítulo 9: Interfaces web para aplicaciones DataSnap

Unas de las razones que ha impulsado desde hace unos años el crecimiento de la arquitectura distribuida en el diseño de aplicaciones, frente a la clásica cliente/servidor, ha sido la gran difusión de la Web. Partiendo de que ya sabemos cómo conectar con un RDBMS, implementar servicios en un servidor de aplicaciones con DataSnap y desarrollar clientes en forma de aplicaciones ejecutables, en este capítulo conectaremos el último eslabón.

Servidores web y de aplicaciones

Al planificar el desarrollo de una aplicación distribuida que ha de contar con una interfaz de usuario web, a fin de que los clientes puedan utilizarla desde un navegador sin necesidad de instalar software adicional, Delphi XE2 nos ofrece fundamentalmente dos vías:

- Integrar el servidor DataSnap y la generación de la interfaz web en un mismo ejecutable, de forma que servidor de aplicaciones y servidor web operen en una misma máquina de forma solidaria como un único componente.
- Separar ambos elementos en dos ejecutables distintos: un servidor de aplicaciones DataSnap similar al desarrollado en el séptimo capítulo, por una parte, y el módulo que generará la interfaz ejecutándose en el servidor web, por ejemplo con IntraWeb como se explicó en el capítulo previo.

Un servidor DataSnap, como se indicó en el capítulo dedicado a esta tecnología, contempla el establecimiento de conexiones por HTTP y HTTPS, lo que unido a REST[96] y la generación automática de clases *proxy* en distintos lenguajes hace posible que actúe también como servidor web. La interfaz de la aplicación habría que diseñarla directamente en forma de páginas HTML, hojas de estilo CSS y código JavaScript embebido para invocar a los servicios ofrecidos por el servidor de aplicaciones.

La alternativa: separar como proyectos independientes el servidor DataSnap de la aplicación web, agrega una capa más a la arquitectura de la solución y permite un desarrollo y mantenimiento independiente de ambas partes. Asimismo ofrece una mayor libertad en cuanto a la técnica empleada para genera la interfaz de usuario que, por ejemplo, podría estar basada en los componentes IntraWeb que conocimos en el capítulo anterior.

---

96 *Representational State Transfer*. Es una arquitectura de tipo cliente/servidor asociada al protocolo HTTP, nació junto a la versión 1.1 de éste, y la propia evolución de la web. Cada solicitud es independiente del resto: no se mantiene información de estado, aceptándose operaciones representadas por los cuatro métodos de HTTP: `GET`, `PUT`, `POST` y `DELETE`, interpretados como operaciones de recuperación de datos, actualización de datos, inserción y eliminación, respectivamente. REST es usado actualmente en el desarrollo de servicios web en detrimento de otras técnicas como SOAP. Puedes encontrar una buena introducción a REST en http://www.infoq.com/articles/rest-introduction.

Interfaz IntraWeb y servidor DataSnap

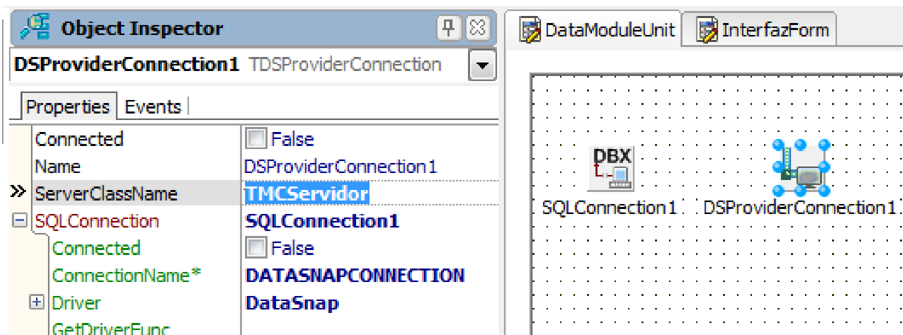
Partiendo del servidor DataSnap que habíamos desarrollado en un capítulo previo, y que facilitaba tanto el acceso a la información de la base de datos de ordenadores como los servicios adicionales para trabajar sobre ella, veamos cuáles serían los pasos a seguir para diseñar una interfaz de usuario web con IntraWeb.

Lo primero será poner en marcha el servidor, ejecutándolo y dejándolo en segundo plano durante el desarrollo de la nueva interfaz. De esta manera nos será más fácil configurar varios de los componentes. A continuación usaremos el asistente que conocimos en el capítulo anterior para generar una aplicación IntraWeb, aceptando la configuración que se ofrece por defecto salvo la opción Pool Data Connections que hemos de activar.

Conexión con el servidor DataSnap

Nuestro nuevo proyecto cuenta con un módulo llamado `DatamoduleUnit1` en el que se define la clase `TDataModule1`. Ésta actúa como un módulo de datos y lo interesante es que el servidor web puede mantener un *pool* de objetos de este tipo a fin de no saturar el servidor de aplicaciones con una cantidad enorme de conexiones, por una parte, y ofrecer al mismo tiempo a la interfaz conexiones ya preparadas.

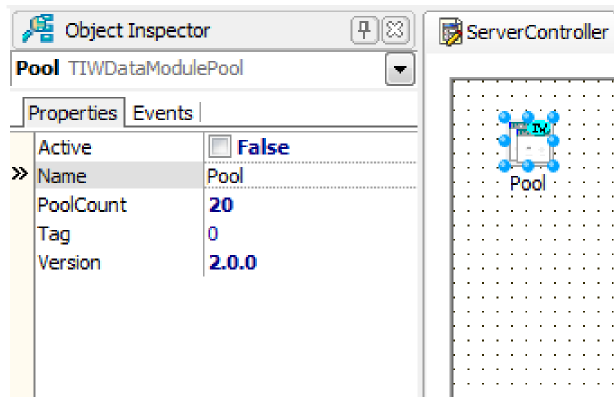
Introduciremos en este módulo los dos componentes que pueden verse en la imagen inferior: un `TSQLConnection` y un `TDSProviderConnection`.



Configuraremos el `TSQLConnection` para conectar con el servidor DataSnap, que está ejecutándose, y enlazaremos el `TDSProviderConnection` con el anterior. A continuación asignaremos el valor `TMCServidor` a la propiedad `ServerClassName` del `TDSProviderConnection` y, para comprobar que todo funciona bien, activaremos la propiedad `Connected`. Si ésta toma el valor `True` es que ha sido podido establecer la conexión con el servidor sin problemas.

NOTA

A diferencia de los componentes introducidos en el módulo `UserSessionUnit` o el propio formulario, cuyo estado se almacenada en la sesión de cada usuario y por tanto implicaría la creación de una conexión para cada cliente activo, el contenido del módulo de datos está asociado al servidor y, en consecuencia, se creará un número limitado de objetos `TDataModule1` a compartir entre todos los clientes. El número concreto dependerá de la propiedad `PoolCount` del objeto `TIWDataModulePool` existente en el módulo `ServerController`.



Si introdujésemos la conexión con el servidor en el módulo de sesión de usuario o el formulario, prescindiendo del módulo de datos, el servidor DataSnap tendría que hacer frente a tantos usuarios simultáneos como sesiones activas existiesen y dicho número, cuando se trata de aplicaciones accesibles a través de la web, puede ser muy grande.

Acceso a los datos desde la interfaz

Los componentes que hemos colocado en el módulo de datos, por sí solos, no nos facilitarán el acceso a los datos propiamente dichos. Necesitamos un TClientDataSet para almacenar localmente la información sobre la que vamos a trabajar. Este componente, y el TDataSource asociado, podemos insertarlo tanto en el formulario como en el módulo UserSession, de forma que cada usuario cuente con su propia copia del TClientDataSet. Esto es necesario ya que éste almacena no solamente los datos, sino también los cambios que cada cliente pueda efectuar sobre ellos.

La conexión entre el TClientDataSet y el TDSProviderConnection que añadimos antes al módulo de datos no se establecerá en la fase de diseño[97]. Hay que recordar que el servidor mantendrá un *pool* de objetos TDataModule1, siendo necesario obtener uno de ellos durante la ejecución para conectar con el servidor DataSnap y solicitar datos o enviar cambios.

El conjunto de filas inicial se obtendrá en el momento en que se abra el formulario: evento OnCreate. En ese momento debemos obtener una instancia del módulo de datos del *pool* que mantiene el componente TIWDataModulePool, para lo cual aprovecharemos la función LockDataModule[98] que el asistente ha introducido en el controlador del servidor. Ésta nos devuelve el objeto TDataModule1 que usaremos para recuperar una referencia al TDSProviderConnection, asignándola a la propiedad RemoteServer el TClientDataSet. La propiedad ProviderName, de tipo String, debe contener el nombre del proveedor de datos remoto. Establecida la configuración podemos abrir el TClientDataSet, lo que provocará la transferencia de los datos hasta el cliente. Completado el proceso eliminaremos el enlace entre el TClientDataSet y el TDSProviderConnection, finalizando con la liberación del objeto TDataModule1 que obtuvimos al inicio entregándolo como parámetro al procedimiento UnlockDataModule.

---

97 Podemos configurar la conexión durante el diseño para facilitar el enlace de los componentes de datos, pero al terminar deberíamos eliminar el contenido de las propiedades RemoteServer y ProviderName del TClientDataSet.

98 El componente TIWDataModulePool ofrece un método Lock que devuelve uno de los objetos TDataModule disponibles en el *pool*. La función LockDataModule definida en TIWServerController sencillamente llama a ese método y realiza la necesaria conversión de TDataModule a TDataModule1, permitiéndonos trabajar con los elementos específicos de nuestro módulo de datos.

El código a introducir en el método asociado al evento OnCreate será el mostrado a continuación:

```
procedure TInterfaz.IWAppFormCreate(Sender: TObject);
var
  dm: TDataModule1;
begin
  dm := LockDataModule;
  with ClientDataSet1 do begin
    RemoteServer := dm.DSProviderConnection1;
    ProviderName := 'DSPClassic';
    Open;
    RemoteServer := nil;
  end;
  UnlockDataModule(dm);
end;
```

Si la interfaz web va a permitir a los usuarios modificar, agregar o eliminar datos será necesario transmitir los cambios al servidor de aplicaciones que, a su vez, lo hará llegar al RDBMS. Sabemos que para ello hemos de llamar al método ApplyUpdates, gestionar los conflictos que pudieran surgir y después, opcionalmente, invocar a Refresh para actualizar la información contenida en el TClientDataSet.

Antes de llamar a ApplyUpdates tendremos que reconectar el TClientDataSet con el TDSProviderConnection, mediante el mismo procedimiento anterior: obtener una instancia del módulo de datos, dar valor a las propiedades RemoteServer y ProviderName y finalmente enviar los cambios. Terminada la operación volveríamos a liberar el objeto TDataModule1 obtenido al principio.

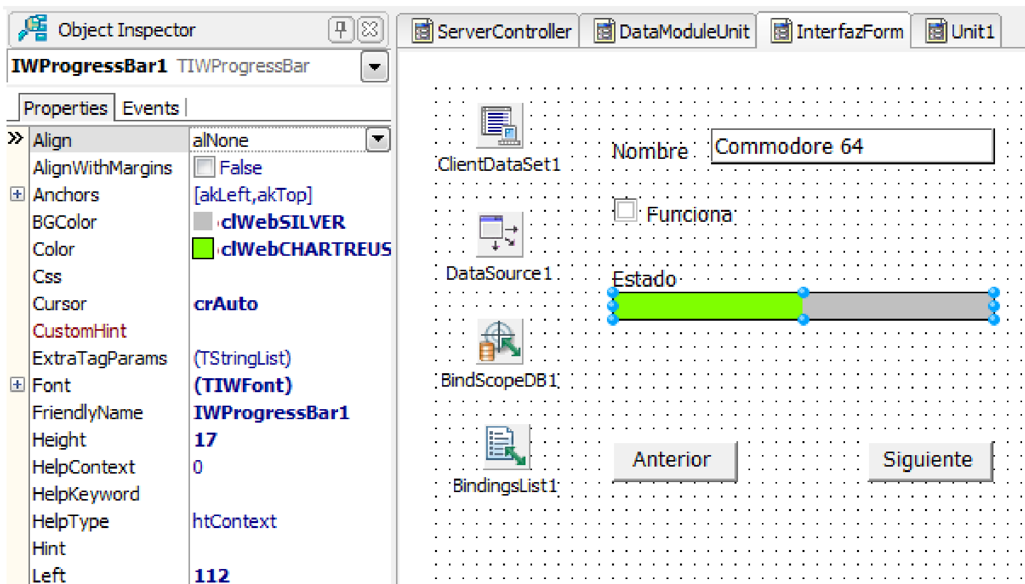
Diseño de la interfaz – LiveBindings

Una vez que tenemos resuelta la configuración de acceso a los datos vamos a concentrarnos en el diseño de la interfaz de usuario. Tenemos dos alternativas: usar los componentes de IntraWeb alojados en la página IW Standard y los LiveBindings que conocimos en el quinto capítulo, que será lo que hagamos en este punto, o bien recurrir a los controles de la página IW Data que son similares a los componentes de datos de la VCL. Nos ocuparemos de esa opción en el punto siguiente.

Lo primero que haremos será agregar al formulario un componente TBindScopeDB, que actuará como intermediario entre los componentes de

datos y los controles visuales, enlazándolo con el TDataSource que ya teníamos mediante la propiedad DataSource del primero. El TBindScopeDB actuará como componente de origen en los LiveBinding que vamos a definir de inmediato.

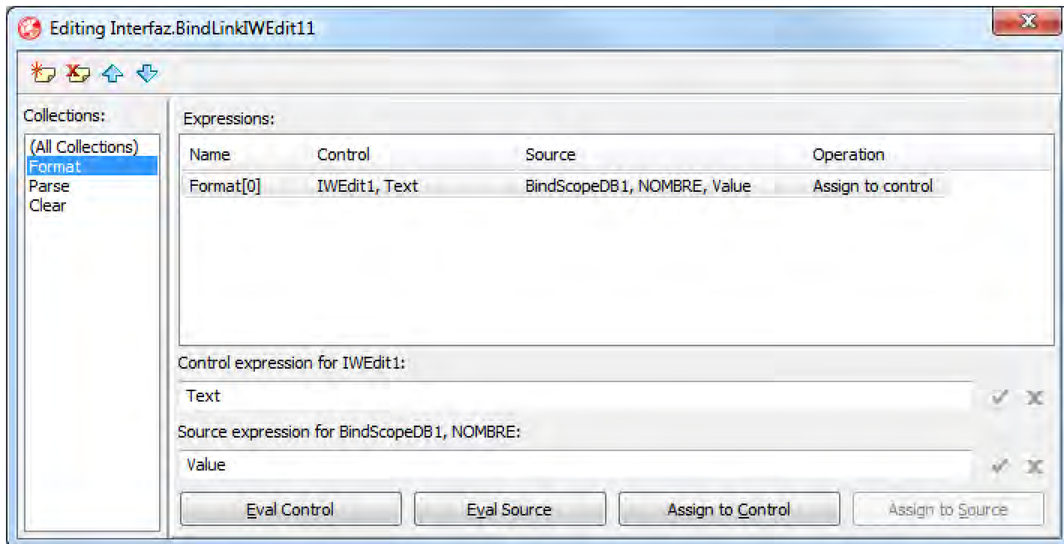
Nuestro objetivo es navegar por los ordenadores existentes en la base de datos accediendo a su nombre y estado de funcionamiento, para lo cual introduciremos en el formulario un componente TIWEdit, un TIWCheckBox y un TIWProgressBar, aparte de dos TIWButton que serán los encargados de avanzar al siguiente registro y volver al anterior. El aspecto de la interfaz durante el diseño será el mostrado en la imagen inferior.



Teniendo seleccionado el TIWEdit usamos la opción New LiveBinding (aparece tanto en el menú contextual como en la parte inferior del Inspector de objetos) para definir el enlace de datos. Las propiedades a establecer en el nuevo TBindLink serán:

- ControlComponent: Hará ya referencia al control TIWEdit.
- SourceComponent: Elegimos de la lista el objeto TBindScopeDB1.
- SourceMemberName: En la lista asociada aparecen las columnas de la tabla de libros, información obtenida del RDBMS a través del servidor DataSnap. Elegimos la columna NOMBRE.

A continuación usaremos la opción Expressions del LiveBinding para abrir la ventana de edición de expresiones (véase la imagen inferior). Usando el primero de los botones agregaremos una nueva expresión en la categoría Format, vinculando la propiedad Text del TIWEdit con la propiedad Value de la columna NOMBRE[99].

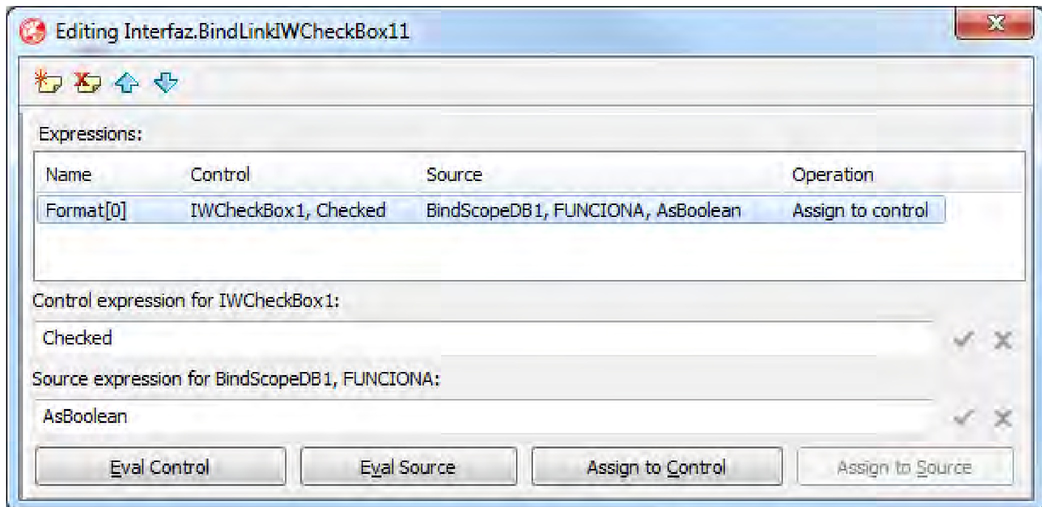


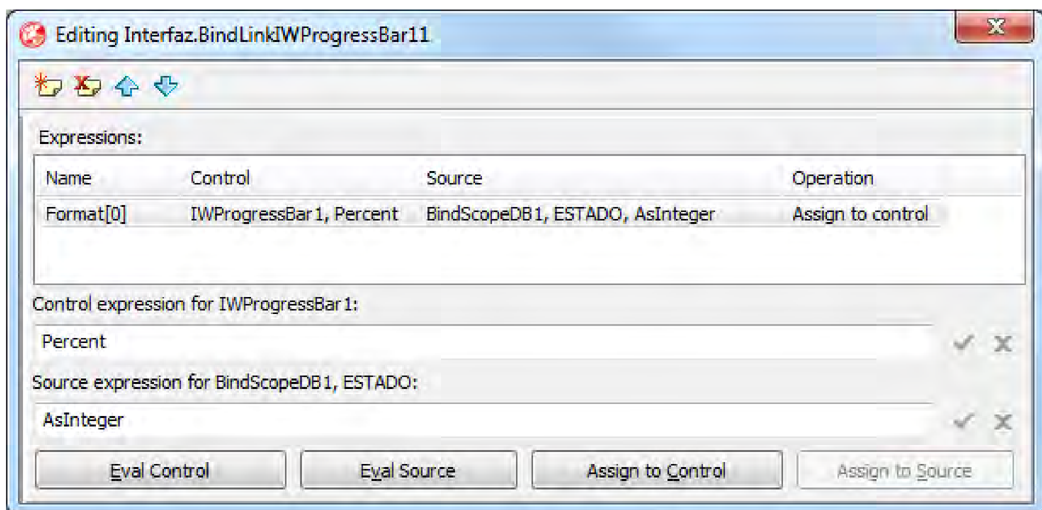
Repetiremos los pasos del procedimiento anterior para enlazar el control TIWCheckBox con la columna FUNCIONA de la tabla de ordenadores. Dicha columna contiene las cadenas de caracteres 'True' o 'False', valores que es necesario interpretar como booleanos. Por ello la expresión de origen será AsBoolean en lugar de Value. El resultado de la evaluación de esta expresión se asignará a la propiedad Checked del control, tal y como se muestra en la primera imagen de la página siguiente.

Por último crearemos el LiveBinding para enlazar el TIWProgressBar, concretamente su propiedad Percent que puede tomar un valor entre 0 y 100. La columna con la que se enlazará es ESTADO, cuyo contenido interpretaremos como un entero mediante el método AsInteger. En la segunda imagen de la página siguiente puede comprobarse la configuración.

---

99 Sabemos que esta columna almacena un texto, es de tipo String, y que la propiedad Text del TIWEdit es también de tipo String, por lo que no se precisa realizar una conversión de tipos.





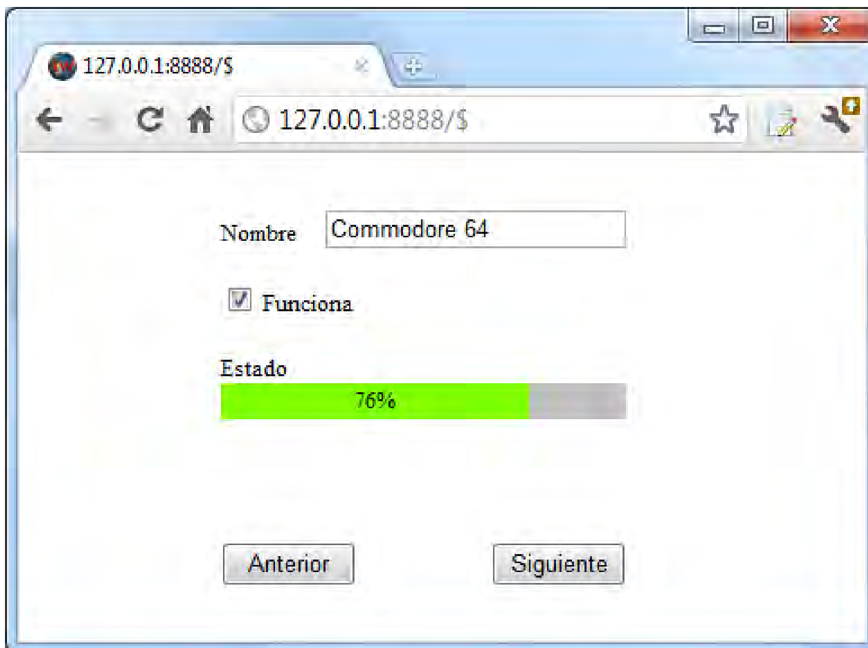
NOTA

Además de la expresión que facilita la evaluación desde el componente origen hasta el controlado, si quisiéramos facilitar la edición habría que crear también la de sentido inverso, en la categoría Parse.

Los dos botones que hemos agregado a la interfaz no estarán enlazados con la posición del TClientDataSet a través de un enlace, sino que se limitarán a invocar a dos métodos de dicho componente: Next y Prior. Como es fácil suponer el primero avanza a la siguiente fila, mientras que el segundo retrocede a la anterior. Los métodos asociados al evento OnClick de los botones quedarán como se muestra a continuación:

```
procedure TInterfaz.IWButton1Click(Sender: TObject);
begin
  ClientDataSet1.Next;
end;

procedure TInterfaz.IWButton2Click(Sender: TObject);
begin
  ClientDataSet1.Prior;
end;
```

De manera análoga podríamos agregar botones para eliminar e insertar filas, entrar en modo de edición de la fila actual o confirmar los cambios, lógicamente completando la interfaz para poder acceder al resto de los datos de cada ordenador. Con lo que hemos hecho, sin embargo, es suficiente para poder navegar por los datos desde una página web como puede verse en la imagen inferior.

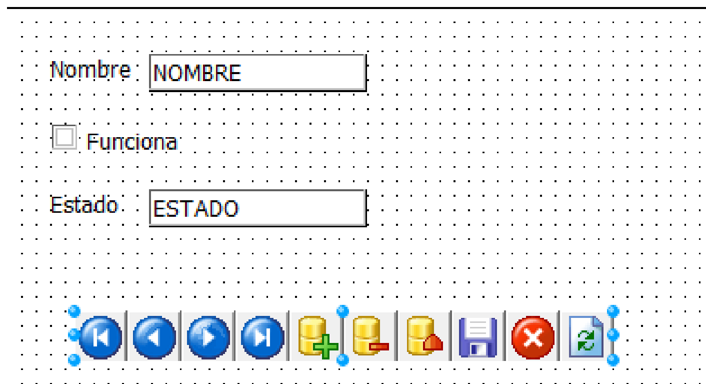


Diseño de la interfaz – Controles DB

Cuando se quiere crear una interfaz web con conexión a datos mediante IntraWeb suele ser más cómodo recurrir a los componentes de la página IW Data, similares a los existentes en la VCL y que se caracterizan por disponer de dos propiedades adicionales: `DataSource` y `DataField`. La primera conecta el control con el componente `TDataSource`, mientras que la segunda selecciona la columna concreta con la que se enlazará.

Por tanto no es necesario el componente `TBindScopeDB`, la creación de un LiveBinding para cada control y la definición de las expresiones asociadas. Además la navegación por los datos, así como la ejecución de las operaciones habituales de inserción, borrado y actualización, se ven facilitadas por el componente `TIWDBNavigator`.

Agregando a nuestro formulario dicho componente, junto con dos `TIWDBEdit` y un `TIWDBCheckBox`, todos ellos conectados con el `TDataSource` y las columnas correspondientes, tendremos una interfaz con una funcionalidad similar a la diseñada en el punto anterior como puede verse en la imagen inferior.



> ADVERTENCIA
>
> Si se mantienen visibles en el `TIWDBNavigator` botones como el de actualización, adición o borrado de datos hemos de agregar el código necesario para conectar el `TClientDataSet` con el proveedor remoto, realizar la operación y desconectar, como se indicó anteriormente.

Servidor DataSnap y REST

Acabamos de comprobar cómo usando componentes IntraWeb es posible desarrollar un cliente ligero, para usuarios que acceden a la aplicación mediante un navegador, de forma muy sencilla. El servidor DataSnap es exactamente el mismo que teníamos de un capítulo previo, no ha sido necesario introducir cambios. Para él no existen diferencias entre un cliente que conecta con la aplicación FMX desarrollada en el séptimo capítulo o la aplicación web creada en el punto anterior, él mismo no ofrece más que una canal de comunicación a través de TCP/IP por el que enviar las solicitudes y obtener las respuestas.

Existe la posibilidad, sin embargo, de crear un servidor DataSnap que actúe a su vez como servidor web, ofreciendo la interfaz final para el usuario en forma de páginas HTML en las que se combina contenido estático con otro generado dinámicamente, facilitando incluso el acceso directo a los servicios expuestos gracias a REST y JSON[100]. Un servidor así contaría con los mismos componentes básicos que conocimos en el capítulo dedicado a DataSnap: un `TDSServer` que controla el funcionamiento global del servidor, uno o más `TDSServerClass` que asocian con la aplicación las clases que ofrecen los servicios y un componente encargado de facilitar la comunicación que en este caso sería un `TDSHTTPWebDispatcher`.

Esos elementos, a los que se añadirán otros que vamos a conocer de inmediato, se incluirán en un módulo contenedor que no estará derivado de `TDataModule` como en un servidor DataSnap estándar sino de `TWebModule`. Éste puede alojar componentes no visuales de acceso a datos, relacionados con DataSnap y otros específicos para la gestión de solicitudes HTTP y generación de respuestas como son `TPageProducer`.

El proyecto constará, además de con el módulo derivado de `TWebModule` conteniendo los componentes citados, uno o más módulos derivados de `TDSServerModule` conteniendo servicios, una interfaz de administración del servidor que puede ser un formulario VCL o una aplicación de consola y múltiples módulos web: páginas HTML, hojas de estilo, JavaScript, etc.

---

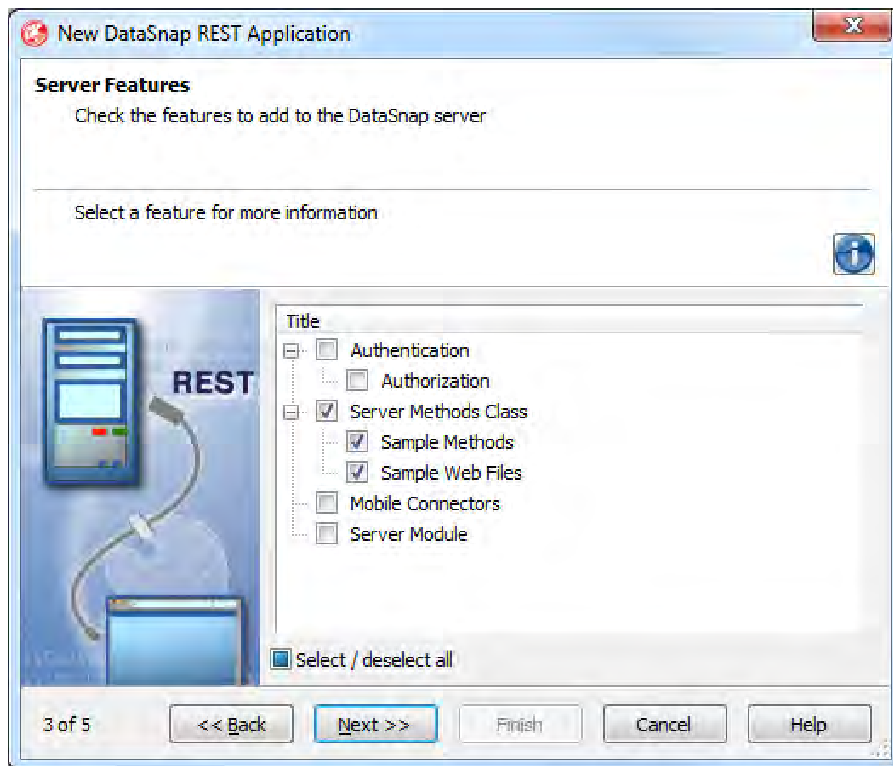
100 *JavaScript Object Notation*. Es un formato de intercambio de información alternativo a XML que se caracteriza por ser más compacto, aparte de fácilmente tratable en un navegador que cuente con un motor JavaScript que son la mayoría. Puedes encontrar una introducción a JSON en el sitio web oficial de este estándar: http://www.json.org/json-es.html.

El asistente DataSnap REST Application

Lanzaremos este asistente, que encontramos en el mismo grupo DataSnap Server que el que usábamos en el séptimo capítulo, para crear un nuevo servidor DataSnap de tipo REST. Los pasos del asistente son similares a los que conocimos entonces, comenzando por la elección del tipo de proyecto: aplicación VCL, de consola o módulo ISAPI para usar bajo IIS.

Sin darnos opción a elegir protocolos de comunicación, el único posible será HTTP y opcionalmente HTTPS, el asistente nos pedirá que indiquemos un puerto para el servidor siempre que el tipo de proyecto no sea ISAPI, ya que en ese caso será IIS el que se ocupe de gestionar las solicitudes. El puerto por defecto en los otros casos será el 8080, pero podemos cambiarlo por el 80 siempre que en la máquina no tengamos ya un servidor web ocupando el puerto estándar para este tipo de servicio.

El siguiente paso del asistente ofrece múltiples opciones (véase la imagen inferior) de las cuales solamente está marcada Server Methods Class.



Las opciones Authentication/Authorization se limitan a agregar los elementos necesarios para identificar a los usuarios y controlar el acceso a los servicios, tema que ya conocimos en el capítulo dedicado a DataSnap. La opción Sample Methods añade al proyecto un módulo con los dos métodos de ejemplo que también conocemos: `Echo` y `ReverseString`. Esa clase puede estar derivada de `TComponent`, `TDataModule` o `TDSServerModule`, opción que se ofrecerá en el siguiente paso del asistente. Dejando marcada la opción Sample Web Files también se agregarán al proyecto varios módulos web que mostrarán cómo acceder a los métodos del servidor desde un navegador.

Con la opción Mobile Connector, desactivada por defecto, el asistente introducirá los componentes que harían posible el acceso al servidor DataSnap desde dispositivos móviles. No es algo que nos interese en este momento.

Por defecto el asistente alojará todos los componentes del servidor DataSnap en el módulo derivado de `TWebModule`, según se ha indicado antes, salvo que activemos la opción Server Module. En ese caso el proyecto contará con un módulo adicional, una clase derivada de `TDataModule`, para que actúe como contenedor exclusivamente de los componentes DataSnap: `TDSServer`, `TDSServerClass` y `TDSAutheticationManager` si procede. Esto facilita fundamentalmente la separación del código asociado con el servidor de aplicaciones del que corresponde al servidor web.

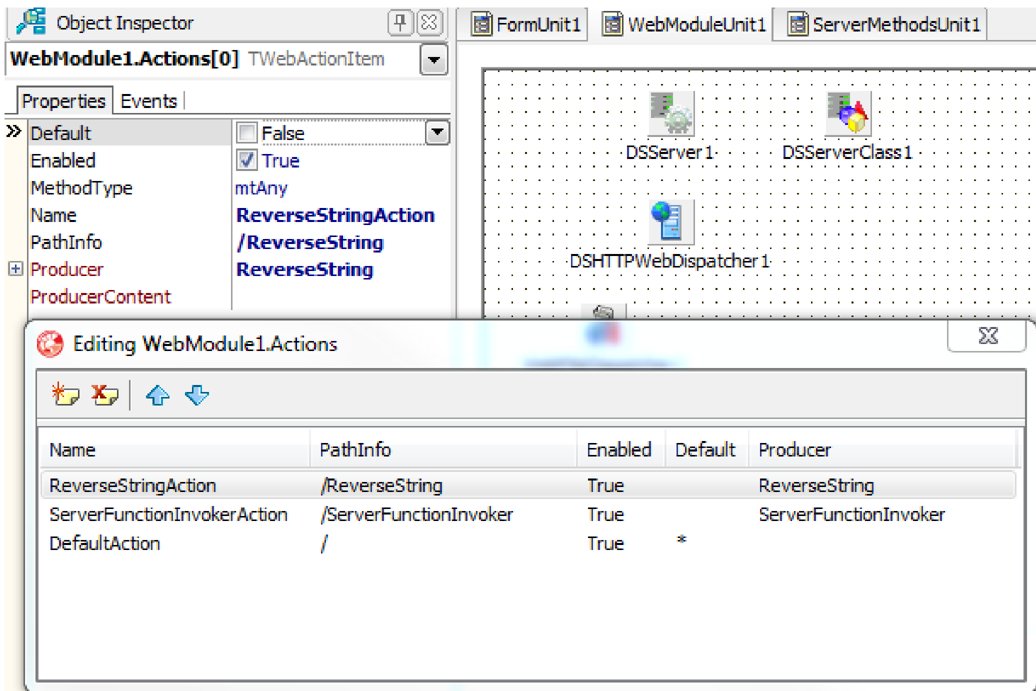
En nuestro caso aceptaremos la configuración que se ofrece por defecto, seleccionaremos en el siguiente paso del asistente la clase de la que se derivará el módulo con los servicios y, finalmente, indicaremos la carpeta en la que se crearán todos los módulos del nuevo proyecto.

Rutas de acceso a la aplicación

La aplicación resultante de este proyecto va a operar como un servidor web y, por tanto, responderá a solicitudes recibidas mediante el protocolo HTTP a través del puerto que se haya indicado. Esas solicitudes toman la forma de identificadores universales de recursos que, en cierta forma, son rutas que parten de la dirección base donde esté alojado el servidor.

Es tarea del módulo web de la aplicación, la clase derivada de `TWebModule`, asociar una acción a cada una de esas rutas. Si abrimos el módulo `WebModuleUnit1`, hacemos clic sobre el fondo del módulo y examinamos el

Inspector de objetos encontraremos la propiedad Actions. Ésta es una colección de objetos TWebActionItem que, como puede apreciarse en la imagen inferior, cuenta inicialmente con tres elementos.

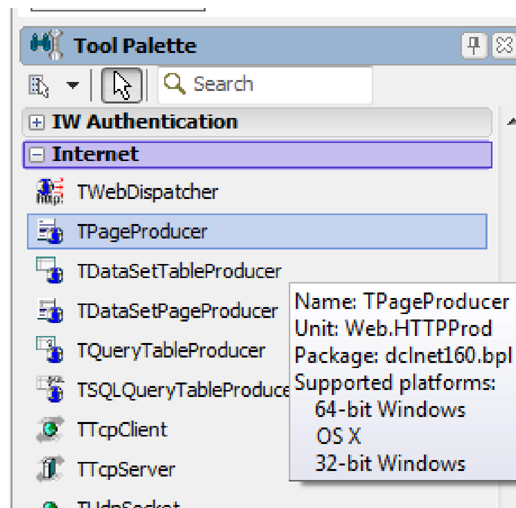


Al seleccionar una de las acciones (también podemos agregar otras) veremos en el Inspector de objetos sus propiedades. Las fundamentales son:

- Default: De tipo Boolean, solamente uno de los objetos puede actuar como acción por defecto. Normalmente será la que tenga asociada la ruta /.
- Enabled: Indica si la acción está activa, y generará una respuesta ante una solicitud, o por el contrario está inactiva.
- MethodType: Permite limitar el tipo de solicitudes ante las que responde una acción. Por defecto es mtAny pero hay cuatro valores más disponibles: mtGet, mtPut, mtPost y mtHead, correspondientes a los métodos GET, PUT, POST y HEAD del protocolo HTTP.
- PathInfo: Ruta asociada a la acción.
- Producer: Referencia al objeto que generará la respuesta.

Generación del contenido

La propiedad `Producer` de cada acción apunta al componente que ha de generar la respuesta ante una solicitud que llegue con la ruta asociada. Ese componente puede ser cualquier derivado de `TCustomContentProducer`, de los cuales podemos encontrar varios en la página Internet de la Paleta de herramientas (véase la imagen inferior) como `TPageProducer` o `TDataSetTableProducer`.



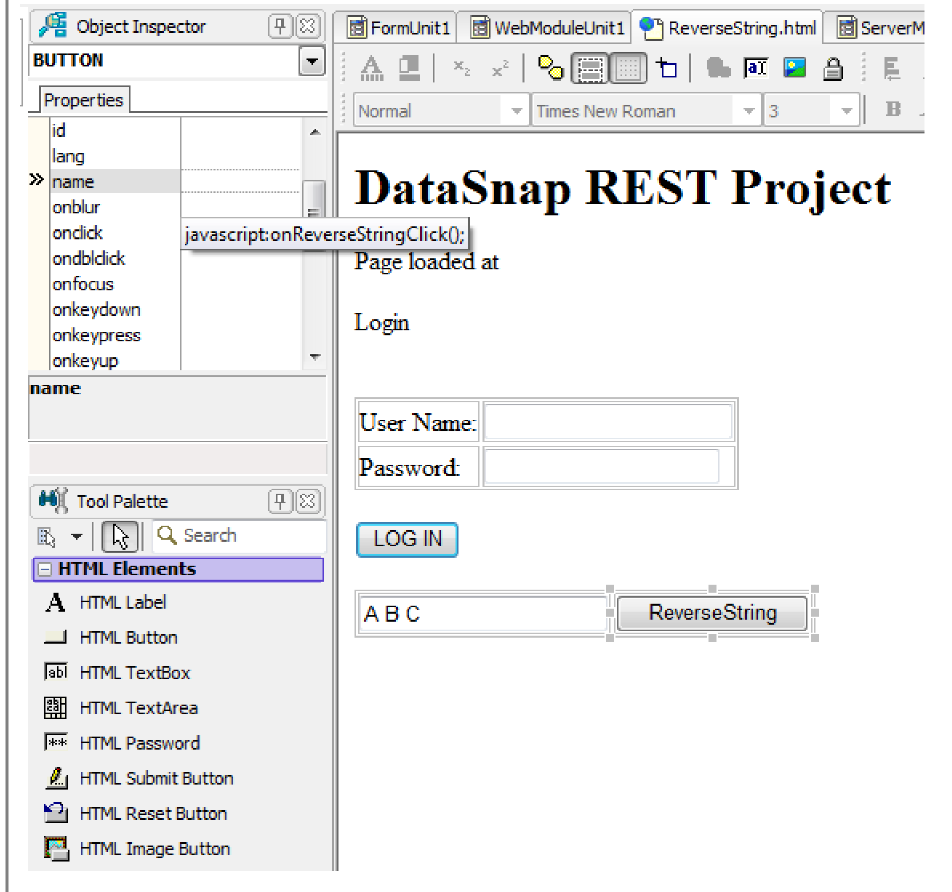
En el módulo generado por el asistente encontraremos dos `TPageProducer` encargados de facilitar el contenido asociado a las rutas `ReverseString` y `ServerFunctionInvoker`. Cada uno de ellos tiene en su propiedad `HTMLFile` el nombre de la página HTML que ha de enviarse al cliente. Alternativamente podría utilizarse la propiedad `HTMLDoc`, de tipo `TStrings`, para almacenar el documento HTML en el propio componente.

En cualquier caso ese contenido no se envía sin más en respuesta a la solicitud, antes se efectúa un preprocesamiento que permite introducir contenido dinámico. Para ello el `TPageProducer` genera un evento `OnHTMLTag` cada vez que encuentra una etiqueta del tipo `<#etiqueta>`, facilitando un parámetro `TagString` que contendrá únicamente el nombre de la etiqueta, sin los delimitadores, y otro llamado `ReplaceText` en el que introduciremos el contenido final a enviar al cliente en sustitución de la etiqueta.

Si no se asocia un productor de contenido a la propiedad `Producer` (o `ProducerContent`) del `TPageProducer`, cuando se reciba una solicitud la acción generará el evento `OnAction`. Éste va acompañado de varios parámetros, entre ellos uno de tipo `TWebRequest` con los datos de la solicitud y otro de tipo `TWebResponse` que es el encargado de preparar la respuesta a devolver al cliente.

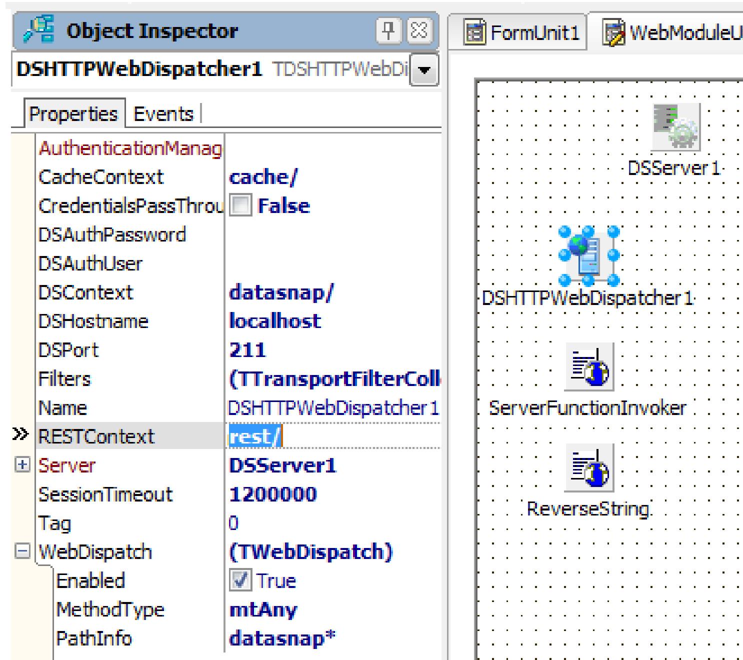
NOTA

Aunque no se trate de una aplicación específica para diseño web, el diseñador de Delphi nos permite componer páginas HTML con los elementos básicos y definir sus propiedades, como puede verse en la imagen adjunta.



Configuración del servidor web

La mayor parte de los parámetros que regirán el funcionamiento del servidor web, exceptuando las rutas de acceso que están asociadas al módulo, se encuentran en el componente TDSHTTPWebDispatcher. Éste establece, por ejemplo, si se llevará a cabo identificación de usuarios y qué componente es el encargado de gestionar esa tarea, los filtros de comunicación que se usarán o el tiempo de expiración de sesión.



TDSHTTPWebDispatcher hereda la mayor parte de sus miembros de la clase de la que deriva: TDSHTTPServerTransport. Ésta utiliza internamente un objeto TDSHTTPServer para gestionar las solicitudes de los clientes, delegando en él la configuración de la parte de la aplicación que actuará como servidor web. Cada solicitud que no haga referencia a una de las rutas que tienen asociada una acción llegará hasta el método DoCommand de TDSHttpServer, en el que se comprobará si la ruta comienza por el prefijo indicado en la propiedad DSContext que, por defecto, es datasnap/. En caso afirmativo se obtendrá la siguiente parte del URL y se verificará si es uno de los siguientes:

- json: Se utiliza para identificar las solicitudes asociadas a JSON.
- tunnel: Corresponden a solicitudes que encapsulan comandos asociados a conexiones DBX.
- RESTContext: Esta propiedad indica cuál será el identificador del contexto REST, siendo por defecto rest/. Las solicitudes de este tipo permiten acceder a los servicios expuestos y se ajustarán a la sintaxis base/datasnap/rest/clase/método/arg.
- CacheContext: Esta propiedad contiene por defecto el valor cache/, indicando que éste identifica al contexto que permite obtener contenidos de la caché o eliminarlos. Los elementos de la caché se gestionan mediante la sintaxis base/datasnap/cache/ID.

Para atender a las solicitudes de tipo REST se delegará en un objeto de clase TDSRESTService que, dependiendo del tipo HTTP de la solicitud, procederá a invocar al método indicado en el URL o una de sus variantes. Suponiendo que el método ofrecido fuese TServerMethods1.getHora, el URL base/datasnap/rest/TServerMethods1/getHora ejecutaría uno de los siguientes métodos:

- getHora: Si para enviar la solicitud se uso el método GET.
- AcceptgetHora: Si para enviar la solicitud se usó el método PUT.
- UpdategetHora: Si se envió la solicitud con el método POST.
- CancelgetHora: Si se envió la solicitud con el método DELETE.

NOTA

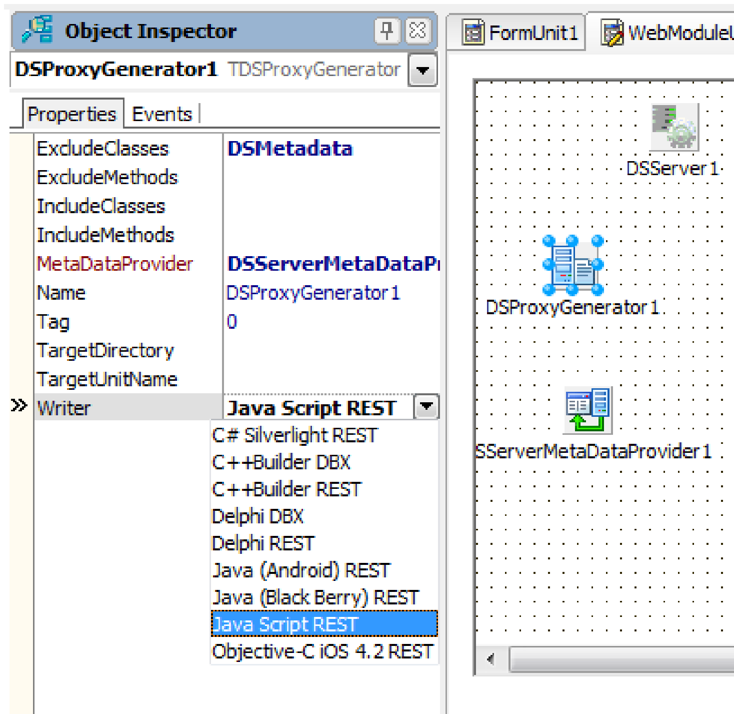
En ocasiones tendrá sentido el uso de ciertos métodos HTTP y en otros no, dependiendo de la naturaleza del servicio. Para getHora, que supuestamente devolvería la hora del servidor, solamente el método GET sería aplicable, pero cuando se opera sobre datos las cuatro operaciones resultan útiles.

Por tanto, de los valores que asignemos a las propiedades DSContext y RESTContext dependerá la ruta que haya que utilizar para acceder a los servicios del servidor mediante la técnica REST. Es necesario tener esa información para codificar la parte cliente de la aplicación: páginas HTML, especialmente los formularios, y código JavaScript asociado a los eventos de los elementos HTML.

Proxys JavaScript de acceso a servicios

Consumir los servicios de un servidor DataSnap a través de peticiones HTTP y REST requiere, como acabamos de comprobar en el punto previo, conocer el nombre de cada clase y el nombre de sus métodos, la lista de parámetros que aceptan y el resultado que devuelven y también los métodos HTTP a que responden.

Escribir todo el código JavaScript necesario para invocar a cada método e interpretar la respuesta sería un trabajo realmente tedioso que, por suerte, no es necesario, ya que Delphi cuenta con componentes capaces de generar un *proxy* en JavaScript que facilite esa tarea. Encontramos dichos componentes ya en el módulo web (véase la imagen inferior): un componente `TDSServerMetaDataProvider` y un `TDSProxyGenerator`.



El primero se conecta con el `TDSServer` y, a través de éste, obtiene todas las clases que ofrecen servicios y recupera información sobre cada uno de sus miembros. Esos datos los envía al `TDSProxyGenerator`, ambos están

enlazados a través de la propiedad MetaDataProvider de este último, a fin de producir el *proxy* en el lenguaje que establezca la propiedad Writer. Por defecto el valor de ésta será Java Script REST que es precisamente el que nos interesa[101].

A pesar de que al seleccionar el componente TDSProxyGenerator vemos aparecer la opción Generate DataSnap client classes, tanto en el Inspector de objetos como en el menú contextual, al intentar ejecutarla obtendremos un mensaje indicando que no es posible completar la operación en la fase de diseño. Es algo que, por tanto, debe hacerse durante la ejecución, usando para ello el método Write de dicha clase.

> NOTA
>
> En realidad no tendremos que preocuparnos por la generación del *proxy* si hemos creado el proyecto con el asistente, ya que éste ha incluido el código necesario para ello como comprobaremos de inmediato.

Solicitudes de archivos

Hasta el momento, en los elementos que hemos analizado, el servidor ofrece medios para acceder a los servicios expuestos, mediante REST y el *proxy* JavaScript, y también contenidos asociados a ciertas rutas gracias a las acciones del TWebModule y los componentes TPageProducer.

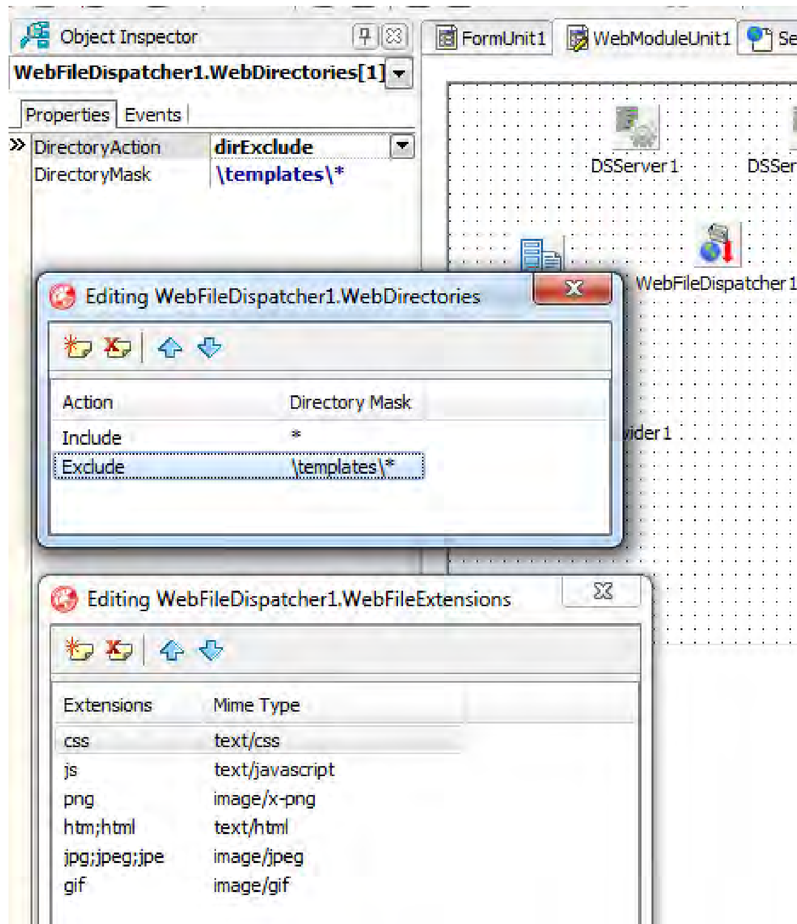
En el Gestor de proyectos podemos ver que existen varias carpetas con archivos adicionales: imágenes, hoja de estilo y archivos con código JavaScript no generado dinámicamente. El cliente (el navegador que emplee el usuario) nos solicitará esos elementos en cuanto se haga referencia a ellos en la página web. Para atender esas solicitudes tenemos en el módulo web un componente TWebFileDispatcher.

Mediante la propiedad WebDirectories de este componente, una colección de objetos TWebDirectoryItem, configuraremos los directorios a los que se tendrá acceso desde los clientes y aquellos a los que no. Cada objeto tendrá

---

101 Las demás propiedades de la clase TDSProxyGenerator permiten configurar la generación del *proxy*, dando un nombre al módulo de código: TargetUnitName, indicando dónde debe ser almacenado: TargetDirectory y excluyendo determinadas clases y/o métodos: ExcludeClasses y ExcludeMethods.

asociada una máscara: propiedad DirectoryMask, y una acción: propiedad DirectoryAction. Lo habitual es que se permita el acceso a todos los subdirectorios del directorio donde está la aplicación, usando * como máscara y dirInclude como acción, excluyendo después cada uno de los directorios que no interese poner a disposición de los clientes, mediante acciones dirExclude individuales. Puede verse esta técnica en la configuración por defecto que es la mostrada en la imagen inferior.



En la carpeta templates del proyecto se encuentran los documentos HTML asociados a los componentes TPageProducer, archivos que no están pensados para enviarse directamente al cliente ya que, como sabemos, dicho componente hará un preprocesamiento incluyendo contenido dinámico.

La segunda propiedad de interés es `WebFileExtensions`, cuya finalidad es asociar a cada extensión de archivo el tipo MIME[102] que le corresponda. Esta información es fundamental para que el navegador pueda procesar correctamente el contenido de cada archivo. En la imagen de la página anterior puede verse que el asistente ya ha definido los tipos correspondientes a imágenes, páginas HTML, hojas de estilo y módulos JavaScript. Si introdujésemos otros módulos en el proyecto, por ejemplo *applets* Java o contenido en Flash, deberíamos agregar aquí la asociación entre extensión y tipo MIME.

Además de las dos propiedades citadas este componente también expone dos eventos: `BeforeDispatch` y `AfterDispatch`. El primero se produce antes de preparar la respuesta a la solicitud recibida, permitiendo personalizar esta fase o incluso sustituir completamente la acción por defecto. En ese caso sería necesario dar el valor `True` al parámetro `Handled` que se recibe en el método asociado al evento. Aprovechando dicho evento el código generado por el asistente comprueba si el *proxy* JavaScript que facilita el acceso a los servicios ha sido creado y se corresponde con la versión actual del proyecto (se comparan las fechas de modificación), procediendo si es necesario a crear o actualizar dicho archivo sirviéndose del componente `TDSProxyGenerator`.

Interfaz del servidor

El único módulo del proyecto que nos queda por analizar es el correspondiente al formulario que actuará como interfaz del servidor, un formulario VCL con los elementos necesarios para detener y poner en marcha el servidor, cambiar el puerto en que éste escucha y abrir la aplicación en el navegador por defecto.

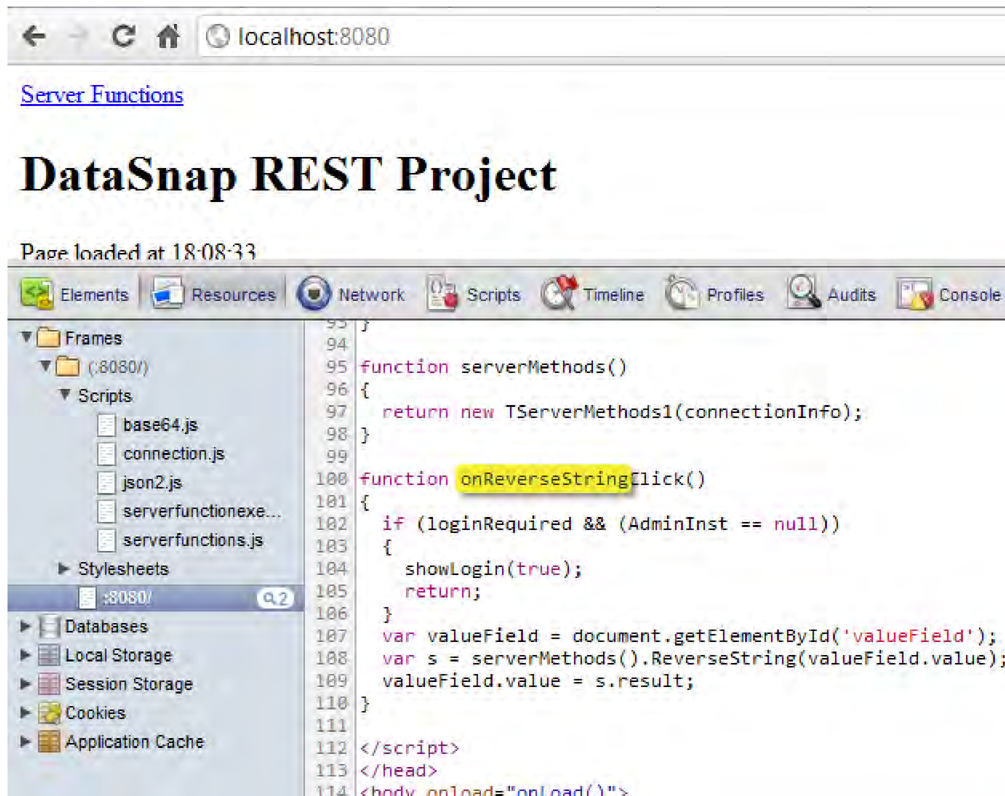
Al igual que hicimos en el séptimo capítulo, podríamos agregar a esta interfaz una lista para mostrar información sobre las conexiones efectuadas por los clientes, los recursos que se solicitan, etc.

---

102 *Multipurpose Internet Mail Extensions*. Es un estándar de Internet que facilita el tratamiento de distintos tipos de archivos por parte de los clientes, sean éstos navegadores web, clientes de correo o de otro tipo. Hay que tener en cuenta que dichas aplicaciones pueden desconocer la extensión que tenía originalmente el archivo o, incluso, no reconocerla. Con MIME se le indica el `tipo/subtipo` del archivo para facilitar su interpretación.

Ejecución de la aplicación

Al ejecutar este proyecto veremos aparecer la interfaz del servidor: el formulario que acaba de describirse en el punto previo. No tenemos más que lanzar el navegador y obtendremos la página por defecto que es la misma asociada a la ruta `ReverseString`, donde podremos introducir una cadena cualquiera y obtenerla invertida. Esa conversión se lleva a cabo en el servidor DataSnap, hasta el que llega la solicitud HTTP gracias al código JavaScript que hay embebido en la página web. Podemos observar este código con las herramientas de depuración/desarrollo de nuestro navegador, como se muestra en la imagen inferior.



La pulsación del botón ReverseString se traduce en la llamada a la función JavaScript `onReverseStringClick`, desde la que se invoca al método `ReserveString` de un objeto devuelto por la función `serverMethods`.

Podemos encontrar la implementación de dicho método JavaScript en el módulo ServerFunctions.js, del que se ha extraído el fragmento mostrado a continuación:

```
function TServerMethods1(connectionInfo)
{
  this.executor = new ServerFunctionExecutor(
                "TServerMethods1",connectionInfo);
  this.ReverseString = function(Value) {
    var returnObject = this.executor.executeMethod(
      'ReverseString', "GET", [Value], arguments[1],
       true, arguments[2], arguments[3]);
    if (arguments[1] == null) {
      if (returnObject != null &&
          returnObject.result != null &&
          isArray(returnObject.result)) {
        var resultArray = returnObject.result;
        var resultObject = new Object();
        resultObject.Value = Value;
        resultObject.result = resultArray[0];
        return resultObject;
      }
      return returnObject;
    }
```

Este código se limita a establecer el método a emplear para efectuar la solicitud: GET, el método a ejecutar y la clase a la que pertenece: ReverseString y TServerMethods1 respectivamente, y los parámetros de entrada y salida. Todos esos elementos se empaquetan y son facilitados a executeMethod, miembro de ServerFunctionExecutor, desempaquetando a continuación la respuesta obtenida.

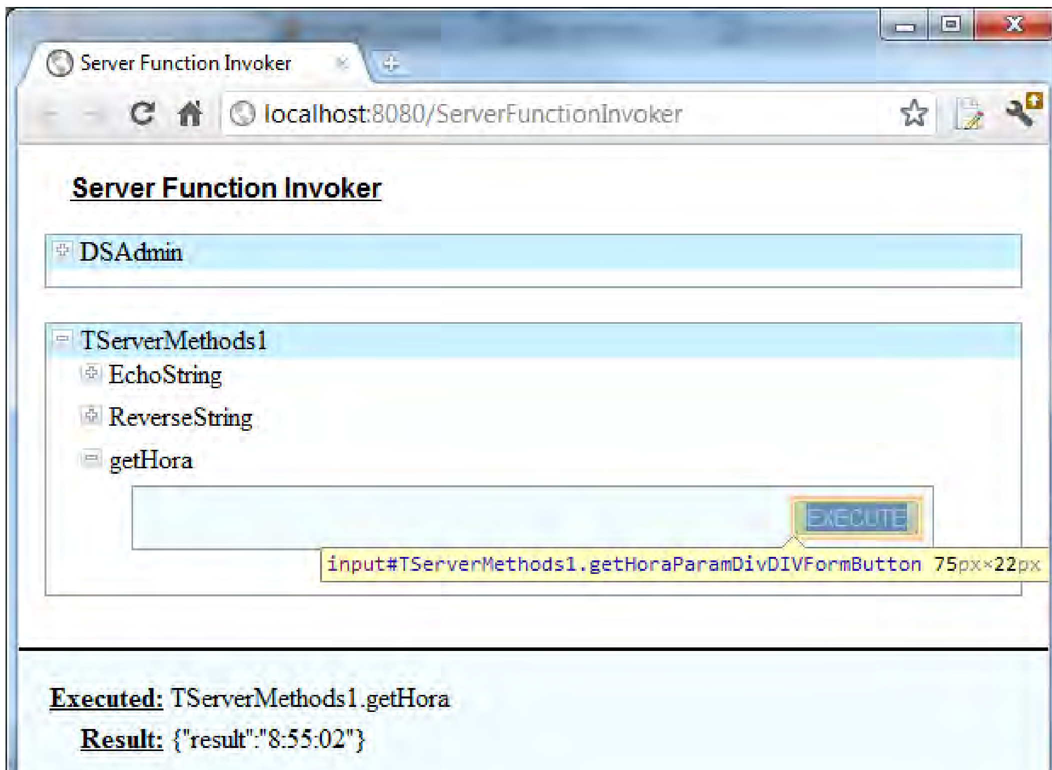
Todo el procesamiento de bajo nivel se encuentra en el código de ServerFunctionExecutor, alojado en el módulo del mismo nombre y con extensión .js. A diferencia de ServerFunctions.js, que es el módulo *proxy* generado dinámicamente a partir de la información que se obtiene de los servicios ofrecidos, el contenido del módulo ServerFunctionExecutor.js es siempre el mismo.

NOTA

Podemos verificar cómo se actualiza el *proxy* simplemente agregando a la clase TServerMethods1 un nuevo método, por ejemplo getHora, y volviendo a ejecutar la aplicación. Veremos aparecer en ServerFunctions.js el código JavaScript necesario para invocarlo.

La segunda página que ofrece la aplicación genera la mayor parte de su contenido dinámicamente, en el servidor DataSnap, a partir de la información obtenida de las clases existentes y los métodos que exponen. Esto permite ejecutar cualquiera de ellos sin necesidad de diseñar una página específica para cada uno, algo que obviamente solo es útil para tareas de depuración y prueba.

Suponiendo que hubiésemos agregado la función `getHora` indicada antes, en el panel de la clase `TServerMethods1` que hay en la página habrá una sección para ejecutar el método. Como se aprecia en la imagen inferior, el valor devuelto es mostrado en la parte inferior de la página.

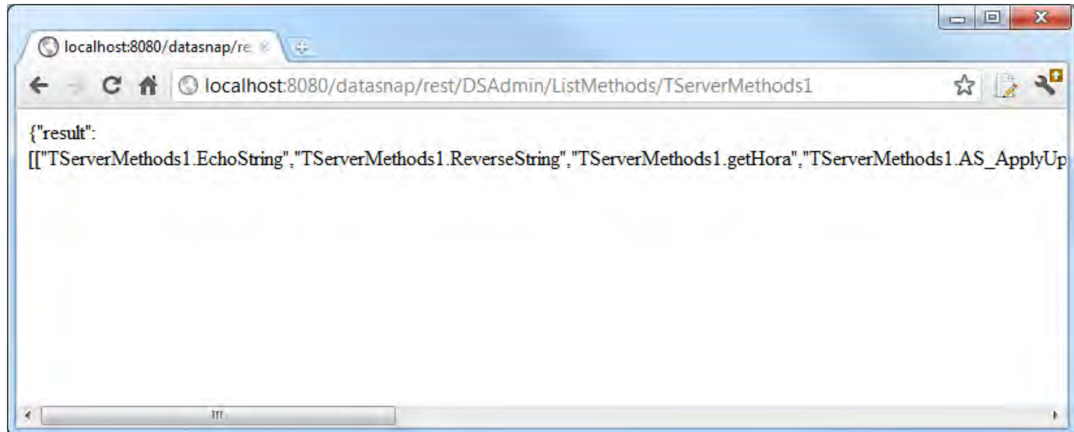


En este caso el resultado es una simple cadena, pero se muestra en el formato en que se ha recibido: JSON. Si el método devolviese una matriz o un objeto veríamos entre las llaves los diferentes elementos/propiedades y su contenido.

Acceso a los servicios REST

En realidad, dado que el servidor funciona de acuerdo a la mecánica de REST, no precisamos de una página para acceder a la mayoría de los servicios ya que podemos usarlos directamente desde el navegador[103], componiendo el URL adecuado.

La clase que ofrece los servicios de administración en un servidor DataSnap es `DSAdmin` (se vio en el séptimo capítulo), contando entre otros con un método llamado `ListMethods` que toma como parámetro el nombre de una clase y devuelve como resultado una lista de los métodos que expone. Podemos ejecutar directamente ese método introduciendo en el navegador un URL como puede verse en la imagen inferior.



`localhost:8080` es el nombre del servidor web y el puerto en el que está a la escucha. La ruta `/datasnap/rest/` activa el contexto REST que se mencionaba en un punto previo. Si hemos modificado las propiedades `DSContext` y/o `RESTContext` habría que adecuar dicha ruta. Tras el contexto se introduce el nombre de la clase y el del método a ejecutar: `DSAdmin/ListMethods`. Finalmente, en caso de que fuese necesario, se agregan los parámetros de entrada que precise el método que, en este caso concreto, es el nombre de la clase cuya lista de métodos se quiere obtener.

---

103 Esto es cierto para los métodos accesibles mediante el método `GET` de HTTP. El resto de métodos: `POST`, `PUT` y `DELETE`, requerirían la composición de una solicitud mediante un formulario HTML o bien código JavaScript.

De manera similar podemos conocer la lista de parámetros que necesita cada método, la lista de clases expuestas por el servidor, etc. Toda esta información nos permitiría consumir cualquiera de los servicios disponibles, por ejemplo:

- `localhost:8080/datasnap/rest/TServerMethods1/getHora`: Se ejecuta el método `TServerMethods1.getHora` que no precisa parámetros. La página mostrará el objeto `result` conteniendo únicamente la cadena con la hora.
- `localhost:8080/datasnap/rest/TServerMethods1/ReverseString/Hola%20qué%20tal%20estamos`: En este caso se facilita al método `ReverseString` una cadena de caracteres como parámetro, obteniéndose como resultado la misma cadena invertida.

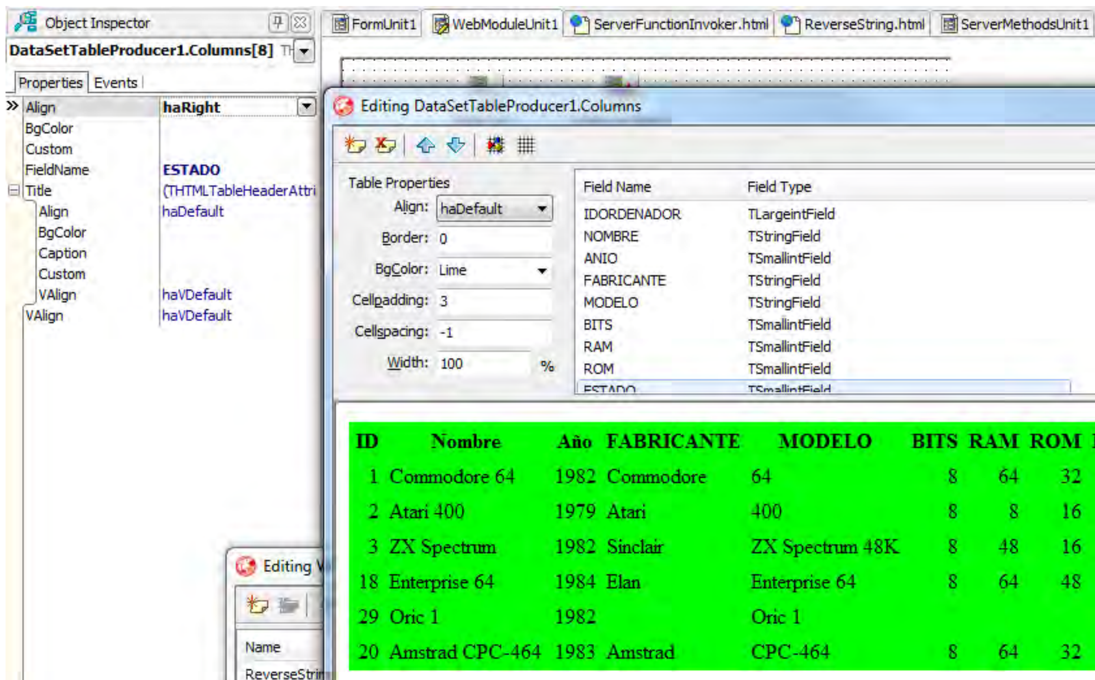
> NOTA
>
> Mediante esta técnica podríamos, por ejemplo, invocar desde el código JavaScript de la página web a un método del servidor DataSnap que obtenga el siguiente valor del generador que definimos en su momento en la base de datos, facilitando así la adición de nuevos datos. Para ello, sin embargo, tendríamos que saber cómo generar una página web a partir del contenido de la base de datos.

Generación de HTML a partir de datos

En un servidor DataSnap de tipo REST, como el que nos ocupa, podemos introducir todos los componentes necesarios para acceder a una base de datos mediante dbExpress, tal y como hicimos en su momento en el séptimo capítulo. No hay diferencias en este sentido.

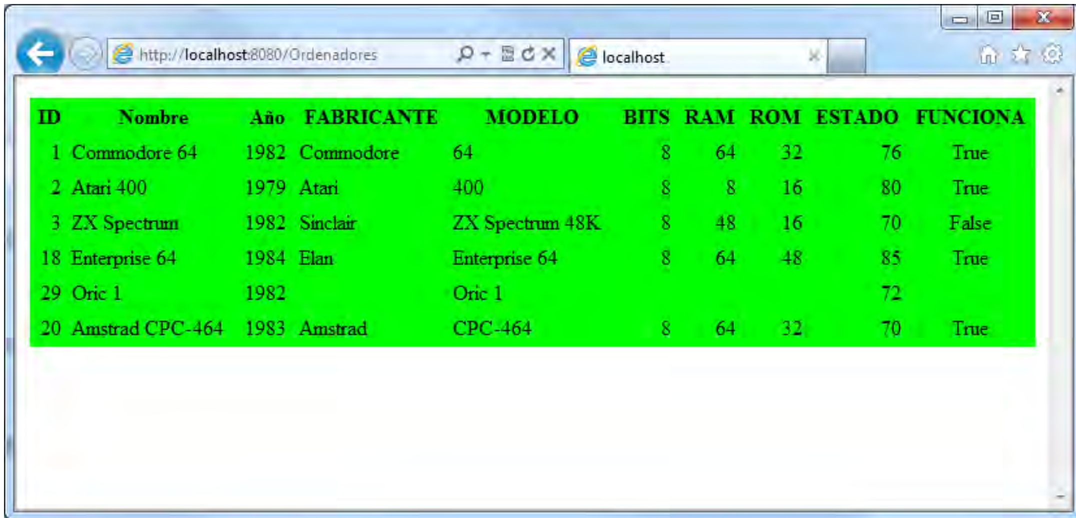
Una vez que se tiene acceso a los datos la primera necesidad que surge es cómo generar código HTML a partir de ellos, páginas que sirvan a los usuarios para ver los datos y trabajar sobre ellos. En la página Internet de la Paleta de herramientas encontramos varios componentes con esa capacidad, entre ellos `TDataSetPageProducer` y `TDataSetTableProducer`. El primero toma una plantilla HTML e introduce en ella los datos en los puntos señalados por marcas sustituibles, mientras que el segundo genera una tabla HTML ajustándose a la configuración indicada.

Asumiendo que hubiésemos agregado al módulo web los componentes dbExpress para conectar con la base de datos del proyecto `MicroClassic`, podríamos insertar un `TDataSetTableProducer` y enlazarlo con los datos mediante su propiedad `DataSet`. A continuación, usando el editor asociado a la propiedad `Columns` (véase la imagen inferior), seleccionaríamos las columnas a mostrar en la tabla y la configuración de cada una de ellas: título, alineación horizontal y vertical, color, etc.



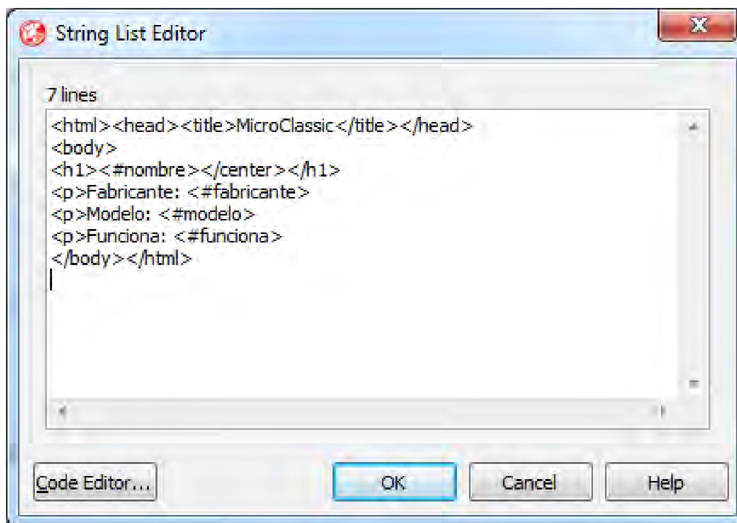
Las propiedades `Header` y `Footer` permiten introducir código HTML a usar como encabezado y pie de la tabla. El número de filas mostradas en ésta estará limitado por la propiedad `MaxRows`.

> NOTA
>
> Para que la página producida por el componente `TDataSetTableProducer` sea accesible habrá que agregar una nueva acción al módulo web, asociándola con una ruta que podamos usar en el navegador tal y como se aprecia en la imagen de la página siguiente.



ID	Nombre	Año	FABRICANTE	MODELO	BITS	RAM	ROM	ESTADO	FUNCIONA
1	Commodore 64	1982	Commodore	64	8	64	32	76	True
2	Atari 400	1979	Atari	400	8	8	16	80	True
3	ZX Spectrum	1982	Sinclair	ZX Spectrum 48K	8	48	16	70	False
18	Enterprise 64	1984	Elan	Enterprise 64	8	64	48	85	True
29	Oric 1	1982		Oric 1				72	
20	Amstrad CPC-464	1983	Amstrad	CPC-464	8	64	32	70	True

Si optamos por usar un `TDataSetPageProducer` tendremos que preparar una plantilla HTML, ya sea en un archivo independiente cuyo nombre facilitaremos en la propiedad `HTMLFile` o introduciéndola directamente en la propiedad `HTMLDoc` (véase la imagen inferior). En esa plantilla es posible usar como marcas sustituibles los nombres de las columnas de datos, lo cual nos permite colocarlas donde nos interese.



Esas marcas podrían formar parte de un formulario HTML, de manera que los datos pudieran ser modificados por el usuario directamente desde su navegador. En el formulario también incluiríamos los botones que facilitarían la navegación, asociando cada uno de ellos con una llamada a un método JavaScript que estaría enlazado, a través del *proxy*, con un método del servidor DataSnap que, a la postre, sería el que actuaría sobre el `TClientDataSet` avanzando, retrocediendo, actualizando, etc.

Conclusión

Al terminar este capítulo conocemos dos de las vías que nos ofrece Delphi XE2 a la hora de desarrollar interfaces de usuario que facilitan el acceso a servicios ofrecidos por un servidor DataSnap.

La primera opción permite consumir esos servicios sin necesidad de introducir cambios en un servidor DataSnap existente, al tiempo que ofrece la comodidad del diseño de la interfaz web con IntraWeb. Es un modelo que agrega una capa adicional al proyecto, al separar el servidor de aplicaciones del servidor web.

Con la segunda opción el propio servidor DataSnap actúa como servidor web, exponiendo sus servicios mediante el protocolo HTTP y la técnica REST. El diseño de la interfaz no resulta tan cómodo ya que no trabajamos con componentes similares a los de la VCL o FMX, sino con plantillas HTML en las que es necesario definir marcas sustituibles y código JavaScript en el servidor.

A continuación

En la siguiente parte del libro nos ocupamos de uno de los aspectos más interesantes de Delphi XE2: su capacidad para generar aplicaciones multiplataforma. Es algo que le diferencia de las versiones previas, que estaban limitadas a producir ejecutables para Windows de 32 bits, y abre las puertas para llegar a un mayor número de potenciales usuarios.

Apartado IV: Aplicaciones multiplataforma

La cuarta parte del libro, compuesta de los cuatro capítulos indicados a continuación, se centra en el desarrollo de aplicaciones multiplataforma con Delphi XE2, una de las novedades más atractivas de esta nueva versión.

- **Capítulo 10: Aplicaciones de 64 bits**
- **Capítulo 11: Interfaces FMX**
- **Capítulo 12: Delphi XE2 y MacOS X**
- **Capítulo 13: Aplicaciones para otras plataformas**

Capítulo 10: Aplicaciones de 64 bits

Delphi XE2 tiene la posibilidad de generar aplicaciones para varias plataformas de ejecución distintas, entre las cuales se encuentra Windows de 64 bits (Win64). Poder escribir código de 64 bits es una característica importante que, en determinados casos, puede llegar a ser indispensable, ya que abre la puerta al tratamiento de grandes cantidades de información alojadas directamente en memoria, no en un archivo o una base de datos.

En este capítulo comprobaremos que crear una aplicación de 64 bits con Delphi XE2 es una tarea sencilla, al tiempo que nos servirá para familiarizarnos con algunas herramientas y opciones del entorno que hasta el momento no hemos usado. Esto nos será útil en capítulos posteriores, al abordar el desarrollo de aplicaciones para MacOS X.

32 bits versus 64 bits

La primera versión de Delphi contaba con un compilador de 16 bits, un hecho que conllevaba serias limitaciones en un tiempo en que los ordenadores personales de 32 bits ya eran habituales, lo cual llevó en poco tiempo a la presentación de Delphi 2.0 y su compilador de 32 bits. La situación actual es similar: los microprocesadores de 64 bits llevan varios años integrándose en ordenadores de consumo, no solamente en servidores y estaciones de trabajo, y los sistemas operativos preparados para operar en esta nueva arquitectura también están disponibles desde hace mucho tiempo. Solamente necesitamos contar con una herramienta de desarrollo que tenga capacidad para generar código de 64 bits pero, ¿para qué?

Esta cuestión se responde fácilmente si se conocen algunos detalles, de relativo bajo nivel, concernientes a la arquitectura hardware sobre la que se trabaja. Los bits que se mencionan asociándolos habitualmente a un microprocesador influyen básicamente en dos aspectos fundamentales: el tamaño de los buses[104] de direcciones y datos y el tamaño de los registros internos con que el microprocesador cuenta para almacenar temporalmente la información sobre la que ha de operar.

Contar un con un bus de direcciones de mayor tamaño implica poder generar más direcciones diferentes, lo cual se traduce en el acceso a una cantidad más grande de posiciones de memoria, es decir: los programas tienen a su disposición más espacio en el que almacenar sus datos. Una aplicación de 32 bits tiene un espacio de direccionamiento teórico de 4 gigabytes, si bien en la práctica la cantidad de memoria que realmente pueden usar es sensiblemente inferior por las necesidades que tiene el propio sistema operativo. Actualmente es habitual que los ordenadores personales dispongan de 6, 8 y hasta 12 gigabytes de memoria,un recurso que quedará infrautilizado a menos que se disponga de software de 64 bits, tanto el sistema operativo como las aplicaciones. En este caso el límite teórico están en el orden de los exabytes, muy por encima de la memoria de cualquier máquina actual.

---

104 El número de pines con que cuenta el microprocesador para comunicarse con el resto del sistema a la hora de enviar/recibir una dirección en la que va a escribirse o de la que se leerá y a la hora de transferir datos entre el microprocesador y la memoria y dispositivos.

El espacio de direccionamiento disponible no afecta únicamente a la cantidad de memoria que los programas pueden usar, sino que influye en muchos otros aspectos. Un caso típico es el uso de los servicios que facilitan el tratamiento de archivos como si estuviesen en memoria (*memory-mapped files*). Actualmente no es extraño trabajar con archivos de más de 4 gigabytes de tamaño y éstos, puesto que tienen más posiciones (bytes) que direcciones pueden generarse con un bus de 32 bits, no pueden ser tratados adecuadamente.

En cuanto al tamaño de los registros del microprocesador, que suele ser el mismo que el del bus de datos, es un aspecto que influye en los tipos de datos que es posible tratar nativamente. La mayoría de los lenguajes de programación cuentan con un tipo para operar con números enteros y dicho tipo suele tener el tamaño del acumulador[105] del microprocesador, lo que significa que tendrá 2, 4 u 8 bytes según que el compilador genere código de 16, 32 ó 64 bits, respectivamente.

Trabajar con tipos de datos de mayor tamaño, por ejemplo valores enteros de 64 bits, significa que podrá procesarse una misma cantidad de información realizando menos transferencias entre la memoria[106] y el microprocesador. Además las operaciones básicas implementadas por la unidad aritmético lógica podrán realizarse sobre datos de mayor tamaño en un solo paso, en lugar de dividirse en varias fases más simples que después es necesario combinar. Todo ello contribuye a que se tarde menos en realizar el mismo trabajo que, a la postre, es lo que todos queremos conseguir.

ADVERTENCIA

Con Delphi XE2 solo podemos generar código de 64 bits para el sistema operativo Windows. Los proyectos para MacOS X siempre serán de 32 bits, a pesar de que la versión del sistema que se utilice sea de 64 bits.

---

105 El acumulador es uno de los registros del microprocesador. Suele actuar como operando y destino en la mayoría de operaciones aritméticas y lógicas.

106 El número de transferencias de datos entre el microprocesador y el exterior es un factor que afecta al rendimiento general de un sistema de manera considerable, ya que la memoria y el resto de dispositivos externos suelen operar a una velocidad muy inferior a la del núcleo del microprocesador. Aunque éste cuenta con mecanismos para adelantarse a las necesidades de un programa en ejecución, como la caché de instrucciones y datos o la predicción de saltos, su impacto es inferior que el que tiene el tamaño del bus de datos.

Las ventajas citadas tienen como contrapartida un pequeño coste: el tamaño de los ejecutables de 64 bits es siempre mayor que el de los ejecutables de 32 bits para un mismo código fuente, asumiendo lógicamente que el compilador no introduce optimizaciones específicas para la primera plataforma respecto a la segunda.

En la mayoría de las aplicaciones se utilizan punteros o bien referencias a objetos (que al fin y al cabo son punteros), almacenando en variables la dirección en que se encuentran los datos u objetos. En el código de 64 bits cada uno de esos punteros requiere el doble de memoria que su contrapartida para 32 bits. En un lenguaje orientado a objetos ésta es probablemente la causa de gran parte del incremento de tamaño del ejecutable, ya que se opera con multitud de objetos y colecciones que, a su vez, mantienen referencias a otros objetos.

Como se indicaba antes este coste puede considerarse pequeño. Para el almacenamiento en disco de la aplicación ni siquiera es significativo. Unos cuantos cientos de kilobytes o incluso algún megabyte de más es apenas apreciable cuando los discos miden ya su capacidad en terabytes. La ocupación del código en memoria tampoco ha de ser un obstáculo ya que, por norma general, cuando se desarrolla una aplicación para sistemas de 64 bits es porque va a ser usada en máquinas con 4 gigabytes de memoria o más.

Aplicaciones de 64 bits en Windows

Lo primero que hemos de tener en cuenta es que para ejecutar las aplicaciones de 64 bits desarrolladas con Delphi necesitamos una versión de 64 bits de Windows. El entorno de Delphi es una aplicación de 32 bits y, como cualquier otro código Windows de 32 bits, funciona tanto en las versiones de 32 bits como de 64 gracias a WOW64[107].

---

107 *Windows-on-Windows 64 bits*. Se trata de una capa software incluida en las versiones de 64 bits de Windows que se encarga de simular un entorno de 32 bits para las aplicaciones que lo necesitan, ocultando las diferencias en la arquitectura hardware real sobre la que se opera: registros de mayor tamaño, buses de direcciones y datos ampliados, etc. Para una aplicación de 32 bits funcionando sobre Windows de 64 bits WOW64 también se encarga de convertir adecuadamente las rutas en el sistema de archivos, en el registro de Windows y otros elementos del sistema que tienen una implementación diferente.

Entre el entorno de ejecución de una aplicación de 32 bits y otra de 64, asumiendo siempre que trabajamos sobre Windows de 64 bits, existe una *barrera* que no es posible cruzar fácilmente. Esto significa que no es posible combinar en una misma aplicación código de 32 bits y código de 64 bits, algo que hemos de tener muy en cuenta si queremos desarrollar proyectos de 64 bits ya que implica que:

- Hemos de contar con versiones de 64 bits de los componentes que tengamos instalados en el entorno de Delphi[108], ya que la versión que usamos en la fase de diseño siempre será de 32 bits.
- Es necesario obtener versiones de 64 bits de todas las bibliotecas (DLL) de las que dependamos para que nuestra aplicación funcione. Si tenemos un programa que conecta con una base de datos Firebird, por ejemplo, tendremos que instalar el software cliente de 64 bits para ese RDBMS. Esto mismo es aplicable para cualquier otro módulo de código externo del que dependa nuestro proyecto.
- Debemos revisar todas las llamadas directas a la API de Windows que existan en nuestro código, verificando no solamente que se invoca a la versión correcta del servicio sino también que los tipos de los parámetros son los adecuados.

Esto se traducirá de manera efectiva en que habremos de trabajar simultáneamente en dos proyectos: una versión de 32 bits que es la que veremos en el entorno, con los componentes y bibliotecas de 32 bits, y una versión de 64 bits que entrará en funcionamiento únicamente al ejecutar el proyecto.

Si estamos usando como sistema de desarrollo una versión de 64 bits de Windows el cambio de contexto, entre 32 y 64 bits, será prácticamente transparente para nosotros durante el proceso de prueba y depuración, ya que a pesar de que ser una aplicación de 32 bits Delphi interactúa directamente con el ejecutable de 64 bits si la ejecución es local: en la misma máquina.

La situación será distinta en caso de que la máquina de desarrollo sea de 32 bits: precisaremos de otro equipo, con Windows de 64 bits, para realizar el despliegue del ejecutable y poder comprobar su funcionamiento.

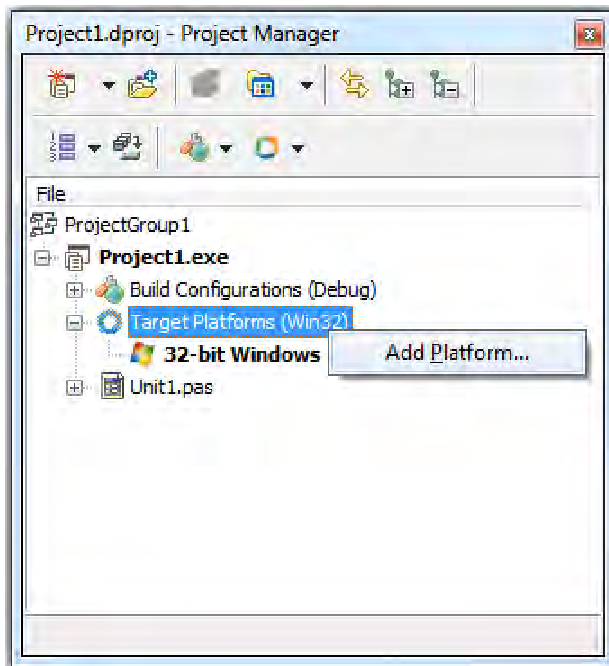
---

108 En la práctica esto implica contar con paquetes de 32 y 64 bits de todos los componentes, ya que si únicamente tenemos versión de 64 bits nos será imposible instalarlos en el entorno de Delphi para usarlo desde los diseñadores. Podríamos emplearlos únicamente mediante código, no visualmente.

Plataformas de un proyecto

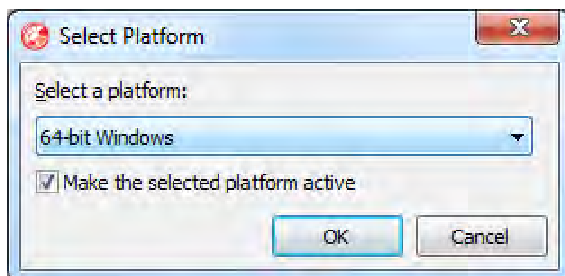
Ahora que tenemos una visión general de lo que implica el desarrollo de una aplicación de 64 bits, las ventajas e inconvenientes que comporta y algunos detalles sobre la forma en que funciona Delphi en estos casos, vamos a describir los pasos a seguir para desarrollar un proyecto de 64 bits.

Cada uno de los proyectos abiertos en Delphi aparece en el Gestor de proyectos como un nodo con varios subnodos, uno de los cuales tiene el título Target Platforms. Dicho subnodo cuenta con un menú contextual que ofrece una sola opción: Add Platform, como puede apreciarse en la imagen inferior.

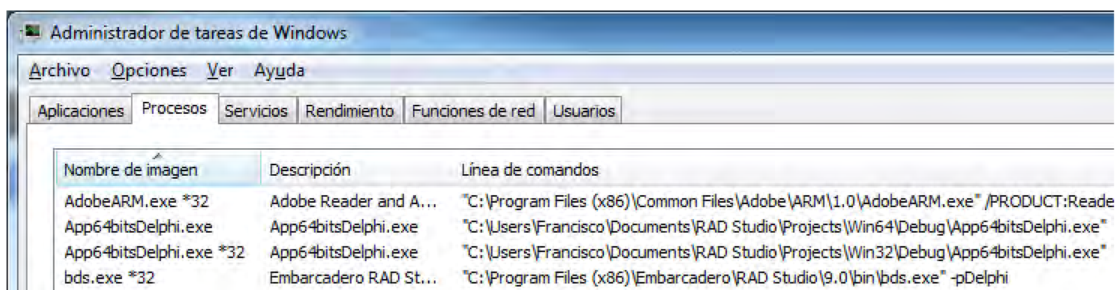


Por defecto la única plataforma objetivo que se añade a cada nuevo proyecto es 32-bit Windows, algo lógico puesto que ésta sigue siendo la plataforma con mayor cuota de mercado (en ordenadores) de forma global. Todos los proyectos que hemos desarrollado en los ejemplos de capítulos previos funcionan sobre esta plataforma: Windows de 32 bits.

La opción indicada antes da paso a un pequeño cuadro de diálogo como el de la imagen inferior, en el que podemos elegir la plataforma a agregar al proyecto. Una lista desplegable nos ofrece tres opciones, entre ellas 64-bits Windows que es la que nos interesa en este momento.



En principio, asumiendo que cumplimos todas las premisas indicadas anteriormente: disponemos de los componentes y bibliotecas en versión de 64 bits y Delphi está ejecutándose sobre Windows de 64 bits, inmediatamente tras agregar la plataforma podemos ejecutar el proyecto. Éste ya será una aplicación de 64 bits, algo que podemos comprobar fácilmente abriendo el Administrador de tareas de Windows (véase la imagen inferior). En un Windows de 64 bits las aplicaciones que se ejecutan sobre WOW64 aparecen con la marca *32. Un ejemplo de ello es el propio entorno de Delphi, cuyo nombre de ejecutable es bds.exe.



> NOTA
>
> Delphi mantiene separado el código objeto de 32 y 64 bits obtenido de un mismo proyecto en dos carpetas independientes: Win32 y Win64. Ambas las encontraremos en la carpeta del proyecto.

Perfiles remotos

¿Qué ocurre si la máquina en que trabajamos con Delphi es de 32 bits o, lo que para el caso es lo mismo, funciona con una versión de 32 bits de Windows? La respuesta es que no podremos ejecutar nuestro proyecto de 64 bits, algo imprescindible para la aplicación de pruebas y depuración del código. Será necesario recurrir a un ordenador con una versión de 64 bits de Windows[109], puede ser el mismo donde vaya a ser usada finalmente la aplicación, en el que presumiblemente no tenemos instalado Delphi ya que de lo contrario no necesitamos dar los pasos que van a describirse a continuación.

Desarrollar en una máquina y ejecutar en otra es una tarea tediosa, ya que requiere la copia de los archivos resultantes del proyecto, incluyendo todas las bibliotecas, componentes y controladores de acceso a datos, actualizándolos[110] con cada cambio que se realice. Además la depuración nos resultará también incómoda, ya que al no existir comunicación entre el ejecutable y el entorno de Delphi nos será imposible ejecutar paso a paso, inspeccionar el estado de los objetos, etc.

Afortunadamente Delphi XE2 contempla este tipo de escenarios y nos ofrece una solución práctica: la definición de perfiles remotos en el entorno de desarrollo que permiten a éste conectar con un servidor de depuración remoto que opera en la máquina donde se ejecuta el proyecto de 64 bits. Lógicamente ambas máquinas han de estar conectadas mediante una infraestructura de red.

> NOTA
>
> El mismo esquema que va a describirse a continuación será el que se emplee también a la hora de probar y depurar aplicaciones para MacOS X, sistema operativo para el que Delphi XE2 puede generar código pero sobre el que no es posible ejecutar el entorno de desarrollo.

---

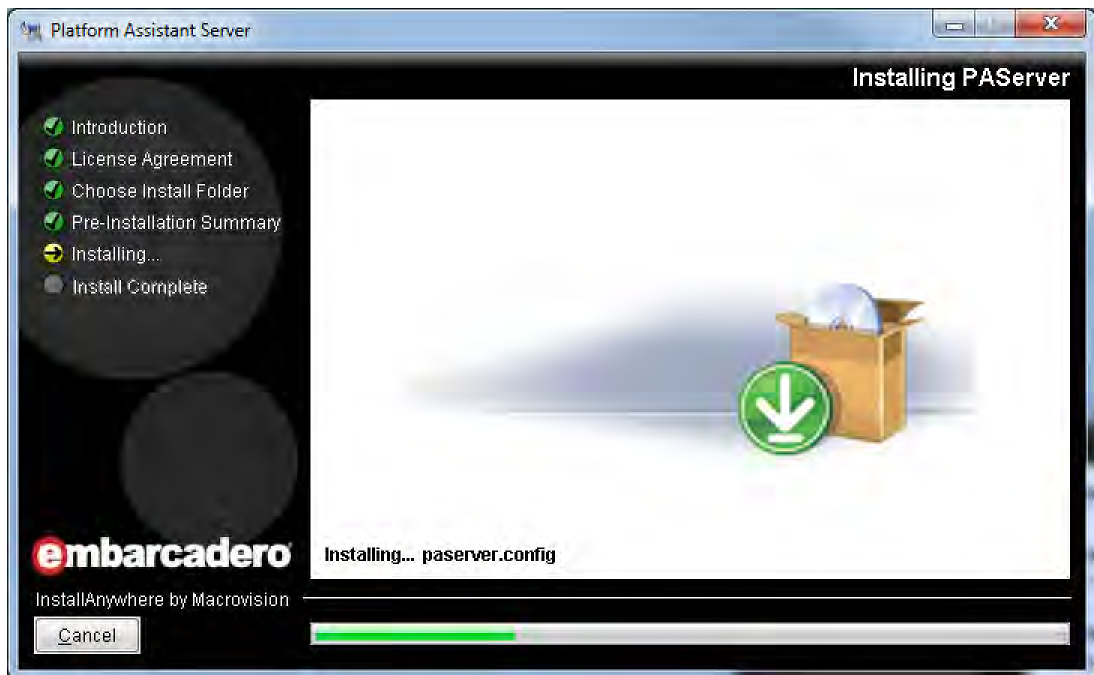
109 Recurriendo a software de virtualización sería posible contar con un Windows de 64 bits ejecutándose en una máquina virtual, todo ello en el mismo ordenador donde tenemos Delphi.

110 En realidad habría que actualizar con cada nueva compilación únicamente el código generado por el proyecto, para el resto de componentes bastaría con una instalación inicial.

Instalación del Asistente de plataforma

Comenzaremos localizando el instalador del Asistente de plataforma (al que llamaremos paserver en adelante) cuyo nombre es `setup_paserver.exe`. Lo encontraremos en la subcarpeta `RAD Studio\9.0\PAServer` de la carpeta donde hubiésemos instalado Delphi que, por regla general, suele ser `C:\Program Files\Embarcadero`. Hemos de tomar ese archivo y copiarlo en la máquina de 64 bits, procediendo a continuación a ejecutarlo.

El proceso de instalación es el típico en la mayoría de las aplicaciones: selección del idioma, aceptación de la licencia de uso, indicación de la ruta destino de la instalación, copia de archivos e informe final.



Tras la instalación no encontraremos ningún elemento añadido al menú Inicio de Windows, por lo que para ejecutar paserver tendremos que acceder directamente a la carpeta donde se ha instalado, por defecto `C:\Program Files\Embarcadero\PAServer`, y hacer doble clic sobre él si hemos usado el Explorador de Windows o introducir el comando `paserver` y pulsar Intro si hemos optado por usar una ventana Símbolo del sistema.

Configuración de paserver

Si lanzamos paserver haciendo doble clic sobre el ejecutable desde el Explorador de Windows, o introduciendo solamente su nombre desde la consola, se utilizará una configuración por defecto, cuyo primer reflejo será la solicitud de una contraseña tal y como se aprecia en la parte superior de la imagen siguiente:

Esa contraseña será la que se solicite para aceptar conexiones[111] provenientes del entorno de desarrollo, constituyendo un mecanismo básico de seguridad. El resto de los parámetros de configuración toman valores por defecto: se escucha por el puerto 64211, se establece como carpeta para el almacenamiento temporal de archivos el subdirectorio `scratch-dir` bajo la carpeta en la que se ejecuta paserver, etc.

Es posible obtener estos parámetros interrogando a la propia aplicación con comandos como `p` y `s`, según puede verse en la imagen superior. El comando `c` enumeraría los clientes conectados en ese momento.

111 El programa `paserver`, como su nombre denota, actúa como si fuese un servidor, aceptando solicitudes enviadas por el entorno de desarrollo para transferir archivos, ejecutar una aplicación y obtener información sobre su estado a fin de facilitar la depuración remota. En este contexto, por tanto, la máquina remota de 64 bits actúa como servidor y Delphi XE2, operando en la máquina de 32 bits, actuaría como cliente.

Ejecutando paserver desde la línea de comandos podemos cambiar esa configuración por defecto mediante las siguientes opciones[112]:

- `password`: Establece la contraseña.
- `port`: Indica el puerto a usar para aceptar solicitudes.
- `scratchdir`: Fija el directorio para almacenamiento temporal.

La sintaxis de uso de estas opciones es la siguiente:

```
paserver -password=contraseña -port=1234 …
```

En caso de que vayamos a ejecutar paserver siempre con un mismo conjunto de opciones, por ejemplo indicando un puerto y directorio alternativos, en lugar de especificar esos parámetros en la línea de comandos podemos almacenarlos en el archivo `paserver.config`. Éste se encuentra en la misma carpeta que paserver y es leído automáticamente cuando se pone el servidor en marcha.

ADVERTENCIA

Por defecto el servidor no permitirá al cliente manipular archivos fuera de la carpeta indicada por `scratchdir`, por lo que ésta actúa como una caja de seguridad que impide el acceso al exterior. Es posible desactivar esta característica mediante la opción `-unrestricted`.

Una vez lanzado, paserver quedará a la espera de recibir solicitudes del entorno de Delphi. Para que éste pueda comunicarse con el servidor será necesario crear un perfil remoto, un tema del que nos ocupamos en el siguiente apartado.

Cuando hayamos terminado la sesión de pruebas o depuración debemos cerrar el servidor, para lo cual no hay más que introducir el comando `q` y pulsar Intro.

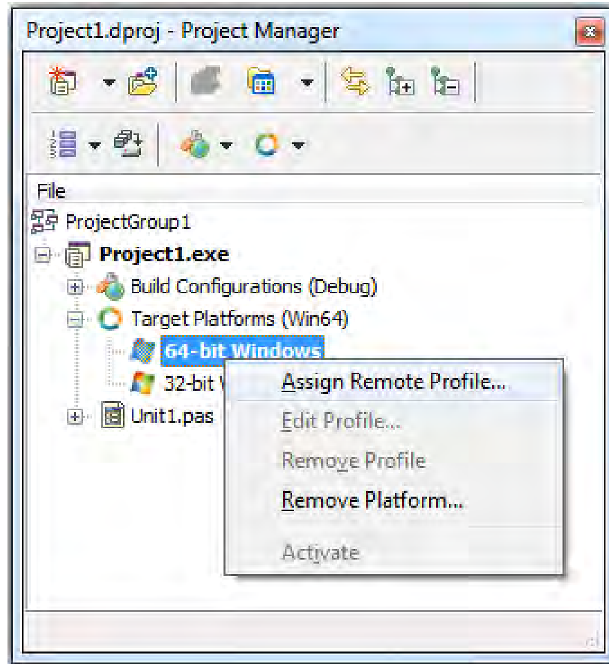
Delphi XE2 sigue contemplando el método de depuración remota heredado de versiones previas, menos cómodo y que requiere una copia manual de los archivos de proyecto y la configuración del servidor de depuración. Es un procedimiento que no aporta ventaja alguna respecto al que acaba de describirse con paserver.

---

112 Podemos obtener una lista de todas las opciones que acepta `paserver` mediante la opción `-help`.

Definición del perfil remoto en Delphi

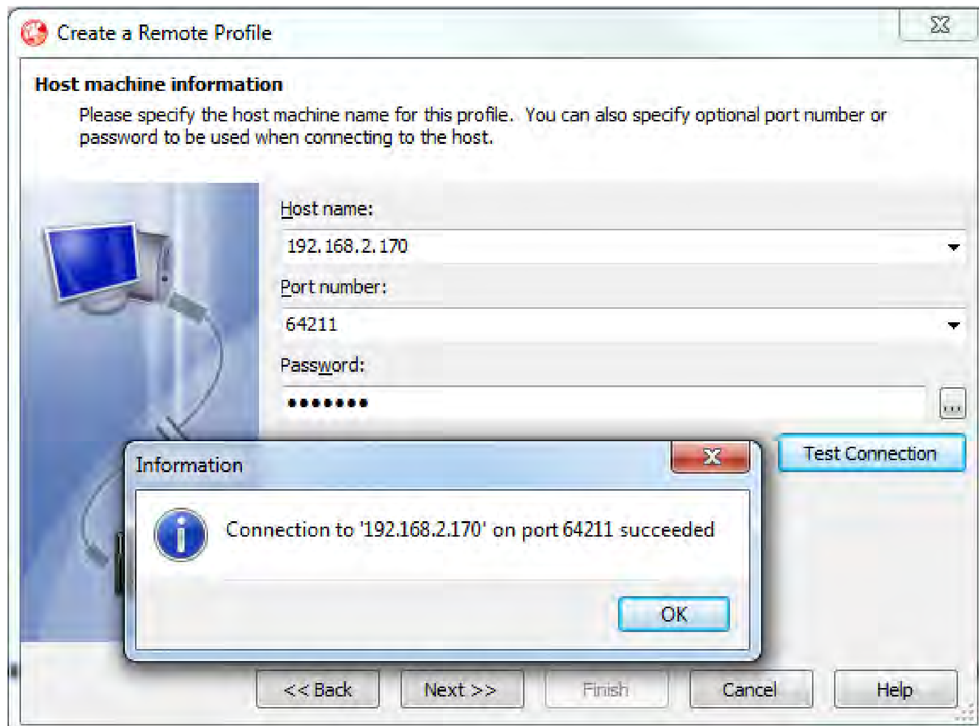
Teniendo el servidor ya en funcionamiento en la máquina de 64 bits, llega el momento de configurar el proyecto en el entorno de Delphi para hacer posible la ejecución y depuración remotas. Con este fin usamos la opción Assign Remote Profile tal y como se muestra en la imagen inferior, abriendo el menú contextual de la plataforma a la que se asociará el perfil.



Los perfiles se definen una sola vez y posteriormente se asocian a tantos proyectos como sea preciso. La opción anterior da paso a una lista con los perfiles que ya existen, inicialmente estaría vacía, de forma que podamos elegir el que corresponda a la máquina remota que deseamos usar.

Puesto que no hay perfiles definidos usaremos el botón Add para añadir uno nuevo. Esto pondrá en marcha el asistente Create a Remote Profile, en cuyo primer paso daremos un nombre al perfil. Normalmente tendremos un perfil por cada máquina remota que vayamos a usar durante el desarrollo, por lo que es recomendable que ese nombre identifique de manera única a cada máquina facilitándonos así la posterior elección.

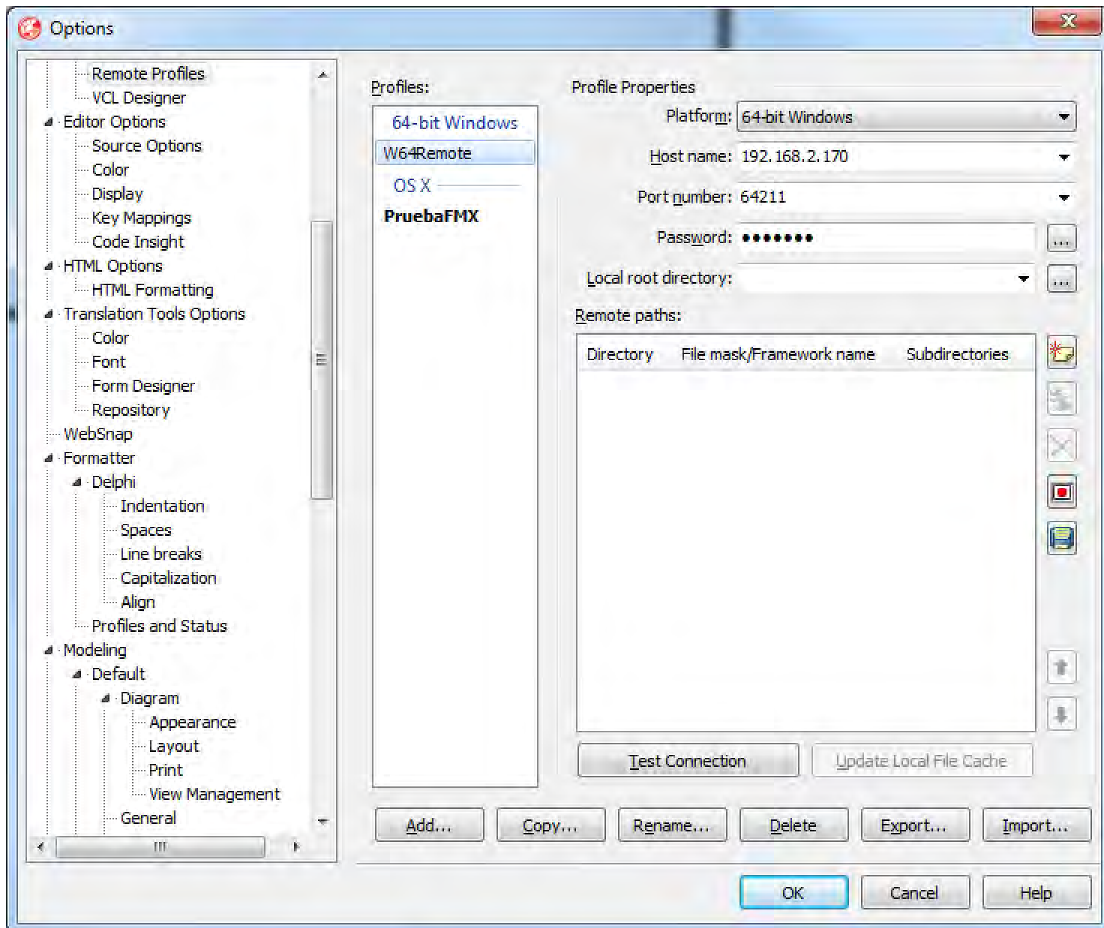
El segundo paso del asistente es el realmente importante ya que en él, como se aprecia en la imagen inferior, debemos facilitar el nombre de la máquina remota (o su dirección IP), el puerto en el que está escuchando el servidor paserver y, si es necesario, también la contraseña que se hubiese introducido al poner en marcha el servidor.



Antes de finalizar la creación del perfil deberíamos usar el botón Test Connection para comprobar que existe conectividad con el servidor. Si la respuesta es un error verificamos que los datos introducidos son correctos y, en caso de que así sea, hemos de asegurarnos de que el software cortafuegos que tengamos instalado en ambas máquinas no está impidiendo la comunicación.

En el último paso del asistente podemos hacer clic directamente en Finish ya que no es aplicable en nuestro caso. En ese momento volveremos a la lista en la que se enumeran los perfiles disponibles, apareciendo en ella el recién creado. No tenemos más que seleccionarlo y hacer clic en en el botón OK para asociarlo con la plataforma 64-bit Windows de nuestro proyecto.

Para proyectos futuros de 64 bits, siempre que vayamos a usar la misma máquina remota para ejecutar la aplicación, podemos agregar este perfil sin necesidad de crear otro nuevo. Si ha cambiado algún dato, como puede ser la contraseña, podemos modificar el perfil con la opción correspondiente del menú contextual. Ésta abre la página general de opciones de perfiles (véase la imagen inferior), desde la que podemos modificar el nombre de la máquina, puerto, etc., y probar de nuevo la conexión.



La lista Profiles que ocupa la parte central de la ventana enumera todos los perfiles definidos, agrupados según la plataforma a que corresponden. Elegido uno de ellos veremos en el panel derecho la configuración actual. Los botones de la parte inferior facilitan la administración de perfiles: crear uno nuevo, copiar uno existente para usarlo como plantilla, exportar, etc.

Despliegue del proyecto

En la mayoría de los casos no tendremos que preocuparnos de ajustar manualmente la configuración del despliegue del proyecto en la máquina remota, ni de realizar el despliegue propiamente dicho ya que es una tarea que se lleva a cabo automáticamente en cuanto ejecutamos la aplicación. No obstante hay casos en que sí puede ser necesario, por ejemplo si nuestra aplicación usa recursos externos que no forman parte del proyecto: imágenes, archivos de configuración, etc.

La opción Project>Deployment abre una nueva página en el entorno que, como puede comprobarse en la imagen inferior, se compone de tres elementos: una barra desplegable que sirve para elegir entre las configuraciones asociadas al proyecto, una barra de botones que da paso a distintas acciones y justo debajo una lista con los módulos que es necesario transferir a la máquina remota junto a sus atributos y estado.

Unit1 | Deployment App64bitsDelphi

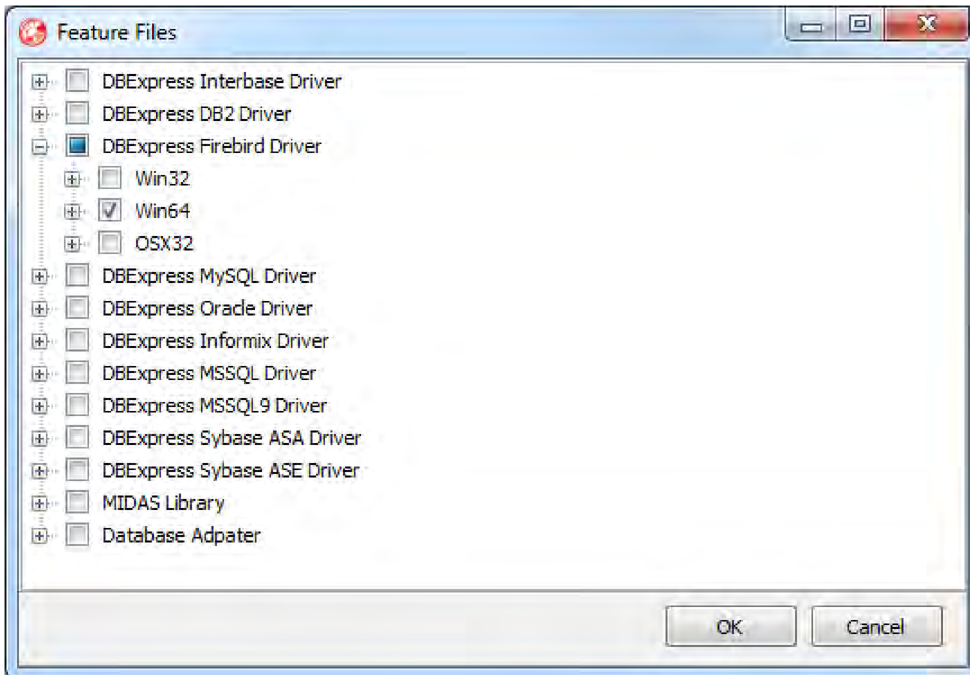
Debug configuration - 64-bit Windows platform

Local Path	Local Name	Type	Platforms	Remote Path	Remote Name	Remote Status
Win64\Debug\	App64bitsDelphi.rsm	DebugSymbols	[Win64]	.\	App64bitsDelphi.rsm	Same
Win64\Debug\	App64bitsDelphi.exe	ProjectOutput	[Win64]	.\	App64bitsDelphi.exe	Same
$(BDS)\bin64\	dbxfb.dll	File	[Win64]	.\	dbxfb.dll	Not Found
Project3\images\	collapse.png	File	[Win64,...	.\	collapse.png	Not Found
Project3\images\	expand.png	File	[Win64,...	.\	expand.png	Not Found

El segundo de los botones sirve para incluir en el proceso de despliegue cualquier archivo accesible desde el entorno de Delphi. Al hacer clic sobre él se abre el típico cuadro de diálogo de selección de archivos, siendo posible elegir varios que se agregarían al final de la lista. Inicialmente se mantendrá el mismo nombre en el sistema remoto, siempre tomando como ruta de destino la carpeta raíz. Son datos que pueden modificarse fácilmente de forma directa en la propia lista en que aparece: un clic sobre el nombre remoto permite cambiarlo, de forma que el archivo se copiará a la máquina destino renombrándolo.

Los archivos añadidos de esta forma inicialmente se asume que hay que incluirlos en las plataformas que tenga asociadas el proyecto, por ejemplo Win32 y Win64. Un clic sobre esa columna nos permite activar y desactivar.

Es muy probable que junto a nuestro proyecto necesitemos distribuir controladores dbExpress y bibliotecas como MIDAS, asociada al componente `TClientDataSet`. La cuarta opción de la barra de botones es Add Feature Files y da paso al cuadro de diálogo mostrado en la imagen inferior. Como puede verse, por cada controlador es posible marcar las plataformas para las que se desplegará.



NOTA

Los Feature Files que gestiona Delphi XE2 corresponden exclusivamente a los controladores dbExpress de acceso a datos, pero en ningún caso se incluye el software cliente de cada RDBMS. Dicho software ha de ser instalado de manera independiente en la mayoría de los casos, recurriendo a la utilidad de instalación del RDBMS que corresponda. En casos concretos, para ciertas versiones de Firebird o Interbase, bastaría con agregar una biblioteca al proceso de despliegue.

El penúltimo botón de la barra tiene la función de conectar con el servidor paserver y comparar las versiones de los archivos entre el equipo local y el remoto, mostrando en la columna Remote Status indicaciones del tipo Same, Older o Not Found, según que los archivos estén sincronizados, el local se haya modificado desde la última operación de despliegue o se trate de archivos recién añadidos, respectivamente. Un clic en el botón precedente, con el icono de una flecha verde, ejecutará la operación de despliegue sincronizando el equipo remoto con el local.

Ejecución y depuración remota

La primera vez que lancemos la ejecución de un proyecto que ha de desplegarse en un equipo remoto notaremos un cierto retardo, ya que han de transferirse todos los archivos que hacen posible la ejecución de la aplicación. En ejecuciones posteriores únicamente se actualizarán los módulos que hayan sufrido cambios, por lo que la respuesta será mucho más rápida.

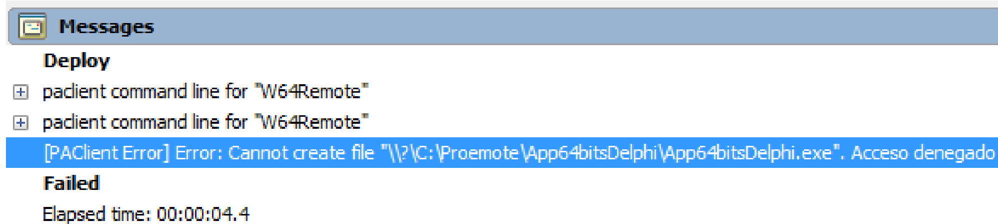
Iniciada la ejecución el proceso es idéntico al que ya conocemos y disponemos de las mismas opciones de depuración: colocación de puntos de parada, evaluación de expresiones, ejecución paso a paso, etc. La diferencia fundamental es que el entorno de Delphi, desde el que accederemos a todas esas operaciones, está en una máquina mientras que la aplicación se encuentra en otra. Obviamente necesitamos tener acceso simultáneo[113] a ambas para poder trabajar cómodamente, sin necesidad de desplazarnos de manera continua entre ambas.

La conexión del cliente (el entorno de Delphi) con el servidor no es continua, sino que se abre y cierra a medida que es preciso. Por ello es posible que el comando c en la consola de paserver informe de que no hay clientes conectados a pesar de que estemos en una sesión de depuración remota. En el entorno de Delphi la ventana Messages, normalmente adosada en el margen inferior, nos permite comprobar los comandos que se están enviando al servidor mediante la utilidad paclient, la parte cliente del asistente de plataforma.

---

113 En caso de que la máquina remota no esté suficientemente cercana a la de desarrollo como para facilitar el proceso, siempre podemos recurrir a la opción Conexión a escritorio remoto de Windows para acceder desde una de las dos máquinas al escritorio de la otra y poder trabajar en paralelo.

Si al iniciar el despliegue del proyecto, lo cual ocurre automáticamente al ejecutarlo tal y como se ha dicho antes, nos encontramos con mensajes de error en la ventana Messages se deberán fundamentalmente a dos razones: la imposibilidad del cliente de establecer conexión con el servidor y la falta de permisos para escribir en la ruta de destino en el equipo remoto. En este segundo caso obtendríamos un error como el mostrado en la imagen inferior.



Para no tener problemas con los permisos es recomendable usar el parámetro `scratchdir` de paserver para apuntar a una carpeta temporal que no sea una subcarpeta de un directorio de sistema, ya que por defecto la configuración de paserver intentará crear un subdirectorio dentro de la carpeta en que está instalado que suele ser `C:\Program Files`, una carpeta de sistema en la que únicamente el administrador puede realizar este tipo de operaciones.

> NOTA
>
> Una alternativa a la selección de una carpeta en la que pueda escribirse sin necesidad de tener credenciales de administrador, en caso de que queramos quedarnos con la configuración por defecto de paserver, es ejecutar el programa `paserver` como administrador, ya sea directamente: con la opción Ejecutar como administrador de su menú contextual en el Explorador de Windows, o bien desde una ventana Símbolo del sistema que se haya abierto con esas credenciales.
>
> En general no es recomendable ejecutar aplicaciones con permisos de administrador salvo que sea imprescindible, por lo que se recomienda la primera opción: usar un directorio no de sistema.

Código Delphi de 64 bits

Ahora que ya sabemos cómo generar una versión de 64 bits de un proyecto Delphi y, en caso necesario, cómo desplegarlo en un equipo remoto para la realización de pruebas y depuración, vamos a concentrarnos en los detalles que es preciso tomar en consideración cuando se escribe código Delphi para una plataforma de 64 bits.

Una de las premisas apuntadas anteriormente es la necesidad de contar con versiones de 64 bits de los componentes que vayan a utilizarse en el proyecto. Tanto la RTL, que es la biblioteca de servicios básicos de Delphi, como la VCL y la FMX, fundamentales para el diseño de interfaces de usuario, cuentan con versiones de 32 y 64 bits para Windows. Esto significa que en principio cualquier tipo de proyecto basado en dichas bibliotecas, y los componentes que se instalan por defecto con Delphi, puede producir una aplicación de 64 bits.

Tipos de datos nativos

Al tratar los tipos de datos enteros[114] supimos que la mayor parte de ellos tendrán siempre el mismo tamaño, indistintamente de la plataforma para la que se compile el proyecto. Esto nos permite definir una variable y saber que siempre podremos almacenar en ella el mismo rango de valores, tanto si la aplicación se ejecuta en un sistema de 32 bits como si lo hace en uno de 64 bits.

Esta *ventaja* de los tipos de datos independientes de la plataforma en ocasiones puede convertirse en un inconveniente, sobre todo cuando es necesario comunicarse directamente con los servicios del sistema operativo. Al invocar a una función de la API de Windows es muy probable que los tipos de los parámetros, tanto de entrada como de salida, difieran según que el sistema sea de 32 o de 64 bits. Es algo que afecta también a los miembros de ciertas estructuras de datos manejadas por el sistema, cuyos tamaños se adaptan a la arquitectura en que están ejecutándose. Por ello Delphi cuenta con los tipos `NativeInt` y `UNativeInt`.

---

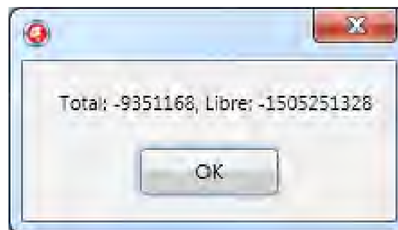
114 Se enumeraron en una tabla en el apartado *Números enteros y en coma flotante* del tercer capítulo.

Estos dos tipos enteros, el primero con signo y el segundo sin él, tendrán un tamaño de 32 ó 64 bits dependiendo de la plataforma para la que se compile el proyecto. Esto hace posible adaptar el tamaño de una variable de forma correcta, por ejemplo si va a utilizarse como parámetro para llamar a un servicio del sistema o para almacenar un resultado.

Veamos un sencillo pero demostrativo ejemplo: queremos obtener la memoria con que cuenta el sistema y la cantidad que aun está libre mostrando ambos datos en pantalla. Para ello vamos a usar la función `GlobalMemoryStatus` con que cuenta la API de Windows, a la que es necesario entregar como parámetro una variable de tipo `TMemoryStatus` [115]. Dicho tipo es una estructura con varios miembros, entre ellos `dwTotalPhys` y `dwAvailPhys` que son los que contienen los datos que nos interesan. Antes de realizar la llamada hemos de asignar al miembro `dwLength` el tamaño de la propia variable, un dato que podemos obtener con el operador `SizeOf`.

Asociamos el código siguiente al evento `OnClick` de un botón en un formulario para conseguir el resultado que buscamos, pero obtenemos lo que puede verse en la imagen inferior:

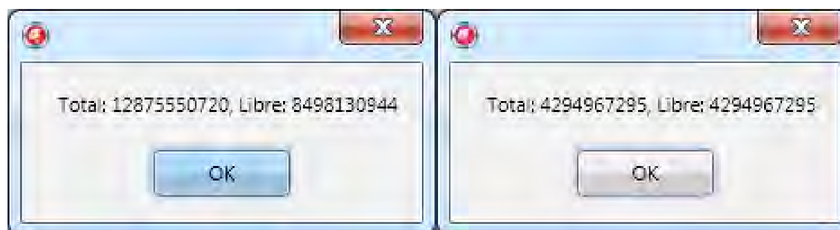
```
var
  memoria: TMemoryStatus;
  total, libre: Integer;
begin
   memoria.dwLength := SizeOf(memoria);
   GlobalMemoryStatus(memoria);
   total := memoria.dwTotalPhys;
   libre := memoria.dwAvailPhys;
   ShowMessage('Total: ' + IntToStr(total) +
               ', Libre: ' + IntToStr(libre));
end;
```



115 Para poder utilizar funciones y tipos de datos propios de Windows es necesario agregar una referencia al módulo `Windows` en la cláusula `uses` de nuestro código.

Lo que ocurre, siempre asumiendo que hemos compilado para 64 bits, es que las variables `total` y `libre` no tienen capacidad para alojar el contenido de los miembros de `TMemoryStatus` indicados, ya que éstos tienen un tamaño de 64 bits. Ese tamaño, sin embargo, se reduciría a 32 bits si el programa fuese de 32 bits.

La solución es cambiar el tipo `Integer` de las dos variables por `UNativeInt`, de forma que tomen el tamaño que corresponda en cada caso. La imagen inferior corresponde a los resultados obtenidos por el mismo programa en sus versiones de 64 (izquierda) y 32 bits. En este segundo caso se informa sobre la máxima cantidad de memoria que podría direccionarse, no del total real de memoria con que cuenta el ordenador que es el que aparece en la versión de 64 bits.



NOTA

En general, cuando vayamos a usar servicios de la API de Windows es preferible definir las variables con los tipos específicos de este sistema operativo, como son `WPARAM`, `LPARAM`, `SIZE_T` o `LRESULT`, todos ellos definidos en el módulo `Windows`. De esta forma no tendremos que preocuparnos por el tamaño de las variables ni el hecho de que tengan o no signo.

En la estructura `TMemoryStatus`, definida en el módulo `Windows`, puede verse cómo los miembros son casi todos de tipo `SIZE_T` que, a su vez, está definido en un punto previo como `UNativeInt`.

Punteros

Una de las mayores fuentes de problemas al escribir código para una plataforma de 64 bits proviene del uso incorrecto de punteros. Las dos operaciones más peligrosas son la conversión de tipo y la aritmética de punteros. En la mayoría de los casos son operaciones que van asociadas en una misma sentencia o grupo de sentencias.

Durante más de una década, desde los tiempos de Delphi 2 hasta la versión XE, hemos podido asumir que el tamaño de un puntero es siempre de 4 bytes, al igual que el tipo Integer y LongInt, por lo que era posible convertir entre dichos tipos sin problemas. Esto nos permitía, por ejemplo, guardar la dirección de una estructura de datos o de un objeto en la propiedad Tag de cualquier componente y recuperarla posteriormente con sentencias como las siguientes:

```
elemLista.Tag := Integer(TMiClase.Create);
...
miObjeto := TMiClase(elemLista.Tag);
```

En un plataforma de 64 bits la expresión TMiClase.Create, con la que se crea un nuevo objeto, devuelve como resultado una referencia (un puntero) de 64 bits, no de 32. Esto significa que al realizar la conversión a Integer estaríamos truncando esa dirección y, en consecuencia, al recuperarla se obtendría una referencia inválida.

Para evitar este tipo de problemas, y conseguir que nuestro código se ejecute correctamente tanto en 32 como en 64 bits, recurriremos a tipos nativos de la plataforma. En el caso anterior bastaría con sustituir Integer por NativeInt y el código funcionaría correctamente siempre, sin que importe si se compiló para 32 ó 64 bits.

En aquellas expresiones en las que se utilice como operando el tamaño que ocupa un puntero no debe asumirse que éste es de 4 bytes. Será así si se compiló para 32 bits, pero no si la plataforma objetivo es de 64 bits. En lugar de una constante debe utilizarse la expresión SizeOf(Pointer) que es independiente de la plataforma, es decir, devolverá el valor 4 u 8 según que el código sea de 32 ó 64 bits.

Las mismas precauciones han de aplicarse cuando, sobre todo al invocar a funciones de la API de Windows, se entregan como parámetros direcciones de funciones, objetos y estructuras de datos. La conversión siempre ha de llevarse a cabo con un tipo independiente de la plataforma.

Conclusión

Delphi XE2 tiene la capacidad de generar a partir del código de un mismo proyecto aplicaciones preparadas para distintas plataformas, entre ellas Windows de 64 bits, gracias a la incorporación de varios compiladores específicos. Estas aplicaciones pueden sacar provecho de las peculiaridades de cada plataforma, lo que en el caso de Win64 significa tener acceso a una cantidad mucho mayor de memoria (toda la que tenga instalada el sistema) y también poder ejecutar de forma más rápida ciertos tipos de operaciones gracias al mayor tamaño de los operandos.

En este capítulo hemos aprendido a agregar una nueva plataforma objetivo a un proyecto, así como a definir un perfil remoto y usar el servidor paserver para poder realizar el despliegue, prueba y depuración en una máquina distinta a la que está ejecutando Delphi. También hemos conocido los detalles más importantes que han de tenerse presentes al escribir código que pueda ejecutarse tanto en 32 como en 64 bits.

Los procedimientos descritos para gestionar una nueva plataforma objetivo en el entorno de Delphi nos serán útiles también al desarrollar aplicaciones para MacOS X, un tema del que nos ocuparemos en el duodécimo capítulo.

A continuación

El tema del que nos ocuparemos en el capítulo siguiente será la biblioteca de componentes FMX, concretamente de sus características avanzadas en relación con la creación de interfaces gráficas de usuario.

A diferencia de la VCL, que es una biblioteca disponible únicamente para proyectos Win32 y Win64, la FMX nos permite contar con interfaces verdaderamente multiplataforma al poder generar código nativo para Win32, Win64 y MacOS X y abrir también la puerta al desarrollo sobre iOS. El conjunto de plataformas objetivo posiblemente se extienda en posteriores versiones de Delphi, alcanzando también a GNU/Linux.

Capítulo 11: Interfaces FMX

La biblioteca FMX, más conocida como FireMonkey, no es un mero rediseño de la VCL para conseguir componentes multiplataforma, si bien ésta es una de sus cualidades más destacables. Es una biblioteca totalmente nueva (en la versión que se incluye en Delphi, ya que FireMonkey es evolución de un producto previo de la empresa KSDev) y con un enfoque distintivo: el aprovechamiento de la potencia que ofrecen las actuales GPU.

Actualmente son muchas las herramientas de desarrollo de aplicaciones que ofrecen componentes visuales y diseñadores que facilitan la composición de interfaces de usuario de última generación, pero en la mayoría de los casos ninguna emplea los recursos gráficos con que cuenta cualquier ordenador relativamente nuevo. FMX sí lo hace y esto le permite llevar las interfaces de usuario un paso más allá.

En este capítulo se describen las características más importantes con que cuentan las interfaces FMX, aquellas que les diferencian de las que podíamos diseñar con versiones previas de Delphi y otras herramientas.

FireMonkey y el hardware gráfico

Los componentes FMX dependen para su funcionamiento de la existencia en el sistema de una GPU *(Graphics Processing Unit)* con unas ciertas características mínimas, algo indispensable para poder producir interfaces gráficas con múltiples estilos visuales, animaciones y efectos 3D manteniendo al mismo tiempo un rendimiento aceptable del resto de la aplicación. Esto implica no sobrecargar la CPU con trabajo *superfluo*[116], que solamente tendrá un reflejo visual, trasladándolo a la unidad más especializada en estas tareas: el hardware gráfico.

Desde hace más de una década los conocidos como adaptadores gráficos o de vídeo que se instalan en los ordenadores no cuentan únicamente con una cierta cantidad de memoria, usada para el almacenamiento de la imagen a visualizar, texturas y otros elementos gráficos; sino que también disponen de su propia unidad de procesamiento de información: la GPU.

Las primeras GPU ofrecían funciones básicas que podían aplicarse a las escenas para generar distintos efectos gráficos, por ejemplo de iluminación, pero no eran programables. No tardaron mucho en aparecer productos con esa posibilidad, de manos de los dos principales fabricantes de hardware gráfico: ATI y NVidia, conjuntamente con el primer modelo de *shaders*[117] definido como estándar y cuya finalidad era básicamente aplicar a cada fragmento de la imagen (cada píxel) una función definida por el usuario.

Los *shaders* pueden escribirse en diversos lenguajes de programación, como Cg, HLSL u OGSL, pero en general las aplicaciones no se comunican directamente con la GPU sino que usan como intermediario una capa de software específica para el sistema operativo en que se trabaje como puede ser DirectX, Quartz u OpenGL. FireMonkey funciona de esta forma y se apoya en Direct2D o GDI+ cuando se trabaja en Windows, en OpenGL o Quartz al operar sobre MacOSX y OpenGL cuando funciona sobre iOS.

---

116 Los programadores tienden a considerar "superfluo" el trabajo que se dedica exclusivamente a conseguir una apariencia visual atractiva , consumiendo memoria y potencia de procesamiento que debería dedicarse a *lo importante*. Hay que considerar, sin embargo, que la interfaz de usuario de una aplicación es el medio que hace posible la comunicación entre dispositivo y la persona que lo utiliza y que, por tanto, es uno de los elementos más importantes de cualquier software si se quiere conseguir que los usuarios estén satisfechos y cómodos con él.

Cualquier PC con menos de 10 años, salvo si cuenta con hardware gráfico integrado en placa base de poca potencia, dispone de una GPU que contempla el uso de la versión 2.0 del modelo de *shaders*, requisito mínimo para el funcionamiento de FireMonkey. En el caso de MacOS X cualquier ordenador con la versión 10.6 o posterior del sistema podrá trabajar con FireMonkey, y para iOS el requisito es tener la versión 4.2 o posterior.

> NOTA
>
> A pesar de que las aplicaciones desarrolladas con FireMonkey pueden ejecutarse sobre versiones posteriores de los sistemas operativos mencionados, por ejemplo iOS 5.0, en general los componentes no podrán aprovechar las nuevas características que incorporan esas nuevas versiones hasta en tanto no se actualice también la biblioteca FMX.

Direct2D versus GDI+ en Windows

FireMonkey usa un conjunto de indicadores definidos en `FMX.Types` para determinar si ha de utilizar o no aceleración hardware para generar efectos visuales y si debe usar Direct2D (cuando se trabaja sobre Windows), emular los servicios Direct2D mediante software o recurrir a GDI+. Dichos indicadores, definidos como variables globales, son los siguientes:

- `GlobalUseHWEffects`: Por defecto toma el valor `True` de forma que se usa la aceleración hardware (la GPU) para generar efectos visuales.
- `GlobalUseDirect2D`: También toma inicialmente el valor `True`, indicando que debe utilizarse Direct2D en lugar de GDI+.
- `GlobalUseDirect2DSoftware`: Se le asigna en principio el valor `False` para impedir que se emulen los servicios de Direct2D mediante software.

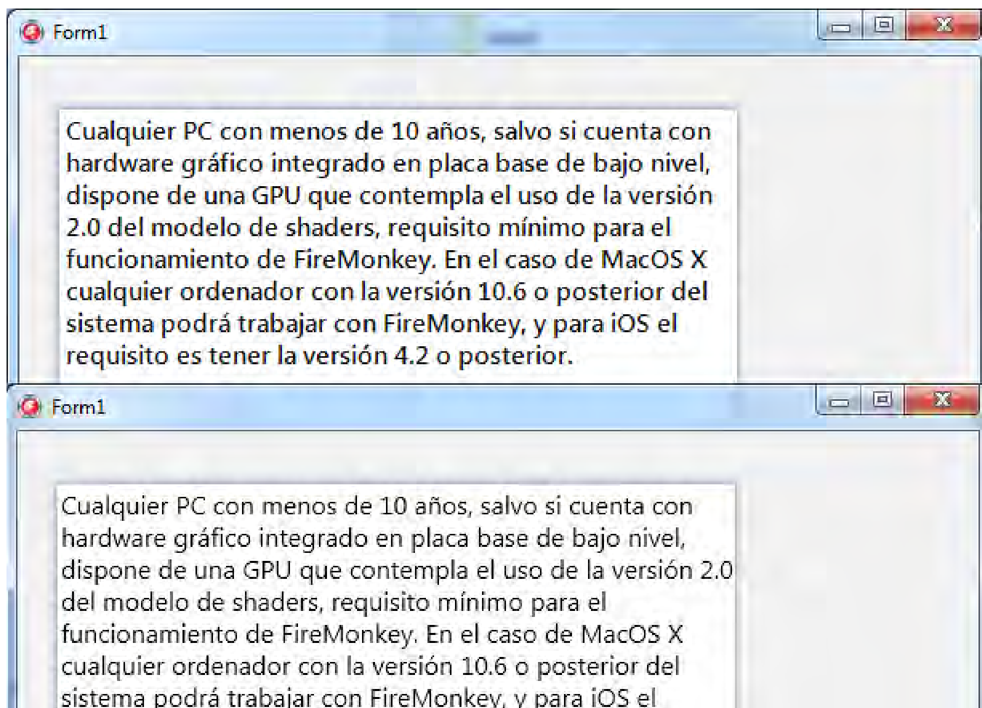
---

117 El término *shader* apareció en la empresa Pixar, concretamente en el equipo que trabajó en la creación de la película *Toy Story*, obra para la cual se desarrolló una tecnología propia de *rendering* gráfico. En 2011 apareció el último modelo de *shaders*, con la versión 4.0, en el que además de *pixel shaders* también existen *vertex shaders* y *geometric shaders*. En http://fcharte.com puedes encontrar un completo curso sobre programación de *shaders*.

- `GlobalDisableFocusEffect`: Por defecto FireMonkey usa un efecto visual para destacar el componente que tiene el foco de entrada, dibujando una especie de halo alrededor del control. Es un efecto que puede ralentizar el funcionamiento de un programa, en especial la navegación por los componentes mediante el tabulador, por lo que es posible desactivarlo asignando el valor `False` a esta variable. Por defecto su contenido es `True`.

La configuración por defecto es aprovechar la aceleración hardware, aplicar el efecto visual asociado al foco de entrada y aprovechar los servicios de Direct2D sin emularlos por software.

Una misma interfaz de usuario puede mostrar un aspecto ligeramente distinto según la API de Windows a la que se recurra: Direct2D o GDI+, especialmente en lo que respecta a la visualización del texto. En la imagen inferior puede verse la misma aplicación, consistente únicamente en un `TMemo` con algo de texto, al usar GDI+ (parte superior) y Direct2D (parte inferior). En el primer caso el texto puede resultar más nítido o intenso.



Por regla general es preferible utilizar Direct2D, sobre todo si la interfaz incorpora efectos de transformación y/o animación como los que se describirán en apartados posteriores, pero si es una interfaz estándar de visualización o introducción de datos en la que lo que prima es la legibilidad posiblemente queramos usar GDI+. Para ello hemos de editar el módulo de código asociado al proyecto, agregando una referencia a `FMX.Types` y asignando el valor `False` a la variable `GlobalUseDirect2D`. Tal y como se muestra a continuación, es una operación a realizar justo antes de que se invoque al método `Initialize` de la aplicación:

```
uses
  FMX.Forms, FMX.Types,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.res}

begin
  FMX.Types.GlobalUseDirect2D := False;
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Éste sería el método a seguir también para modificar cualquier otro de los indicadores antes mencionados.

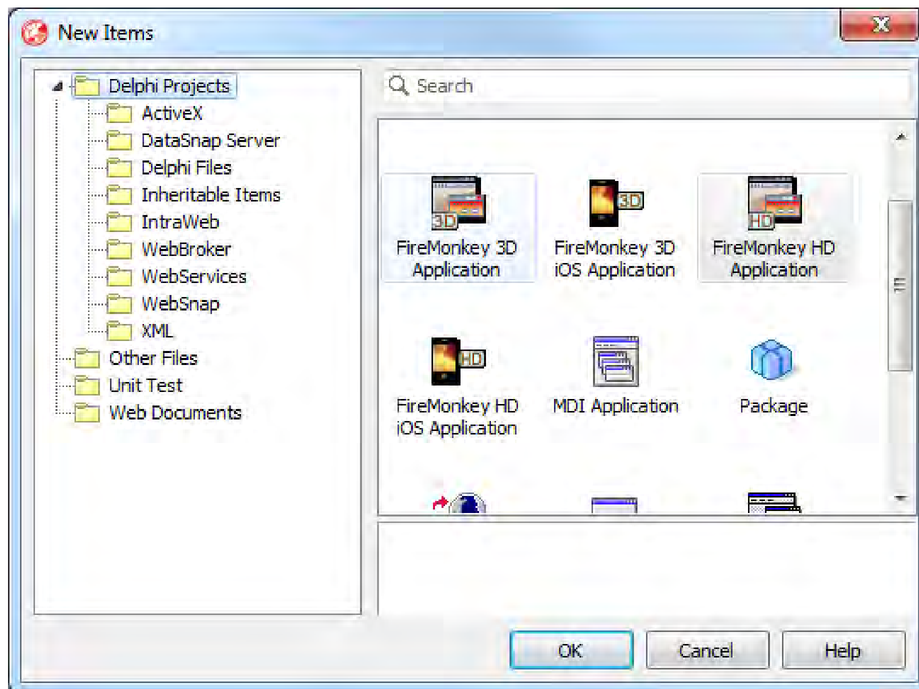
Tipos de interfaces FMX

Una aplicación Delphi basada en la biblioteca FMX puede contar uno de dos tipos diferentes de interfaces de usuario: HD o 3D. Ambos están disponibles para las cuatro plataformas objetivo posibles: Win32, Win64, MacOS X e iOS.

La interfaz de tipo FireMonkey HD se caracteriza por ser muy similar a las interfaces que creábamos con la VCL: tenemos un formulario que actúa como contenedor y una serie de componentes visuales que se colocan en él como en un plano, es decir, en un sistema de coordenadas bidimensional. La denominación HD viene a destacar el hecho de que es posible aplicar a esos componentes estilos visuales, transformaciones, animaciones y filtros con efectos gráficos, todo ello apoyado en la potencia de la GPU, consiguiendo que la interfaz sea mucho más atractiva visualmente hablando.

En contraposición a la anterior, una interfaz FireMonkey 3D usa, como su propio nombre denota, un sistema de coordenadas tridimensional. Esto le permite mostrar objetos que están definidos mediante mallas de puntos en el espacio y aplicar transformaciones 3D. Es un tipo de aplicación que exigirá mucho más a la GPU y que puede dar lugar a un rendimiento pobre si el hardware gráfico con que se cuenta no es lo suficientemente potente.

Al iniciar un nuevo proyecto basado en FireMonkey hemos de seleccionar el tipo de interfaz a usar. Como puede verse en la imagen inferior existen dos parejas de plantillas distintas, una de ellas para los proyectos cuya plataforma objetivo sea Windows/MacOS X y otra distinta cuando el proyecto va a desarrollarse para un dispositivo con iOS.



> NOTA
> Las aplicaciones de tipo FireMonkey HD no pueden contener objetos 3D, pero en una interfaz 3D sí es posible mostrar componentes bidimensionales gracias a un contenedor especializado.

Componentes FMX – Aspectos generales

La mayor parte de los componentes que forman la FMX son controles visuales, clases derivadas todas ellas de la clase `TStyledControl` que, a su vez, es una descendiente de `TControl` y ésta lo es de `TFmxObject`, definidas todas ellas en `FMX.Types`. Es una jerarquía[118] similar a la que existe en la VCL en cuanto a estructura general se refiere, pero con diferencias notables en cuanto a la composición interna de los controles.

El hecho de que todos los controles deriven de un mismo ascendiente conlleva la existencia de una funcionalidad y comportamiento comunes, lo cual nos beneficia como usuarios de estos componentes ya que una gran parte de sus propiedades, métodos y eventos solamente tendremos que aprenderlos una vez y después aplicar ese conocimiento de manera general.

Los apartados de este punto explican brevemente los aspectos básicos comunes que afectan a todos los controles FMX: posición y dimensiones, contenedores, etc. Algunos de ellos, como las animaciones, las transformaciones o los estilos visuales, son descritos de manera más detallada en puntos posteriores de este mismo capítulo.

Distribución de controles en la ventana

Es sin duda una de las tareas a las que se dedica más tiempo durante el diseño de una interfaz de usuario: determinar la posición de cada uno de los controles, sus dimensiones, la alineación de unos respecto a otros y la forma en que han de adaptarse a los cambios que experimente la ventana (o panel) donde están alojados, ya sea al moverse o cambiar de tamaño.

La posición de cualquier control FMX se establece con la propiedad `Position`. Ésta contiene una referencia a un objeto `TPosition` del que, en la mayoría de los casos, solamente usaremos las propiedades `X` e `Y` para fijar la posición horizontal y vertical, respectivamente. Ambas son de tipo `Single`

---

118 El diagrama correspondiente a la jerarquía de controles FMX puedes encontrarlo en http://docwiki.embarcadero.com/RADStudio/XE2/en/FMX.Controls_Class_Hierarchy.

y usan una unidad de medida independiente del dispositivo, por lo que su resolución sería, en caso de necesidad, superior a la que es habitual en la VCL en la que estos parámetros siempre son enteros. En las plataformas que se contemplan actualmente, sin embargo, una de estas unidades siempre corresponde a un píxel físico. El origen de coordenadas `(0,0)` se encuentra situado en la esquina superior izquierda del contenedor.

También podemos tratar la posición como un punto en el plano, mediante la propiedad `Point`, o incluso como un vector de tres componentes: `X`, `Y` y `W`, a través de la propiedad `Vector`. Ésta última sirve básicamente para facilitar la aplicación de transformaciones como las que se describirán más adelante.

> NOTA
>
> La posición de un control FMX siempre son relativas a las coordenadas del componente que actúa como padre y, como se explica en el siguiente apartado, pueden incluso ser negativas.

En cuanto a las dimensiones del control, su ancho y alto, se fijan mediante las propiedades `Width` y `Height` que también son de tipo `Single`. Desde el código del programa se puede establecer simultáneamente la posición y dimensiones mediante el método `SetBounds` entregando cuatro parámetros que corresponderían a la posición: X e Y, y las dimensiones: ancho y alto.

Tanto la posición como las dimensiones de un control pueden verse afectadas en caso de que se produzcan cambios en el componente en que está contenido. Esto dependerá fundamentalmente del valor que se asigne a la propiedad `Align`, cuyo tipo es `TAlignLayout`: una enumeración con una veintena de elementos distintos. El valor tomado por defecto es `alNone`, de forma que el componente no se vea afectado por ajustes automáticos debidos a cambios en su contenedor. El resto de los valores pueden ser agrupados de la siguiente manera:

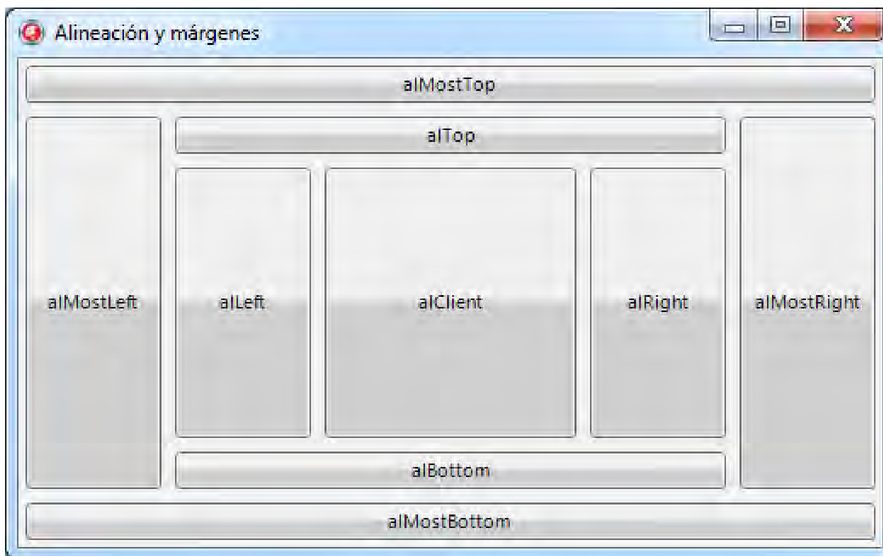
- `alMostTop`, `alMostBottom`, `alMostLeft` y `alMostRight`: Mantendrán el control ajustado al margen superior, inferior, izquierdo o derecho más externo, en los límites de su contenedor. Los dos primeros valores no modifican el alto del componente pero sí el ancho, provocando que ocupen todo el espacio disponible. Los otros dos actúan a la inversa y cambian el alto sin modificar el ancho.

- alTop, alBottom, alLeft y alRight: Actúan exactamente igual que los cuatro anteriores pero formando un anillo más interno, sin llegar a los márgenes del contenedor si es que existen controles con alineaciones del tipo alMostXXX. Si no hay ninguno entonces el resultado sería idéntico.
- alCenter, alHorizontal, alVertical, alHorzCenter y alVertCenter: El control se alineará al centro vertical, horizontal o ambos respecto a su contenedor. El valor alCenter no modifica las dimensiones del control, solamente mantiene su posición, mientras que el resto de valores sí ajustan el ancho o el alto según los casos.
- alFit,alFitLeft, alFitRight y alScale: Modifican las dimensiones del control, en los tres primeros casos para ocupar todo el espacio posible sin cambiar la escala original y manteniéndolo centrado, ajustado a la izquierda o a la derecha. El último valor ajustará el tamaño de control de manera proporcional a la de su contenedor, sin alterar en ningún caso la posición.
- alContent y alClient: El primero ajusta el control de forma que ocupe toda el área de su contenedor, llenándolo y ocultando otros elementos que pudiera haber en él. El segundo también cambia las dimensiones del control para que ocupe más espacio, pero tomando únicamente el que queda libre tras haber colocado el resto de componentes.

Combinando distintos tipos de contenedores, como son TPanel y TGroupBox, con las configuraciones de alineación disponibles y algún elemento auxiliar, como puede ser el componente TSplitter, es posible diseñar interfaces de usuario fluidas que se adaptan sin problemas a cambios de resolución y del propio tamaño de la ventana, prescindiendo por completo del uso de coordenadas absolutas para la colocación.

Cuando se opta por una distribución automática de los elementos de la interfaz como ésta, para evitar que los controles aparezcan totalmente pegados unos a otros hemos de recurrir a las propiedades Padding y Margins, ambas de tipo TBounds. La primera establece la separación entre el borde exterior del control y los elementos que le rodean, mientras que la segunda indica la distancia entre el borde interno del componente y los elementos que contiene. Los formularios no cuentan con la propiedad Padding, ya que es un contenedor de primer nivel, pero sí con la propiedad Margins que podemos usar para evitar que los componentes se ajusten totalmente a los márgenes de la ventana.

Podemos comprobar de una forma sencilla y rápida el efecto de las propiedades Align, Margins y Padding insertando en un formulario un total de nueve componentes TButton (serviría prácticamente cualquier otro) y asignando a la propiedad Align de cada uno de ellos los valores que pueden verse en la imagen inferior. La propiedad Margins del formulario determina la separación entre los márgenes de la ventana y los bordes de los botones que forman el anillo más exterior. Mediante la propiedad Padding de los botones se ajusta la separación entre ellos.



ADVERTENCIA

Al insertar controles FMX debe tenerse en cuenta que se tomará como contenedor el control que se tenga seleccionado en ese instante en el diseñador. Si en éste el control activo es un TButton y hacemos doble clic sobre un TLabel en la Paleta de herramientas el primero se convertirá en padre del segundo y, por tanto, su posición y otros parámetros serán relativos a dicho botón, no a la ventana o un panel. Es recomendable tener abierta la ventana Structure durante la fase de diseño para apreciar cuál es la relación existente entre los controles.

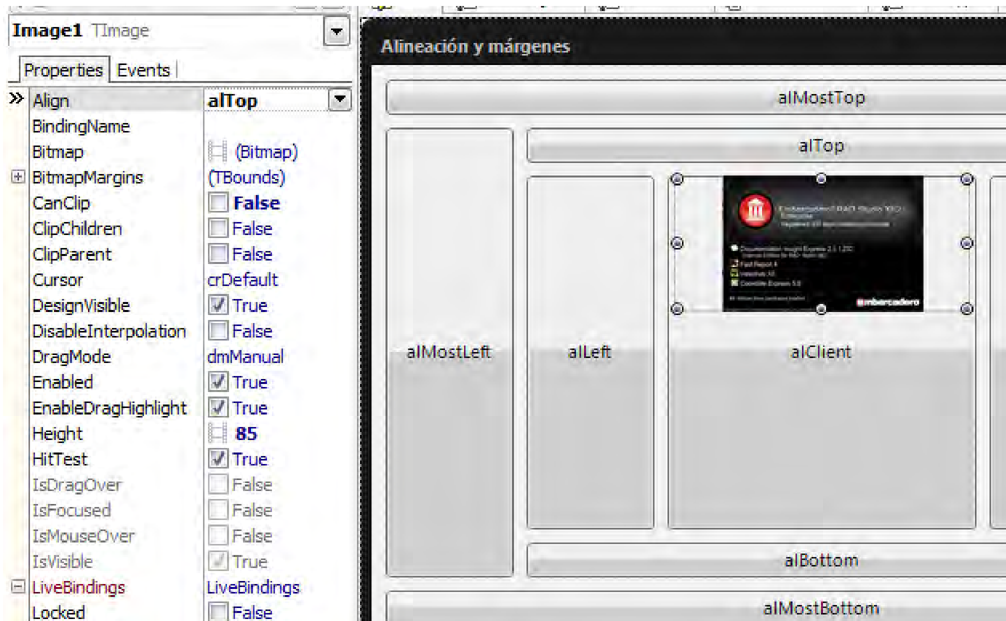
Todos los controles son contenedores

Todos los controles FMX están derivados, directa o indirectamente, de la clase `TFmxObject`, incluso aquellos que no están diseñados para interactuar con el usuario como pueden ser `TPaintBox` o `TLine`. En `TFmxObject` está implementada la funcionalidad básica de cualquier componente FMX: los estilos visuales, las animaciones, la gestión de hijos y padres, etc.

Dos de los miembros fundamentales de `TFmxObject` son las propiedades `Parent` y `Children`. La primera, de tipo `TFmxObject`, mantiene la referencia al componente que actúa como padre del actual. La segunda es una lista (tipo `TList`) de objetos `TFmxObject`: los controles contenidos en el actual. El resultado es que en una interfaz FMX no son contenedores únicamente componentes como `TPanel` o `TGroupBox`, sino que lo son todos los controles, es decir, podemos introducir como hijos de un control a cualquier otro control.

Tomando como referencia un control dado, es necesario distinguir claramente entre su padre: propiedad `Parent`, y el componente en el que se delegará su destrucción: propiedad `Owner`. Esta última está implementada en la clase `TComponent`, ascendiente de `TFmxObject` y nexo de unión entre la VCL y la FMX. El padre y el dueño de un cierto control no ha de ser necesariamente el mismo objeto, de hecho habitualmente el padre será el componente que actúa como contenedor y el dueño suele ser el formulario. El primero es tomado como referencia para determinar la posición y visibilidad del control, mientras que el segundo se encargará de su ciclo de vida. Un botón, por ejemplo, puede estar contenido en un panel que actúa como barra de botones, y que sería su padre, pero el dueño sería el formulario, encargado de destruir todos los componentes una vez que se cierre la ventana.

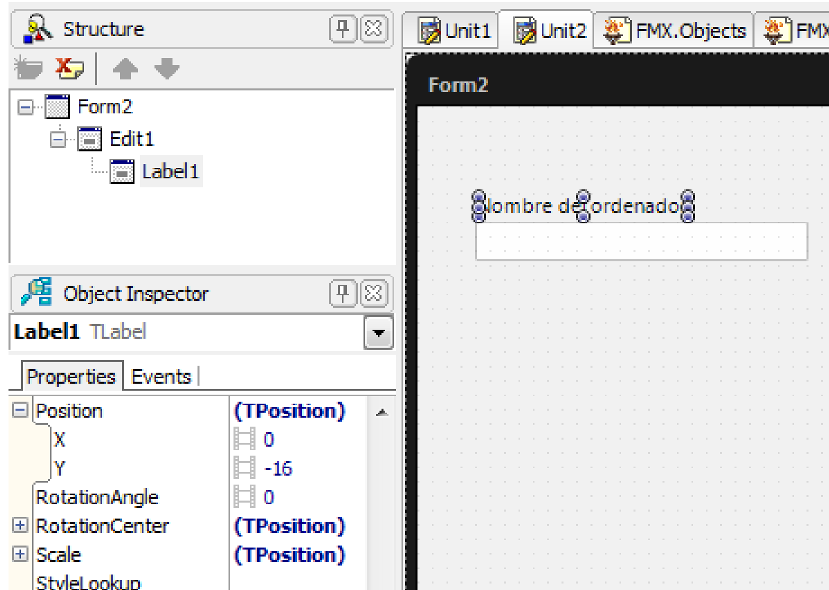
El hecho de que un control pueda ser contenedor de otros nos permite diseñar composiciones virtualmente de cualquier tipo. El control `TButton` no dispone de propiedades que permitan incluir una imagen, ni existe en la FMX un control similar al `TBitBtn` de la VCL. En realidad no es necesario, no tenemos más que tomar un `TImage` de la Paleta de herramientas y agregarlo como hijo de un `TButton`, usando después su propiedad `Align` y la propiedad `Margins` del `TButton` para ajustar automáticamente la posición y dimensiones, tal y como puede apreciarse en la imagen de la página siguiente. Incluso podríamos insertar más de una imagen, sustituir el texto por un `TLabel` con otra configuración visual, etc.



> TRUCO
>
> Siempre que necesitemos asignar el mismo valor a una misma propiedad de varios componentes ahorraremos trabajo si los seleccionamos conjuntamente. Sin embargo al usar la técnica habitual, consistente en mantener pulsada la tecla Mayúsculas mientras se hace clic en los componentes a elegir, el diseñador FMX no refleja la selección de ninguna forma por lo que es difícil saber si hemos marcado o no un cierto control. Nos será más cómodo realizar la selección directamente en la ventana Structure, usando la misma técnica.

Que un control sea contenedor de otro no implica que el segundo esté dentro, visualmente hablando, del primero. Los miembros `X` e `Y` de la propiedad `Position` del control hijo pueden contener valores que le coloquen fuera del área del padre, ya sea porque superen el ancho y alto de éste o bien porque se hayan empleado coordenadas negativas. Seguramente te surgirá la pregunta ¿para qué quiero usar coordenadas negativas?

Un caso típico del uso de coordenadas negativas es el mostrado en la imagen inferior. Se ha introducido un control TEdit y, como hijo de éste, un TLabel que servirá como título del primero. En la ventana Structure puede verse que el padre del TLabel es el TEdit, no el formulario. Justo debajo, en el Inspector de objetos, se aprecia que el miembro Y de la propiedad Position contiene el valor -16. Esto coloca la etiqueta encima del recuadro de texto, manteniéndola siempre alineada ya que el miembro X es 0.



Lo interesante es que, como hijo del TEdit, el control TLabel se mantendrá siempre en esa posición relativa en caso de que el recuadro de texto se mueva o cambie de tamaño. Podemos comprobarlo en el propio diseñador, tomando el TEdit y moviéndolo a otra parte. El TLabel le seguirá, situándose siempre encima de él.

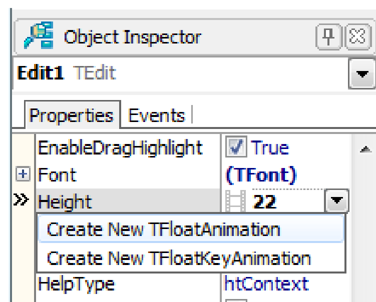
Que un control cualquiera pueda aparecer fuera del área ocupada por su padre es una característica controlada por la propiedad ClipChildren. De tipo Boolean, por defecto contiene el valor False indicando que no se efectuará *recorte* alguno de la visualización de los hijos respecto al área ocupada por el padre. Si le damos el valor el True activamos dicho recorte y, en el caso concreto del ejemplo anterior, la etiqueta de texto no aparecería en el formulario. Dicho estado tiene sentido cuando los hijos están colocados dentro de su padre y no interesa que visualmente salgan de esos límites.

Colores, bordes y fondos

Si hemos desarrollado aplicaciones con versiones previas de Delphi, usando la VCL, al trabajar con componentes FMX echaremos en falta casi de inmediato un gran número de propiedades. Al buscar la propiedad `Color` de cualquier control no la encontraremos, ni siquiera la propiedad `Font` de por ejemplo un `TLabel` ofrece la posibilidad de cambiar el color. Si buscamos en un `TEdit` las propiedades `BorderStyle`, `BevelKind` o `BevelWidth` tampoco las hallaremos. Así podríamos seguir enumerando más y más propiedades *desaparecidas*.

La apariencia visual de los componentes FMX viene determinada por el estilo que se les aplique, entendiendo por estilo no un simple atributo sino una jerarquía completa de elementos cada uno de ellos con sus propiedades. Un estilo FMX se asemeja a los *skins* que usan muchas aplicaciones para cambiar totalmente su apariencia. Es un concepto que también se ha extendido a la VCL[119]. Nos ocuparemos de los estilos en interfaces FMX en el siguiente punto de este capítulo.

Los controles FMX sin embargo cuentan con propiedades que no existían en la VCL, como `Scale` y `Rotation`, que permiten aplicar transformaciones a las coordenadas que establecen su posición y dimensiones. Además muchas propiedades (véase la imagen inferior) pueden tener asociados objetos que cambian su contenido a lo largo del tiempo, generando animaciones. Son temas que también abordaremos en puntos posteriores.



119 La estructura de los estilos en la VCL, así como las herramientas para trabajar sobre ellos, son totalmente diferentes a los de la FMX. El diseñador de interfaces FMX incorpora un editor de estilos y éstos se almacenan en un formato legible y editable por parte del programador, de forma similar a la definición de los formularios en módulos DFM. En el caso de la VCL se depende de una herramienta externa para personalizar los estilos y éstos se almacenan como recursos binarios.

Estilos visuales

FireMonkey no utiliza los servicios de alto nivel con que cuenta el sistema operativo a la hora de crear los componentes de una interfaz de usuario, lo cual tiene, como suele ocurrir con todo, ventajas y desventajas.

Cuando introducimos un `TCheckBox` en un formulario, por ejemplo, este componente FMX no se limita a crear un objeto `BUTTON` (en el caso de Windows) y configurarlo adecuadamente[120]. El propio código de FireMonkey se encarga de dibujar el control en pantalla y de gestionar la interacción con el usuario, lo cual elimina las dependencias del sistema operativo subyacente y facilita la creación de software multiplataforma. El aspecto de las interfaces, sin embargo, puede diferir del nativo de cada sistema, sobre todo si se ha renovado con una nueva versión del sistema operativo y FireMonkey no se ha actualizado para contemplarlo.

El hecho de que en FMX todos los derivados de `TFmxObject` puedan actuar como padres e hijos de otros objetos `TFmxObject`, tal y como se ha expuesto en el anterior apartado, facilita la composición de controles complejos a partir de elementos relativamente sencillos. En este sentido podemos diferenciar dos categorías de componentes:

- Los que derivan directamente de `TControl`: Son componentes que se dibujan a sí mismos, generalmente redefiniendo el método `Paint` definido en `TControl`. Algunos ejemplos son `TImage`, `TRectangle` o `TText`.
- Los que derivan de `TStyledControl`: Son controles que para generar su apariencia visual recurren a la composición, usando una jerarquía de elementos a los que se denomina globalmente el *estilo* del control. A cada estilo se le asigna un nombre y todos ellos se almacenan conjuntamente en un objeto `TStyleBook`, al que puede accederse en cualquier momento a través de la propiedad `StyleBook` del formulario.

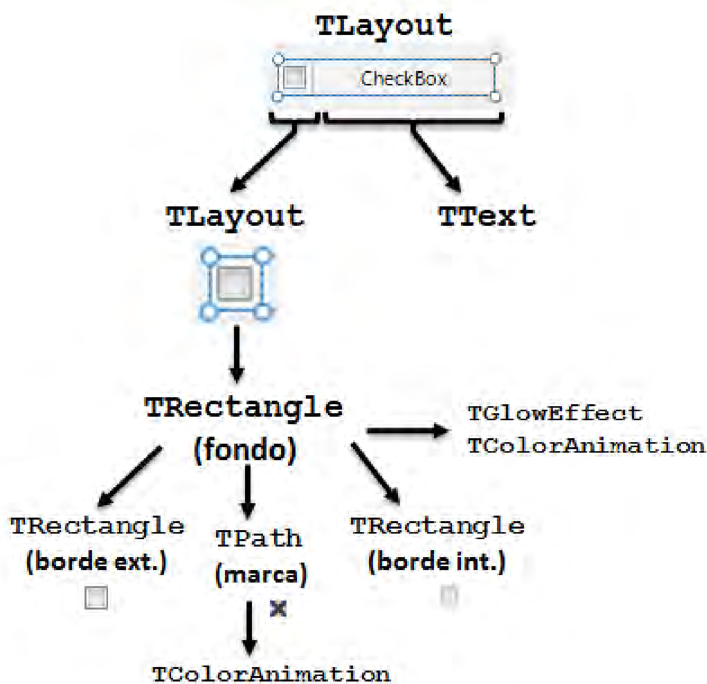
Dado que la clase `TStyledControl` es descendiente directa de `TControl` también existe la opción de redefinir el método `Paint`.

---

120 Ésta es la forma en que funciona la VCL, biblioteca en la que prácticamente todos los controles derivan, directa o indirectamente, de la clase `TWinControl`. Ésta actúa sencillamente como una capa de traducción entre la API y mensajes de Windows y nuestra aplicación.

Composición del estilo de un control

Tomemos como punto de partida el control TCheckBox que mencionábamos antes con el objetivo de analizar la forma en que se compone su estilo visual. El objeto TCheckBox en sí es el responsable de la funcionalidad del control: generación de eventos como OnClick y OnChange, cambiar el estado de IsChecked cuando se hace clic sobre el botón, etc. Para producir la imagen del control, introduciéndola en la posición adecuada de su contenedor, se usa la jerarquía de objetos representada en el diagrama siguiente:

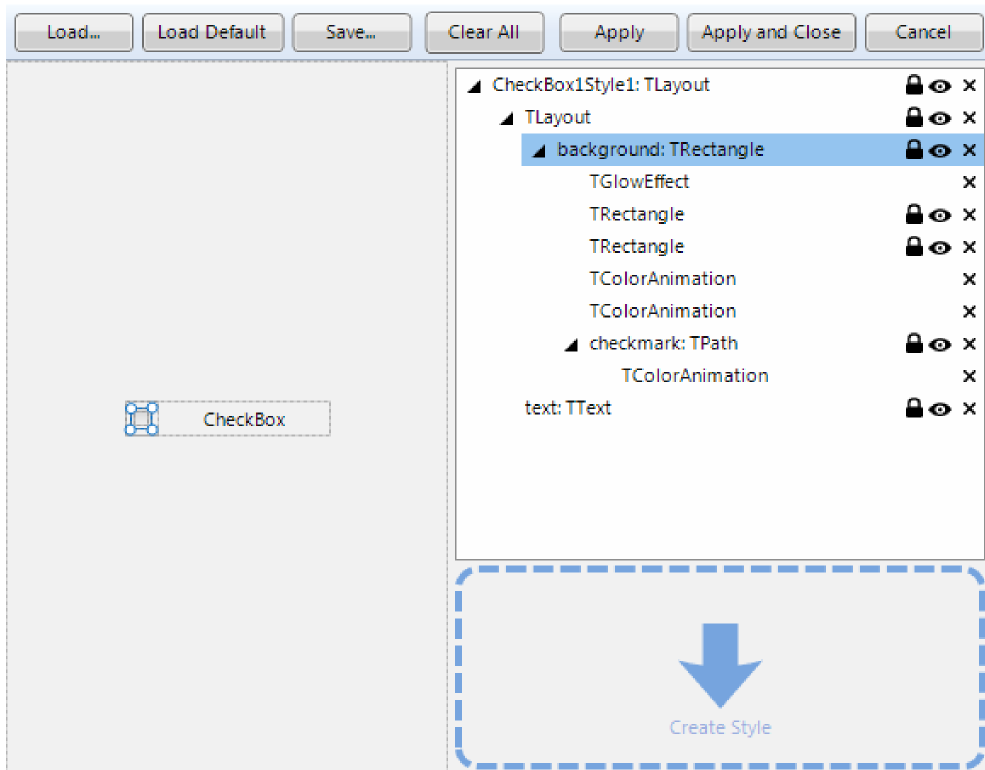


Se parte de un TLayout que sirve para colocar a la izquierda el botón en sí y a la derecha el título asociado. El texto se lleva a la ventana mediante un objeto TText. Por su parte el botón es otro TLayout que sirve para contener un TRectangle que actúa como fondo y, en su interior, dos TRectangle adicionales que dibujan el borde exterior e interior del control. La marca, que aparece cuando IsChecked tiene el valor True, se genera mediante un objeto TPath. Además se usan varios objetos adicionales que sirven para generar los efectos visuales cuando el ratón pasa sobre el control o cuando éste tiene el foco de entrada: TGlowEffect y TColorAnimation.

Cada uno de los objetos enumerados tiene su propio conjunto de propiedades que, a la postre, son las que establecen el color que tendrá el texto, el grosor de los bordes del botón, el tipo de marca que aparecerá cuando `IsChecked` sea `True`, etc. Todas ellas tienen un valor por defecto que podemos cambiar, ya sea editando el estilo general que se aplica a todos los controles del mismo tipo o definiendo un estilo a medida para un control específico.

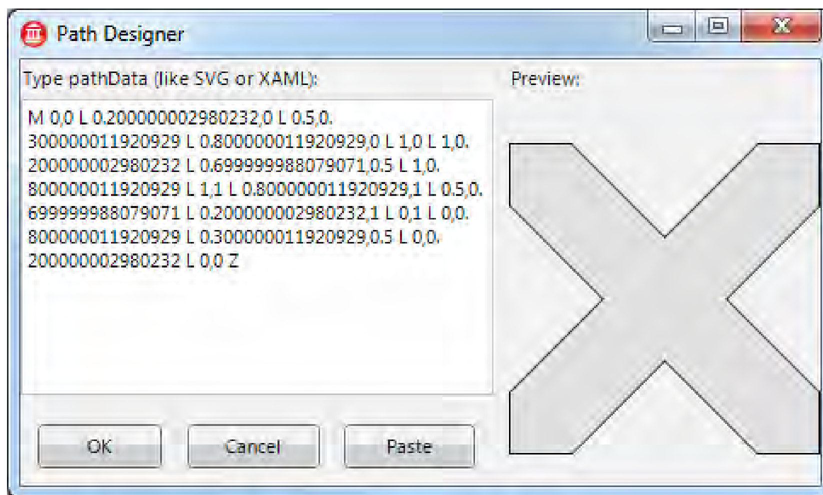
Si insertamos un `TCheckBox` en el formulario, abrimos su menú contextual y seleccionamos la opción Edit Custom Style accederemos al editor integrado de estilos de FireMonkey, presumiblemente para definir un estilo a medida para esta instancia del control (no afectaría a otros `TCheckBox` que pudiese haber en el formulario).

En la parte derecha, si desplegamos todos los nodos, encontraremos la jerarquía de elementos indicada anteriormente (véase la figura inferior). Activando cualquiera de ellos podremos editar en el Inspector de objetos sus propiedades.



Supongamos que queremos cambiar la apariencia de la marca que aparece dentro del recuadro del `TCheckBox` cuando el botón está marcado (propiedad `IsChecked` a `True`). Asumiendo que nos encontramos en el editor del estilo asociado al control, los pasos a seguir serían los indicados a continuación:

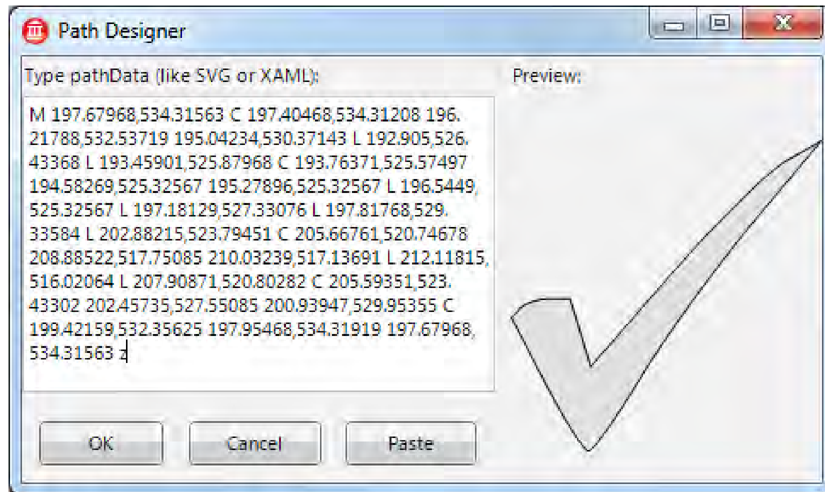
- Seleccionamos en el diseñador de estilos el objeto `TPath`[121], cuyo nombre es `checkmark`.
- Buscamos en el Inspector de objetos la propiedad `Data` y hacemos clic sobre el botón que aparece a la derecha del valor. También podríamos modificar directamente el valor de la propiedad, pero prescindiendo de la previsualización del resultado.
- Se abre una ventana como la inferior en la que podemos ver el código correspondiente al recorrido, en sintaxis SVG[122], que define la marca y a la derecha su aspecto.



121 En la página `Shapes` de la Paleta de herramientas podemos encontrar el componente `TPath` para incluirlo directamente en un formulario y mostrar gráficos vectoriales.

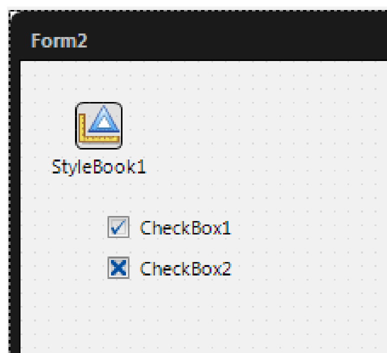
122 *Scalar Vector Graphics*. Es un estándar del W3C para la definición de gráficos vectoriales. Uno de los elementos de SVG son las rutas (*paths*), sucesiones de líneas rectas o curvas y desplazamientos que definen figuras, que se crean con un lenguaje basado en comandos sencillos como `M`: moverse a un punto, `L`: trazar una línea desde la posición actual hasta el punto indicado, `C`: dibujar una curva, o `Z`: cerrar la ruta. Puedes encontrar la referencia de este lenguaje en `http://www.w3.org/TR/SVG/paths.html`.

- Eliminamos el contenido del recuadro de texto e introducimos las nuevas instrucciones para dibujar la marca. Un ejemplo podría ser el que aparece en la imagen inferior, correspondiente a un ejemplo procedente de Wikipedia.



- Hacemos clic en el botón OK para cerrar esta ventana y volver al editor de estilos. En él usaremos el botón Apply and close para guardar el estilo y volver al formulario.

En el formulario aparecerá un componente `TStyleBook` en el que se ha almacenado el nuevo estilo. Si damos el valor `True` a la propiedad `IsChecked` del `TCheckBox` podremos ver el resultado. El nuevo estilo no se aplicará a otros controles del mismo tipo que insertemos en el formulario tal y como puede verse en la imagen inferior.



Reutilizar un estilo en otros controles

En el panel derecho del editor de estilos FMX, como título del nodo raíz, aparece el nombre que se ha asignado al estilo asociado al TCheckBox. El nombre siempre es combinación del nombre del control y una numeración consecutiva tras la palabra Style, de forma que es posible contar con varios estilos distintos para el mismo tipo de control. En este caso, por tanto, el nombre el estilo es CheckBox1Style1.

Tras cerrar el editor de estilos podemos seleccionar el TCheckBox y, en el Inspector de objetos, comprobar que su propiedad StyleLookup hace referencia al anterior nombre de estilo. Esa propiedad se hereda de TStyledControl y su modificación desencadena la ejecución del método ApplyStyleLookup de esa misma clase, encargada de localizar el estilo indicado y aplicarlo al control.

> NOTA
>
> Cuando la propiedad StyleLookup de un control no tiene contenido se aplica un estilo por defecto. Éste es dependiente del sistema operativo en que esté ejecutándose la aplicación, de forma que la apariencia siempre es lo más cercana posible al *look&feel* nativo.

Para reutilizar el estilo que hemos creado, por tanto, bastará con asignar a la propiedad StyleLookup del control objetivo el nombre del estilo. El efecto será inmediato, tanto si hacemos el cambio en la fase de diseño como si optamos por hacerlo en ejecución con una sentencia como la siguiente:

```
CheckBox2.StyleLookup := 'CheckBox1Style1';
```

Aunque pueda parecernos extraño, al asociar un estilo a un control no se realiza verificación alguna relativa a la compatibilidad entre el componente origen de ese estilo y su nuevo destino. Es posible aplicar el estilo de un control a otro con una funcionalidad completamente diferente, algo que cambiará solo su apariencia, no dicha funcionalidad, y que como es lógico confundirá al usuario. Podemos probarlo agregando al formulario un TEdit, asignando a su propiedad StyleLoopkup el valor CheckBox1Style1 y después ejecutando el programa. El TEdit muestra la apariencia de un TCheckBox, pero sigue comportándose como una caja de texto: podemos hacer clic sobre ella y escribir. A pesar de no tener sentido esto nos demuestra la clara separación entre interfaz y funcionalidad en FMX.

Para reutilizar el estilo en otros proyectos que desarrollemos con posterioridad tendremos que guardarlo en un archivo de estilos FMX, recuperándolo cuando lo precisemos. Los botones Save y Load del editor de estilos[123] facilitan esas operaciones.

Los estilos se almacenan en archivos con la extensión `.style` como un texto sin formato, por lo que pueden ser abiertos con cualquier editor básico. Su estructura es similar a la mostrada a continuación, correspondiente a un fragmento del estilo creado anteriormente:

```
object _1: TLayout
  Align = alClient
  Position.Point = '(0,33)'
  Width = 1057.000000000000000000
  Height = 979.000000000000000000
  object TLayout
    StyleName = 'CheckBox1Style1'
    Position.Point = '(468,479)'
    Width = 121.000000000000000000
    Height = 21.000000000000000000
    object TLayout
      Align = alLeft
      Width = 20.000000000000000000
      Height = 21.000000000000000000
      object TRectangle
        StyleName = 'background'
        Align = alCenter
...
      end
    end
    object TText
      StyleName = 'text'
      Align = alClient
      Position.Point = '(21,2)'
      Locked = True
      Width = 99.000000000000000000
      Height = 19.000000000000000000
      Padding.Rect = '(1,2,1,0)'
      HitTest = False
      Text = 'CheckBox'
    end
  end
end
```

123 Podemos abrir el editor de estilos en cualquier momento simplemente haciendo doble clic sobre el componente `TStyleBook` que se insertó en el formulario al crear el estilo.

Estilos por defecto

Si queremos asignar un mismo estilo a todos los controles de un cierto tipo, por ejemplo sustituyendo la marca de todos los TCheckBox por nuestro nuevo diseño, nos resultará mucho más cómodo modificar el estilo por defecto que se aplica a estos componentes cuando la propiedad StyleLookup está vacía. De esta forma no tendremos que asignarle el nombre del estilo a todas las copias del control que usemos en nuestro proyecto.

Suponiendo que no tuviésemos ya definido el estilo que queremos usar, partiendo de cero, los pasos a seguir son:

- Introducimos en el formulario la primera copia del control cuyo estilo por defecto queremos alterar, en este caso un TCheckBox.
- Abrimos su menú contextual y elegimos la opción Edit Default Style para abrir el editor de estilos, en el que aparecerá un nodo de tipo TLayout sin nombre.
- Seleccionamos ese nodo raíz y, en el Inspector de objetos, cambiamos su propiedad StyleName, que está inicialmente vacía, estableciendo como nombre el del tipo de control sin el prefijo T y con el sufijo style. En este caso sería checkboxstyle.
- Efectuamos los ajustes que queramos en el estilo y finalmente lo cerramos y aplicamos, como hicimos antes.

A partir de este momento todos los controles TCheckBox que agreguemos al formulario tendrán el nuevo estilo, sin necesidad de asignar ningún valor a la propiedad StyleLookup.

> NOTA
>
> La clase TStyledControl, de la que derivan todos los componentes FMX a los que puede aplicarse un estilo, cuenta con un método llamado GetStyleObject encargado de obtener el estilo que corresponda a cada control del formulario. Cuando la propiedad StyleLookup de un control está vacía se concatena su nombre de clase, obtenido de la propiedad ClassName, se le añade la cadena 'style' y se elimina la T inicial. Cualquier estilo que se ajuste a esa nomenclatura actuará como estilo por defecto.

Cómo definir un estilo nuevo

La mejor vía para entender cómo funciona un estilo FMX es sin duda crear uno nuevo partiendo desde cero. Es lo que vamos a hacer en este apartado: definir un estilo nuevo pensado para el control `TCheckBox`. El objetivo es que el control aparezca como un rectángulo con esquinas redondeadas, mostrando el texto en la parte inferior y sobre él un círculo que estará vacío o relleno dependiendo del estado de la propiedad `IsChecked`.

Tras abrir el editor de estilos, ya sea con un doble clic sobre el `TStyleBook` que haya en el formulario o una de las opciones indicadas en puntos previos, buscamos en la Paleta de herramientas el componente `TLayout` (se encuentra en la página Layouts) y lo arrastramos hasta el recuadro con borde punteado azul[124] y título Create Style. Veremos aparecer el componente en la lista de estilos y en el Inspector de objetos tenemos acceso a sus propiedades. Cambiaremos el contenido de `StyleName` asignando al nuevo estilo el nombre `cbstyle`.

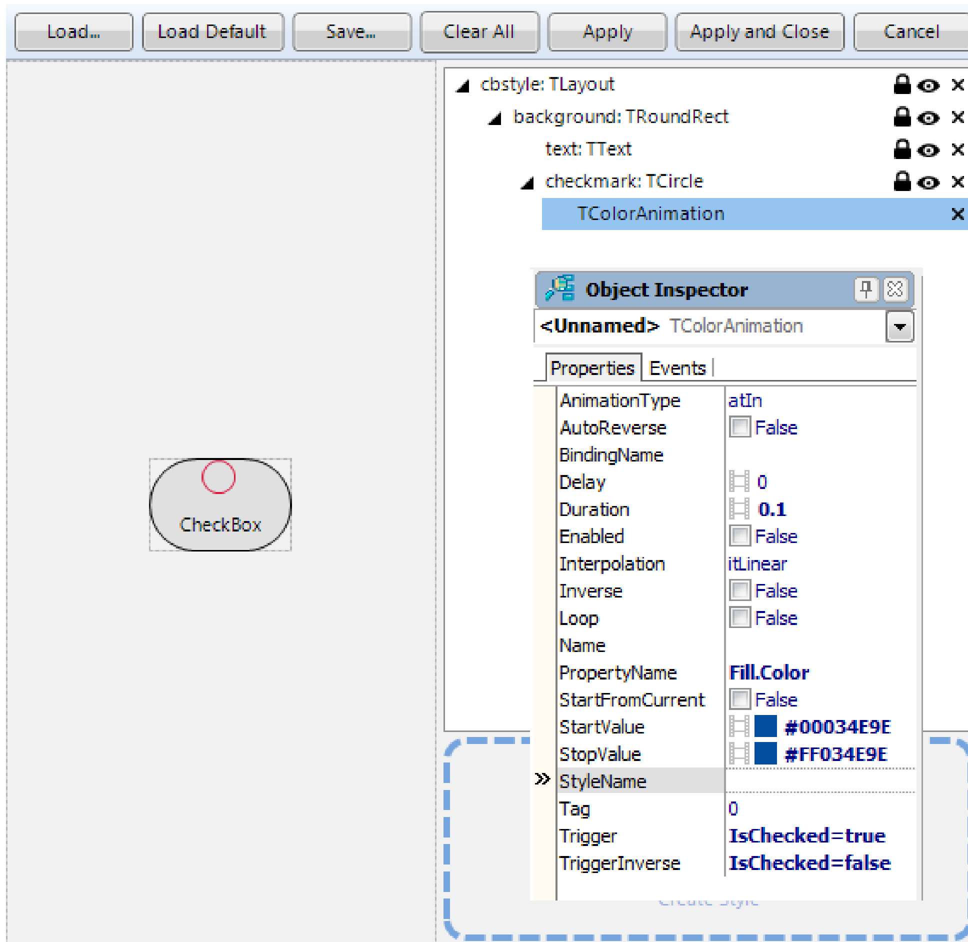
Una vez creado el contenedor del nuevo estilo, que será ese componente `TLayout`, introduciremos los siguientes elementos:

- Tomamos un componente `TRoundRect` de la página Shapes de la Paleta de herramientas y lo arrastramos hasta soltarlo sobre la lista de estilos, concretamente sobre el `TLayout` para que se convierta en hijo de éste. Modificamos la propiedad `Align` del `TRoundRect` para que ocupe todo el espacio disponible: valor `alClient`.
- Insertamos dentro del `TRoundRect` un componente `TText`, también está en la página Shapes, dando a su propiedad `Align` el valor `alBottom`. Para que el contenido de la propiedad `Text` del control al que se asocie este estilo aparezca dentro del `TText` asignaremos a su propiedad `StyleName` el nombre `text`.
- Agregamos un `TCircle`, también como hijo del `TRoundRect`, dando el valor `alCenter` a la propiedad `Align` y ajustando su tamaño para conseguir la apariencia que queramos. Con la propiedad `Stroke` configuramos el borde. La propiedad `Fill.Kind` ha de quedar como `bkSolid`, a fin de que el relleno sea un color uniforme, y el color inicial, almacenado en `Fill.Color`, no es importante.

---

124 El método para iniciar la creación de un nuevo estilo siembre es éste, con independencia del contenido que tenga la lista de estilos en cada momento.

- Finalmente añadimos un TColorAnimation como hijo del TCircle. La función de este componente será cambiar el contenido de la propiedad Fill.Color de su padre, que es el círculo, según el estado que tenga la propiedad IsChecked del TCheckBox. La propiedad a cambiar se asigna a la propiedad PropertyName, el desencadenador del cambio a la propiedad Trigger y el desencadenador de ese mismo cambio en sentido inverso a la propiedad TriggerInverse. El color en un caso y en otro se asignará a las propiedades StartValue y StopValue. En la imagen inferior puede ver la estructura general del nuevo estilo y el detalle de la configuración del TColorAnimation. En el margen izquierdo está la vista previa de cómo quedaría el control.

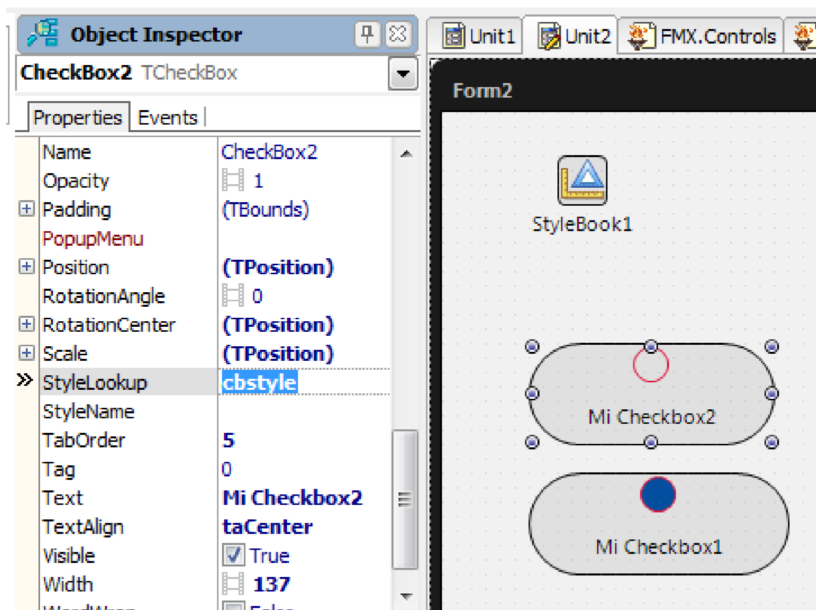


Una vez terminada la definición del estilo hacemos clic en el botón Apply and Close. Antes podemos guardarlo en un archivo para facilitar su reutilización en otros proyectos. La definición completa del estilo es la mostrada a continuación:

```
object TLayout
  object TLayout
    StyleName = 'cbstyle'
    Position.Point = '(486,462)'
    Width = 84.000000000000000000
    Height = 55.000000000000000000
    object TRoundRect
      StyleName = 'background'
      Align = alClient
      Locked = True
      Width = 84.000000000000000000
      Height = 55.000000000000000000
      HitTest = False
      object TText
        StyleName = 'text'
        Align = alBottom
        Position.Point = '(0,23)'
        Locked = True
        Width = 84.000000000000000000
        Height = 32.000000000000000000
        Text = 'CheckBox'
      end
      object TCircle
        StyleName = 'checkmark'
        Align = alCenter
        Position.Point = '(25,1)'
        Locked = True
        Width = 33.000000000000000000
        Height = 20.000000000000000000
        HitTest = False
        Fill.Color = claNull
        Stroke.Color = claCrimson
        object TColorAnimation
          Duration = 0.100000001490116100
          Trigger = 'IsChecked=true'
          TriggerInverse = 'IsChecked=false'
          StartValue = x00034E9E
          StopValue = xFF034E9E
          PropertyName = 'Fill.Color'
        end
      end
    end
  end
end
```

Para que el control responda a los clic del usuario, en ejecución, se ha dado el valor `False` a la propiedad `HitTest` tanto del `TRoundRect` como del `TCircle`, pero no del `TText`. El estado, por tanto, no cambiará si se hace clic sobre el texto, pero sí al pulsar sobre cualquier otro punto de la superficie del área visible del control.

Terminada la definición del nuevo estilo, no tenemos más que asignar su nombre a la propiedad `StyleLookup` de los `TCheckBox` en los que deseemos usarlo. En la imagen inferior pueden verse dos controles con el nuevo aspecto, uno con la propiedad `IsChecked` a `False` y otro a `True`. El título de los botones es el asignado a la propiedad `Text` del `TCheckBox`, si bien su visualización queda en manos del control `TText`.

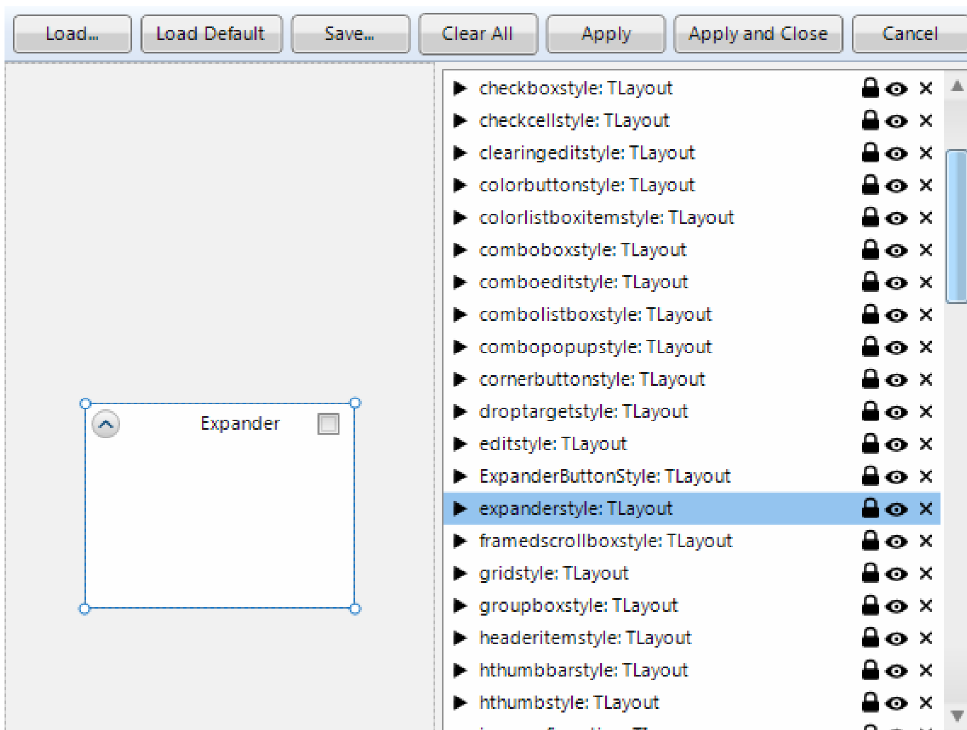


> TRUCO
>
> Si queremos que todos los `TCheckBox` tomen el nuevo aspecto no tenemos más que cambiar el nombre del nuevo estilo, llamándolo `checkboxstyle` en lugar de `cbstyle`. Todos los controles `TCheckBox` que no tengan contenido en su propiedad `StyleLookup` usarán el nuevo estilo.

Estilos predefinidos

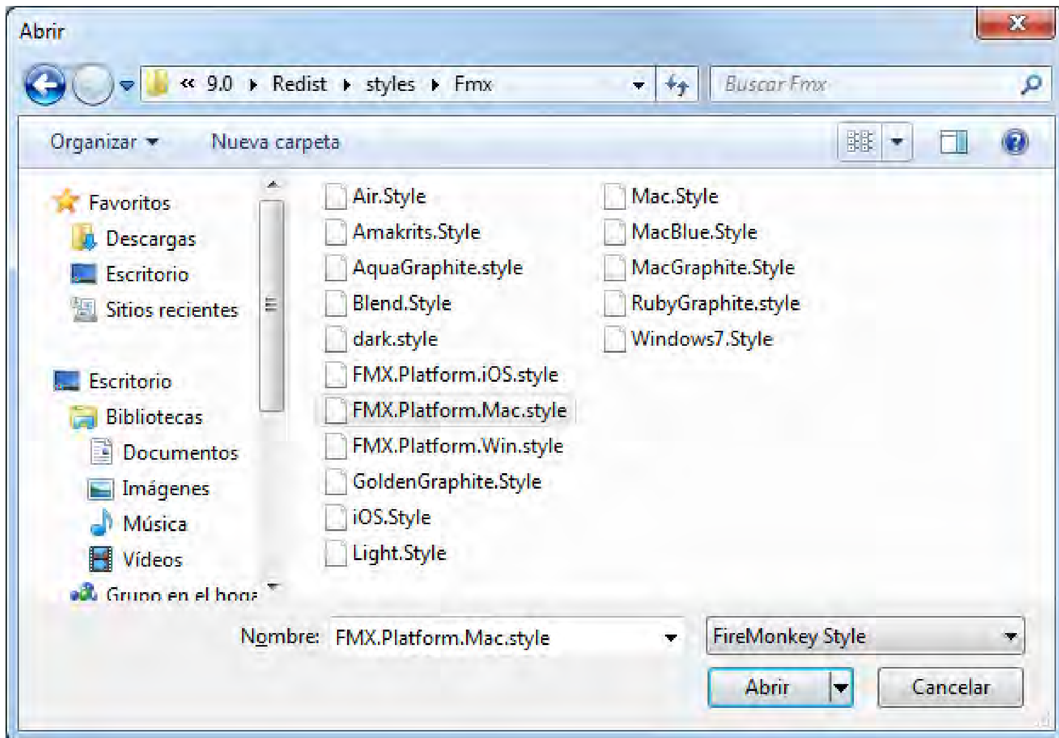
Raramente nos encontraremos en la necesidad de definir estilos nuevos para cada uno de los componentes que podemos usar al diseñar una interfaz de usuario. Si únicamente queremos introducir ligeros cambios en algunos de los estilos que se usan por defecto, nos será mucho más fácil recuperar la configuración predefinida, mediante el botón Load Default del editor de estilos, y después trabajar sobre cada estilo.

La lista de estilos predefinidos, como puede apreciarse en la imagen inferir, es muy extensa ya que se define un estilo para cada control existente en la FMX y, en ocasiones, para partes de controles.



Dado que éstos son los estilos por defecto, cualquier cambio que introduzcamos en ellos afectará a todas las copias de cada control que existan en el formulario salvo aquellos que, de manera explícita, tengan aplicado un estilo distinto. Examinar todos estos estilos nos servirá además para aprender mucho sobre cómo crear otros alternativos.

Aparte del estilo por defecto, que da a los controles FMX la apariencia estándar de la interfaz de Windows, Delphi XE2 incluye varios estilos predefinidos más. Los encontraremos en la subcarpeta `Redist\styles\Fmx` de la carpeta donde esté instalado el producto (véase la imagen inferior). Por una parte tenemos los estilos que corresponden a las diferentes plataformas: `FMX.Platform.XXX`, a los que se agregan otros como `Air`, `Blend` o `dark`.



Usando el botón `Load` del editor de estilos podemos recuperar la definición de cualquiera de ellos y aplicarlos al formulario actual.

> NOTA
>
> Tomando como punto de partida una copia de uno de estos estilos, basta con hacer una copia del archivo original con otro nombre, podemos diseñar variaciones personalizadas y crear un estilo global nuevo.

Cambiar de estilo en ejecución

En un punto previo se mostró cómo era posible cambiar el estilo de un control TCheckBox, aunque el método sería aplicable a cualquier otro componente, mediante una simple sentencia de asignación que modificaba su propiedad StyleLookup. Dicha sentencia puede ser ejecutada, por ejemplo, ante la elección de una opción de menú por parte del usuario, de forma que el estilo se adapte a las preferencias de cada persona. ¿Cómo podemos, sin embargo, cambiar el estilo completo de una ventana sin hacerlo componente a componente? ¿Y si quisiésemos cambiar el estilo de todas las ventanas de la aplicación?

Ya hemos visto anteriormente que un componente TStyleBook puede alojar tantos estilos como se precise, identificado cada uno de ellos con un nombre distinto. No hay limitación en cuanto al número de componentes de ese tipo que pueden existir en un formulario, por lo que podríamos agregar varios TStyleBook y cargar en cada uno de ellos un estilo diferente. Para cambiar de un estilo a otro no habría más que modificar la propiedad StyleBook del formulario en cuestión.

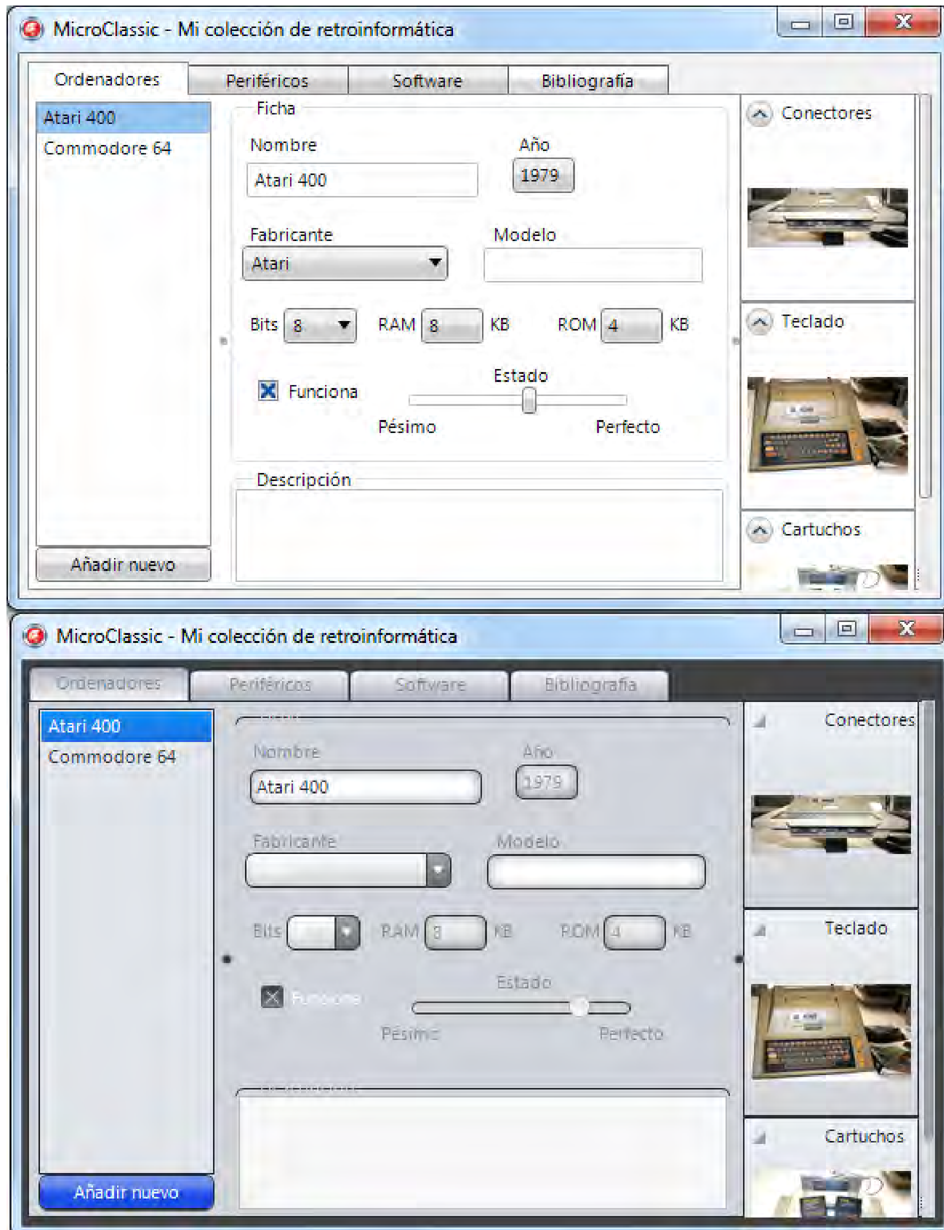
NOTA

Para restablecer el estilo por defecto, que siempre corresponde al aspecto nativo del sistema operativo en que está ejecutándose la aplicación, tenemos que eliminar el contenido de la propiedad StyleBook. Para ello le asignaremos el valor nil.

El único problema que plantea este método es que el cambio de estilo afectará exclusivamente al formulario cuya propiedad StyleBook se ha cambiado, pero no al resto de formularios con que pudiera contar la aplicación. Esto daría lugar a que cada ventana mostrase un estilo distinto algo que, con toda seguridad, no es deseable. Lo habitual es que el estilo sea uniforme en todas las ventanas de un mismo proyecto.

Podemos comprobar este problema tomando cualquier formulario de ejemplo de capítulos previos, como puede ser el del proyecto MicroClassic, e insertando en él un TStyleBook con un estilo que no sea el nativo. A continuación asociamos al evento OnClick de uno de sus botones las sentencias para crear una segunda copia del mismo formulario y cambiar su estilo con StyleBook := StyleBook1. El resultado será

similar al mostrado en la imagen inferior, en la que puede verse que el estilo del primer formulario (parte superior) no se ve afectado por el cambio en el segundo.



La solución que ofrece FireMonkey para solucionar este problema la encontramos en la propiedad `StyleFileName` del objeto `Application`. Si dicha propiedad no tiene contenido, es de tipo `String`, se aplicará el estilo por defecto. En cualquier momento podemos asignarle el nombre de un archivo de estilo, incluyendo la ruta en la que se encuentra si fuese necesario, provocando una actualización que afectará a todas las ventanas del proyecto. Podríamos, por ejemplo, asociar la siguiente sentencia al evento `OnClick` de un botón o una opción de menú:

```
Application.StyleFileName := 'FMX.Platform.Mac.style';
```

El inconveniente de este método estriba en la necesidad de distribuir junto a la aplicación los archivos externos con los estilos que se ofrecen, ya que la propiedad `StyleFileName` únicamente acepta un nombre de archivo y no la referencia a un `TStyleBook`. Como ya sabemos esos archivos pueden ser modificados fácilmente con cualquier editor, por lo que la aplicación podría ver afectada su interfaz de usuario de una forma indeseada.

Una alternativa al anterior método consiste en incorporar en un formulario tantos `TStyleBook` como estilos se deseen ofrecer, configurándolos en la fase de diseño. Toda la información relativa a los estilos quedará almacenada en el propio ejecutable, como un recurso más. Cuando se desee cambiar de un estilo a otro habría que cambiar la propiedad `StyleBook` de todos los formularios existentes en el proyecto. Lo deseable es que el método pueda aplicarse a cualquier proyecto, sin que importe los formularios con que cuente, en lugar de escribir un código específico para cada caso.

La gestión del estilo visual en FireMonkey se lleva a cabo mediante una serie de métodos y propiedades que forman parte de la interfaz `IScene`[125]. Uno de los miembros de dicha interfaz es la propiedad `StyleBook`, que llega a la clase `TForm` a través de su ascendiente `TCustomForm`, encargada de implementar los miembros de `IScene`. Lo interesante es que FireMonkey mantiene una lista de todos los objetos de la aplicación que implementan `IScene`, en una variable llamada `SceneList`, algo que facilita enormemente nuestra tarea. Solamente hemos de recorrer dicha lista y cambiar el contenido de la propiedad `StyleBook` de cada uno de sus elementos, asignándole la referencia al estilo que se desee activar. También podemos devolver el estilo por defecto dando a dicha propiedad el valor `nil`, tal y como se indicó anteriormente.

---

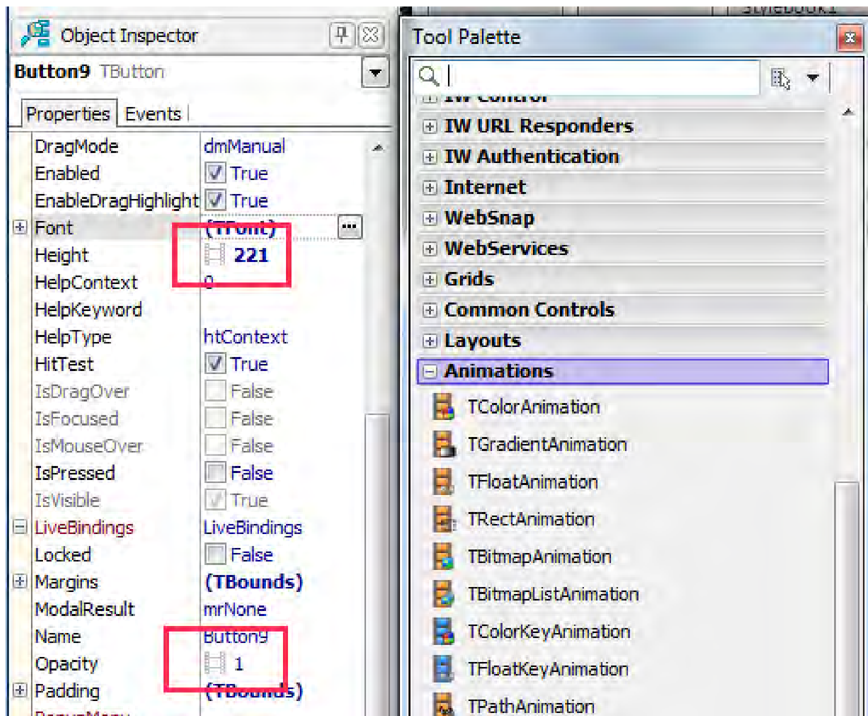
125 Ésta y otras interfaces relativas a FireMonkey están definidas en el módulo `FMX.Types` que es necesario agregar a la cláusula `uses` de nuestro código.

El siguiente fragmento de código muestra cómo se cambiaría el estilo de toda la aplicación estableciendo el que se encuentra almacenado en el componente StyleBook1. En lugar de usar un TStyleBook concreto éste podría recibirse como parámetro, de forma que un mismo método se encargase de cambiar el estilo a uno de los disponibles ante las acciones del usuario.

```
for i := 0 to SceneList.Count - 1 do
    IScene(SceneList[i]).StyleBook := StyleBook1;
```

Animaciones y transformaciones

Si examinamos las propiedades de cualquier componente FMX veremos que muchas de ellos muestran junto a su valor un icono en forma de película (véase la imagen inferior). Esto indica que es posible definir una animación asociada a la propiedad, un componente de la página Animations.



Además todos los controles exponen propiedades que, como Scale o RotationAngle, definen transformaciones a aplicar a la hora de dibujar el control en su contenedor, alterando sus dimensiones, posición y orientación. Es posible combinar animaciones y transformaciones para diseñar interfaces con una cierta componente dinámica.

Tipos de animaciones

Las animaciones en FireMonkey se generan cambiando el valor de una cierta propiedad a lo largo del tiempo. El control de la animación recae en un componente[126] que será incluido como hijo de aquél cuya propiedad se quiere animar. Podemos agrupar las animaciones atendiendo a los siguientes criterios:

- **El tipo de la propiedad a animar**: Hay propiedades que son números en punto flotante, otras representan colores o gradientes de color, otras coordenadas, etc. Según el tipo de la propiedad habrá que recurrir a un componente de animación u otro.
- **El número de intervalos de animación**: La mayoría de los componentes de la página Animations generan una interpolación entre dos valores del tipo que corresponda y que actúan como extremos: valor de inicio y de fin. No obstante también es posible definir animaciones con una lista de valores clave, creando múltiples intervalos de interpolación entre ellos.

Del nombre de cada componente podemos deducir tanto el tipo de dato al que se aplica como el número de intervalos de animación. El patrón usado es Ttipo[Key]Animation, con las siguientes partes:

- tipo: Indica el tipo de propiedad a que se aplica. Puede ser Float, Rect, Color, Gradient, Bitmap o Path, según que la propiedad contenga un número en punto flotante, unas coordenadas (posición y dimensiones), un color, un degradado de color, un mapa de bits (una imagen) o una posición.

---

126 Todos los componentes que generan animaciones están derivados de una clase común: TAnimation. En ésta se definen las propiedades y métodos generales a cualquier animación: AnimationType, Duration, Inverse, Start, Stop, etc.

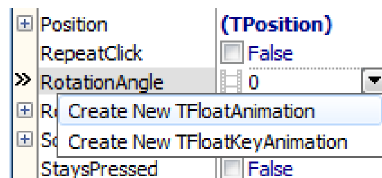
- `Key`: Si aparece esta palabra delante de `Animation` indica que la interpolación se efectúa entre varios valores clave, no solamente dos extremos.

Una excepción a esta regla es `TPathAnimation` que, a pesar de no contener la palabra `Key`, define una animación entre múltiples puntos. El resto de tipos de animación sí se ajustan al patrón descrito: `TFloatAnimation` y `TFloatKeyAnimation`, `TColorAnimation` y `TColorKeyAnimation`, `TRectAnimation`, `TGradientAnimation` y `TBitmapAnimation`.

> NOTA
>
> Aparte de los mencionados hay un componente más de animación, llamado `TBitmapListAnimation`, que tiene poco en común con los anteriores. Su objetivo es generar una secuencia de diapositivas a partir de una lista de imágenes, no modificar una propiedad de un control a lo largo del tiempo.

Configuración básica de una animación

El primer paso para configurar una animación es insertar el componente adecuado, algo que podemos hacer de dos formas diferentes. Si tenemos en el formulario el control cuya propiedad queremos animar y ésta muestra el icono en forma de película podemos, como se muestra en la imagen inferior, abrir la lista desplegable asociada al valor y elegir el tipo de animación a introducir. El tipo de ésta será el que corresponda a la propiedad y el enlace entre ambos componentes se creará automáticamente.



Como vía alternativa podemos tomar el componente de animación de la Paleta de herramientas e insertarlo manualmente. En este caso habremos de poner especial atención, ya que hemos de agregarlo como hijo del control que pretendemos animar. Además tendremos que enlazar el componente de

animación con la propiedad que se quiere modificar en el control a animar, usando para ello la propiedad `PropertyName` del primero. El contenido de ésta será el nombre de una propiedad o, si es necesario, el de un miembro de una propiedad si ésta es compleja. Para desplazar horizontalmente un control, por ejemplo, asignaríamos el valor `Position.X` a la propiedad `PropertyName` de un `TFloatAnimation`.

Establecido el vínculo entre control a animar y componente de animación, la configuración de ésta dependen fundamentalmente de las siguientes propiedades:

- `StartValue` y `StopValue`: Contendrán los valores extremos[127] entre los que se interpolará, asignando a la propiedad objetivo el valor obtenido por la función de interpolación en cada unidad de tiempo.
- `Delay` y `Duration`: Ambas vienen expresadas en segundos. La primera indica el tiempo de espera antes de comenzar la animación una vez que ésta se ponga en marcha. La segunda establece el tiempo que durará la animación, tiempo durante el cual se irá recalculando la función de interpolación y obteniendo nuevos valores a asignar a la propiedad objetivo. Mientras esto sucede el componente de animación genera periódicamente el evento `OnProcess`.
- `Interpolation`: Selecciona el tipo de función de interpolación a usar para la animación. La función afecta a la forma en que se distribuyen los cambios a lo largo del tiempo indicado por `Duration`, pudiendo optar por una evolución uniforme que es la usada por defecto: `itLinear`, o bien cambiarla por otra de la decena de posibilidades existentes. Un interpolación cuadrática (`itQuadratic`), por ejemplo, hará que el cambio en el valor de la propiedad sea menor al principio y mayor al final, efecto que se acentúa con las funciones `itCubic`, `itQuartic` e `itQuintic`. También hay funciones que provocan una oscilación en el grado de la velocidad con que se cambia el valor, como `itSinusoidal`, oscilación que incluso puede ir hacia atrás como ocurre con `itBounce`.
- `AnimationType`: Controla la forma en que se aplica la función de interpolación para ir cambiando el valor de la propiedad objetivo. No

127 En los componentes de animación con varios intervalos estas propiedades se verán sustituidas por una sola llamada `Keys`, una colección de elementos `TFloatKeyAnimation` conteniendo cada uno de ellos una clave y un valor .

tiene efecto si la función es lineal, pero sí en el resto de los casos. El valor por defecto es atIn, de forma que la función se aplica en su secuencia natural desde el principio de la animación. Para itExponential, por ejemplo, el cambio sería muy lento al principio y acelerado al final. Si lo cambiamos por atOut la función se aplicará en sentido inverso, acelerando los cambios al inicio y ralentizándolos al final. Con el valor atInOut se usa la función original hasta la mitad del tiempo y, de ese punto en adelante, se aplica la misma función pero invertida. Esto provocaría que el cambio fuese lento en los extremos y rápido en la parte central.

- Loop y AutoReverse: Si damos el valor True a Loop la animación se repetirá, volviendo al valor de partida al inicio de cada ciclo salvo que se dé el valor True a AutoReverse. En este caso se actuará como si los valores de inicio y fin se invirtiesen.

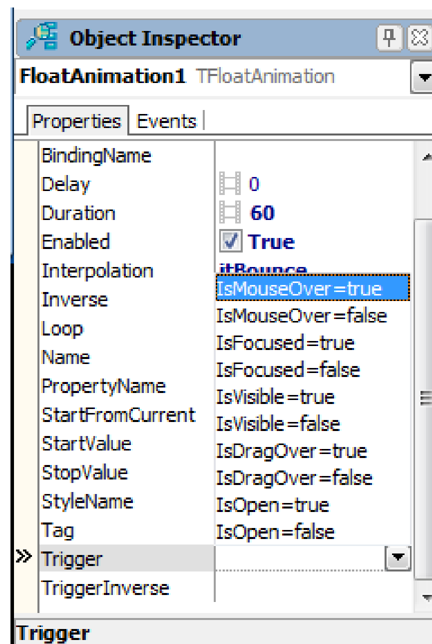
En la mayoría de las ocasiones nos limitaremos a configurar la animación con las propiedades StartValue, StopValue y Duration, pero como puede verse podemos obtener un control mucho mayor sobre la forma en que se llevará a cabo la modificación del valor de la propiedad objetivo a lo largo del tiempo.

Control de la animación

Definidos los parámetros de la animación llega el momento de decidir cuándo ha de ponerse en marcha y cuándo debe detenerse. Tenemos fundamentalmente tres alternativas a nuestra disposición:

- **Inicio automático**: Si queremos que la animación se ponga en marcha de manera inmediata, en cuanto se haga visible la ventana en que está el componente, no tenemos más que dar el valor True a la propiedad Enabled en la fase de diseño. La animación se detendrá finalizado el tiempo indicado por Duration a menos que Loop también sea True, en cuyo caso se estará repitiendo de manera indefinida.
- **Inicio manual**: Usando los métodos Start y Stop podemos iniciar y detener la animación cuando convenga. También es posible detenerla en el valor actual, asignando el valor True a Pause, para reiniciarla después devolviéndole el valor False a esa misma propiedad.

- **Inicio por desencadenador**: La animación se iniciará de manera automática ante un cierto suceso desencadenante. Éste ha de introducirse en la propiedad Trigger y suele tomar la forma propiedad=valor, indicando la propiedad del control a animar que ha de supervisarse. Trigger tiene asociada una lista desplegable (véase la figura inferior) con los desencadenadores disponibles, todos ellos de tipo booleano. También podemos usar la propiedad TriggerInverse para indicar el evento que desencadenaría la ejecución de la animación en sentido inverso.



> TRUCO
>
> Las propiedades Trigger y TriggerInverse pueden contener más de una condición de tipo booleano, separadas unas de otras con punto y coma. Por ejemplo:
>
> IsMouseOver=true; IsFocused=false.
>
> La animación se desencadena solamente si se cumplen todas las condiciones, es decir, el punto y coma actúa como el operador And.

Rotación y escalado

Entre las propiedades a las que puede aplicarse un proceso de animación mediante los componentes mencionados en los apartados previos, concretamente con TFloatAnimation y TFloatKeyAnimation, hay dos especialmente interesantes: Scale y RotationAngle. Con ellas podemos alterar tanto las dimensiones de un componente como su orientación en el plano.

La propiedad Scale es de tipo TPosition, por lo que cuenta con dos miembros: X e Y. En realidad no indica una posición, sino un factor de escala en el eje X y el eje Y, respectivamente, expresado como un número en punto flotante. Un factor inferior a 1.0 reduciría el ancho o alto del componente, mientras que un valor superior a 1.0 lo extendería en el eje correspondiente. El valor por defecto de ambos miembros es 1.0 que corresponde a una escala del 100%.

Con la propiedad RotationAngle se establece el ángulo de rotación con el que se dibujará el control, siendo por defecto 0. El valor de esta propiedad habrá de estar entre 0 y 359, para valores superiores siempre se obtendrá el resto de dividir entre 360. También es posible aplicar ángulos negativos[128].

Al realizar la rotación de un objeto cualquiera hay que establecer el punto que se tomará como referencia del giro. Si actuamos sobre un rectángulo, por ejemplo, el giro será distinto si el punto de referencia es la esquina inferior izquierda que si tomamos el punto central en ambos ejes. La propiedad RotationCenter es la encargada de situar la posición de referencia para la rotación en un el intervalo [0, 1] en ambos ejes. Por defecto los miembros X e Y de esta propiedad tienen el valor 0.5, por lo que la rotación se hará respecto al centro del control al que se aplica.

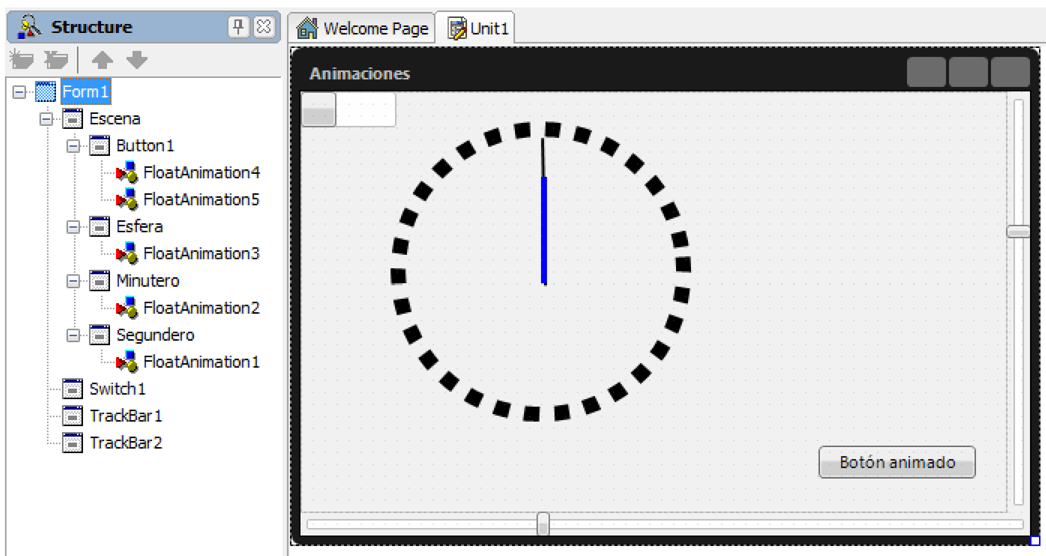
La tercera transformación clásica que puede aplicarse un objeto, la traslación, es mucho más sencilla: no hay más que modificar la propiedad Position del control sumando o restando el valor que se desee a sus miembros X e Y. De esta forma se conseguirá un desplazamiento del control en el eje correspondiente, hacia abajo a la derecha si se ha sumado o hacia arriba o la izquierda si se ha restado.

---

128 Una rotación positiva se efectúa en el sentido de las agujas del reloj, mientras que una negativa va en sentido contrario al de las agujas del reloj pero manteniendo la magnitud del giro.

Animaciones y transformaciones en la práctica

Para terminar este punto veamos con un sencillo ejemplo cómo aplicar algunas animaciones y transformaciones a componentes FMX. Comenzaremos insertando en un formulario un TLayout y, en su interior, un TCircle, dos TLine y un TButton. Fuera del TLayout añadiremos dos TTrackBar y un TSwitch. El aspecto del formulario en el diseñador es el mostrado en la imagen inferior. En ella se aprecia que cada uno de los controles que hay en el TLayout tiene como hijo uno o más componentes TFloatAnimation.



El TCircle actuará como la esfera de un reloj analógico. Modificaremos su propiedad Fill para que no tenga relleno y las propiedades StrokeDash y StrokeThickness para darle la apariencia de la imagen anterior. Las propiedades Width y Height tendrán el mismo valor, de forma que obtengamos un cuadrado cuyas dimensiones tomaremos como referencia para colocar dentro los dos controles TLine.

Colocaremos los dos controles TLine, uno será más largo que otro (su propiedad Height tendrá un valor mayor), de forma que su parte inferior quede exactamente en el centro del TCircle. Para ello no hay más que

tomar las dimensiones de éste, dividir por la mitad y aplicar el desplazamiento que corresponda a la posición en que se encuentra. El `TLine` más largo hará las veces de segundero y el más corto las de minutero. Para diferenciarlos haremos que este último, que será el más corto, tenga un color distinto y también sea más grueso, modificando para ello las propiedades `Stroke.Color` y `StrokeThickness`.

Teniendo seleccionado el `TLine` que funciona como minutero abriremos en el Inspector de objetos la lista asociada a su propiedad `RotationAngle` para agregar una nueva animación. De ésta cambiaremos las propiedades `StopValue`, que será `359` para realizar un giro completo; `Duration`, al que asignaremos el valor `60` para que la animación dure 60 segundos; `Loop`, que activaremos para que la animación se repita continuamente, y `Enabled` que también quedará a `True` para iniciar la animación en cuanto se ejecute el programa.

A continuación seleccionamos el segundo `TLine` y repetimos la operación. La configuración de la animación será exactamente la misma salvo el valor de `Duration` que en este caso será 600, ya que el minutero ha de recorrer la esfera completa en una hora.

Para ambos `TLine` el centro de rotación habrá que desplazarlo hasta el extremo de la línea que está situado en el centro de la esfera, porque de lo contrario el giro se realizaría como si cada línea fuese el aspa de un ventilador en lugar de la manecilla de un reloj. Por tanto modificaremos la propiedad `RotationCenter`, dando el valor `1` a los dos miembros que la componen.

> NOTA
>
> En este momento los controles que harán las veces de cronómetro, contando minutos y segundos, ya están completamente configurados. Si ejecutamos el programa podremos ver su funcionamiento, y no hemos escrito ni una sola línea de código para conseguir este resultado.

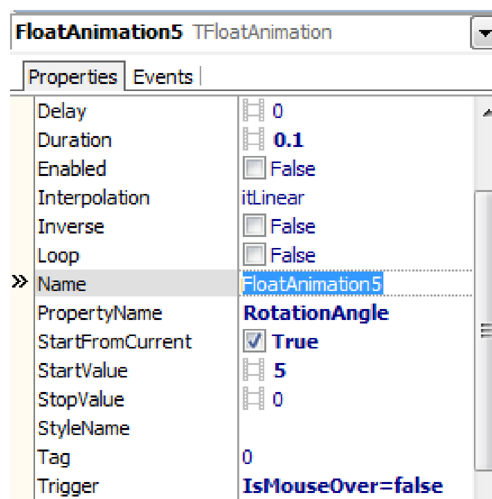
El siguiente paso será definir la animación asociada al `TButton`. En realidad, como se aprecia en la imagen de la página anterior, este control cuenta con dos `TFloatAnimation` como hijos. Un mismo control puede tener asociadas tantas animaciones como se quiera, normalmente vinculada cada una de ellas a un desencadenador distinto a menos que vayan a ser controladas por otros medios (propiedad `Enabled` o manualmente).

Nuestro objetivo es que al situar el puntero del ratón sobre el botón éste se balancee ligeramente a un lado y a otro alternativamente, de manera continua. Para ello animaremos su propiedad `RotationAngle` mediante el primer `TFloatAnimation`, usando como valores extremos `-5` y `5`. La duración de un ciclo será de un segundo, por lo que daremos el valor `1` a la propiedad `Duration`.

Activaremos las propiedades `Loop` y `AutoReverse`, de forma que la animación se repita y que al final de cada ciclo se invierta el sentido. Si no diésemos el valor `True` a `AutoReverse` al llegar al `StopValue`, que en este caso es 5, se volvería bruscamente al `StartValue`. Al hacerlo es como si el contenido de esas dos propiedades se intercambiase, reproduciéndose la animación en sentido inverso.

Finalmente , en cuanto al primer `TFloatAnimation` se refiere, daremos a la propiedad `Trigger` el valor `IsMouseOver=true` para que la animación se ponga en marcha automáticamente cuando el puntero del ratón pase sobre el `TButton`.

Una vez que la animación del botón se ha iniciado ya no se detendrá, porque hemos dado el valor `True` a `Loop`. Podríamos pensar en usar la propiedad `TriggerInverse`, pero esto solamente conseguiría reiniciar la animación en sentido inverso, no detenerla. Ésta es la razón por la que agregamos al `TButton` un segundo `TFloatAnimation`, con la configuración que puede verse en la imagen inferior para las propiedades `StopValue`, `Duration` y `Trigger`.



Lo que se consigue es que cuando el puntero del ratón deje de estar sobre el botón, momento en que IsMouseOver toma el valor False, se asigne a la propiedad RotationAngle del TButton el valor 0 de manera inmediata, de ahí que la duración sea de 0.1.

> NOTA
>
> Un método alternativo para conseguir este efecto habría sido asociar una animación a las propiedades StartValue/StopValue del primer TFloatAnimation. Un componente de animación puede tener como hijo a otro similar con el objetivo de cambiar sus propiedades, en especial las dos anteriores pero también otras como Delay y Duration.

La función de los controles TTrackBar que hemos añadido, uno lo alinearemos a la parte inferior del formulario y el otro al margen derecho, será alterar la escala del TLayout en el que están contenidos el resto de elementos de la interfaz. Ambos tendrán la misma configuración en las propiedades Min, Max y Value con los valores 25, 250 y 100, respectivamente, y compartirán un mismo gestor para el evento OnChange que será el siguiente:

```
procedure TForm1.TrackBar1Change(Sender: TObject);
begin
  Escena.Scale.X := TrackBar1.Value / 100;
  Escena.Scale.Y := TrackBar2.Value / 100;
end;
```

Escena es el nombre que se ha dado al TLayout que actúa como contenedor. Al asignar un valor distinto a la escala en el eje X y el eje Y podremos en cierta manera *deformar* el aspecto de la interfaz, estirándola a lo ancho o alto. Si quisiésemos mantener la proporción bastaría con usar un solo TTrackBar cuyo valor se usaría como escala en ambos ejes.

Finalmente el componente TSwitch[129] que colocamos en la parte superior izquierda de la ventana, con la propiedad IsChecked inicialmente a False, servirá para invertir toda la interfaz. Para ello rotaremos 180 grados el TLayout cuando el anterior control esté marcado, devolviéndola a su estado inicial cuando se desmarque. Con este fin asociaremos el código siguiente al evento OnClick del TSwitch:

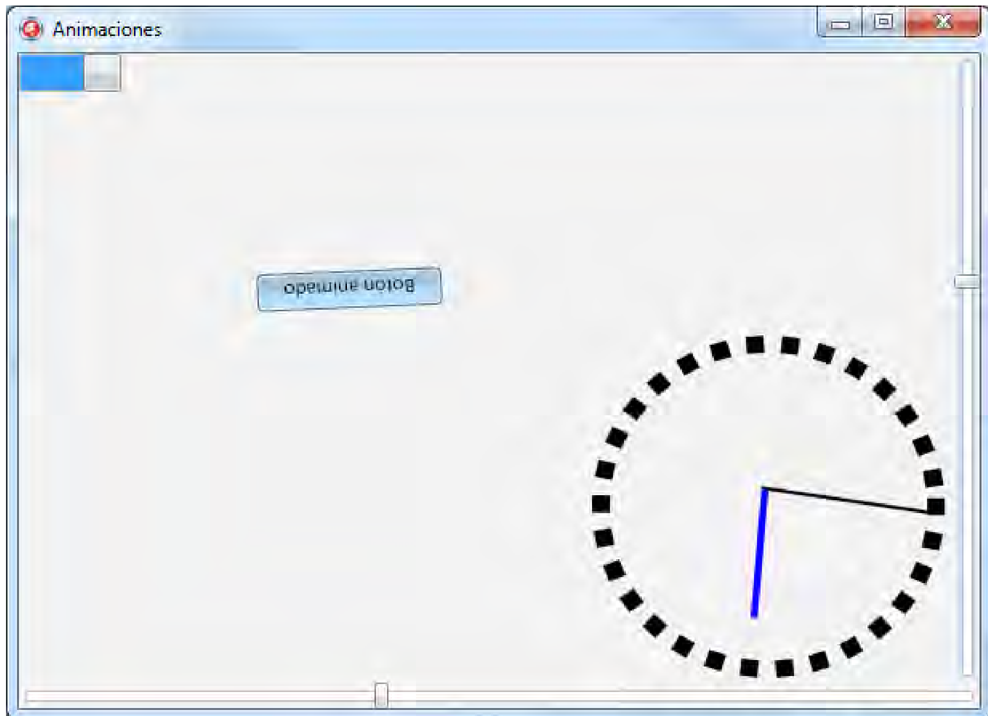
---

129 Este control es similar a un TCheckBox aunque con una apariencia diferente, pudiendo estar marcado o desmarcado según el estado de la propiedad IsChecked.

```
procedure TForm1.Switch1Click(Sender: TObject);
begin
  with Switch1 do
    Escena.RotationAngle := IfThen(IsChecked, 180, 0);
end;
```

> ADVERTENCIA
>
> Para poder usar la función `IfThen` hemos de agregar en nuestro código una referencia al módulo `Math`. Esta función es apropiada en este caso concreto porque las expresiones a evaluar en caso de que la condición sea cierta o falsa son dos constantes, pero podría no serlo en otras situaciones.

Con este código habríamos finalizado el desarrollo del proyecto. En la imagen inferior puede verse la ventana con el cronómetro en funcionamiento, la escala ligeramente ampliada, el botón balanceándose y la interfaz rotada 180 grados.



Efectos visuales y filtros

Mediante la modificación de ciertas propiedades a lo largo de un intervalo de tiempo, con componentes de animación como los descritos en el punto anterior, podemos generar efectos visuales básicos: cambios de posición, alteración de las dimensiones, modificar la orientación, realizar fundidos de color, etc.

FireMonkey nos ofrece muchos otros efectos más avanzados, algunos de ellos similares a los que pueden encontrarse en programas de retoque fotográfico. La mayor parte de los efectos están generados mediante filtros gráficos implementados como *shaders*, lo cual significa que se ejecutan en la GPU del ordenador sin afectar al rendimiento de la CPU.

Aunque muchos de los efectos están pensados para aplicarse a fotografías, un ejemplo concreto podría ser el que convierte una imagen en color a tonos sepia para darle una apariencia antigua, es posible usar estos efectos sobre cualquier control FMX.

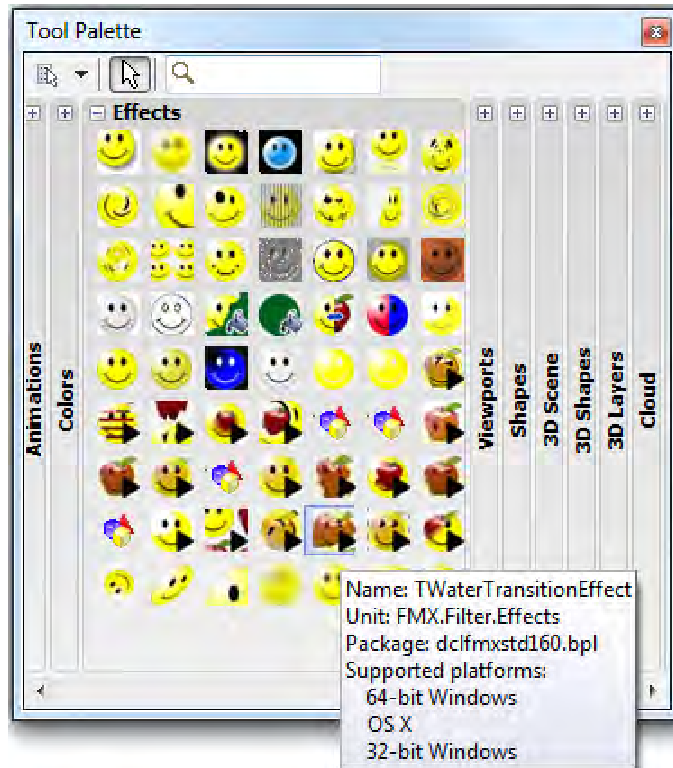
La jerarquía TEffect

Cada uno de los efectos visuales está implementado como una clase que, de forma directa o indirecta, deriva de `TEffect`. Esta clase, definida en el módulo `FMX.Types`, aporta dos propiedades: `Enabled` y `Trigger`, así como el método `ProcessEffect` que es el encargado de aplicar el efecto sobre los píxeles de un `TBitmap` que recibe como parámetro.

La propiedad `Enabled` es la encargada de activar o desactivar el efecto, un estado que puede alterarse mediante un desencadenador asignado a la propiedad `Trigger`. El esquema de funcionamiento es el mismo que ya conocemos para la propiedad del mismo nombre de los componentes de animación.

En el módulo `FMX.Effects` encontramos un reducido conjunto de clases derivadas directamente de `TEffect`. Son efectos visuales generados en su mayor parte mediante código ejecutado por la CPU, como el sombreado: `TShadowEffect`; los reflejos: `TReflectionEffect`, o el biselado: `TBevelEffect`. Cada uno de ellos cuenta con las propiedades heredadas de `TEffect` y otras específicas para configurar el efecto concreto, como pueden ser `Direction` y `ShadowColor` para ajustar el sombreado.

Uno de los derivados de `TEffect` es `TImageFXEffect`, clase que sirve como base para decenas de efectos gráficos que tienen un punto en común: todos ellos se apoyan en un filtro, representado por un objeto de la clase `TFilter`, implementado en forma de *shader* y que se ejecuta en la GPU. La mayor parte de los más de 60 componentes que encontramos en la página Effects de la Paleta de herramientas, visibles en la imagen inferior, son derivados de `TImageFXEffect`.



Estos componentes podemos separarlos en dos grandes categorías: los que aplican un efecto gráfico puntual según unos ciertos parámetros de configuración, por ejemplo resaltar los bordes de una imagen, y aquellos que están pensados para realizar una transición entre dos estados de manera gradual. Los primeros tienen un nombre que se ajusta al patrón `TnombreEffect` y los segundos al patrón `TnombreTransitionEffect`. Además estos últimos siempre cuentan con la propiedad `Progress` que es la que establece el porcentaje en que se encuentra el efecto de transición.

Es normal encontrar un mismo efecto visual en ambas categorías, por ejemplo TPixelateEffect y TPixelateTransitionEffect. El primero expone la propiedad BlockCount que nos permite ajustar el efecto de *pixelado* que queremos conseguir, mientras que el segundo prescinde de ese parámetro y ajusta el efecto automáticamente según el valor de la propiedad Progress. Esto mismo es aplicable a otras parejas de componentes: TBlurEffect y TBlurTransitionEffect, TMagnifyEffect y TMagnifyTransitionEffect, etc.

Configuración de un efecto visual

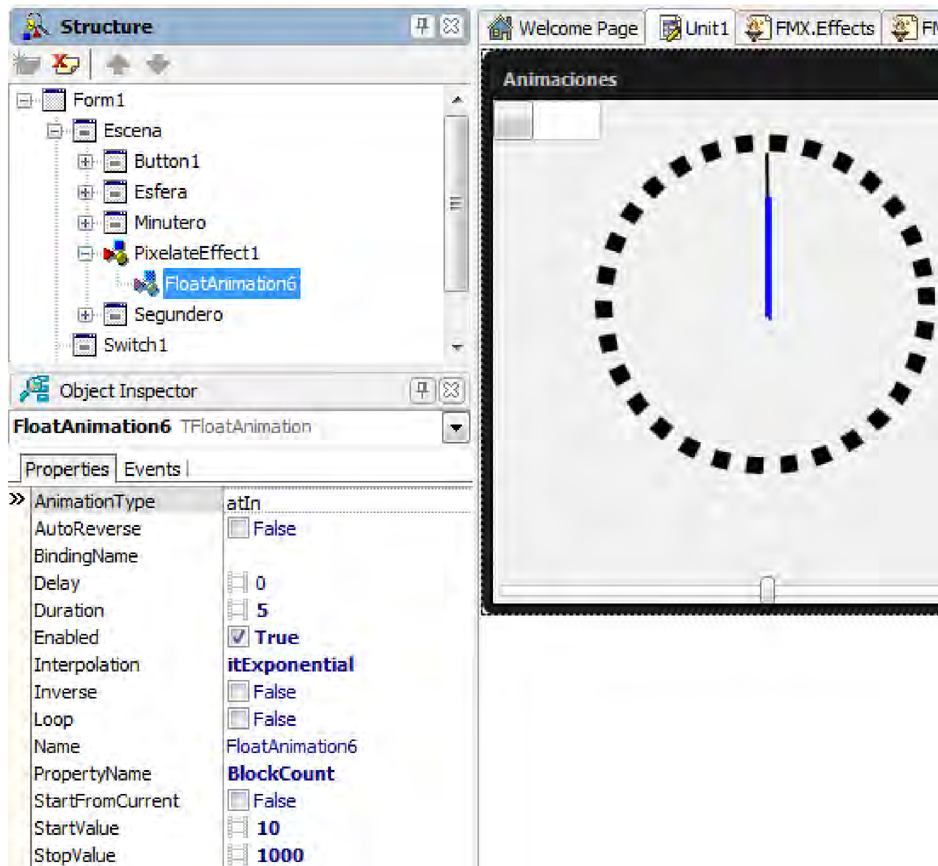
La asociación entre un componente que genera un efecto visual y el control sobre el que se aplicará se establece, como en el caso de las animaciones, introduciendo el primero como hijo del segundo. Para ello ya sabemos que hemos de tener seleccionado el padre en el diseñador a la hora de insertar el hijo desde la Paleta de herramientas.

> NOTA
>
> En caso de que creemos en ejecución (mediante código) el objeto que representa el efecto visual, o un componente para definir una animación, la asociación entre éste y el control sobre el que actuará se crea asignando a la propiedad Parent del primero una referencia al segundo. Éste es un paso fundamental, sin él el componente no encontraría el objeto sobre el que debe actuar.

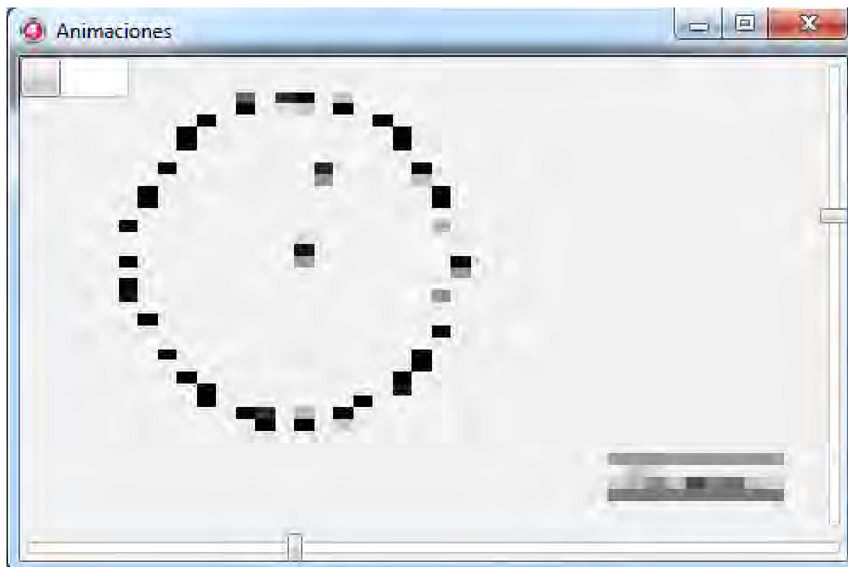
Aunque lo habitual, como se indicó anteriormente, es que el control sea un TImage o similar, de forma que el efecto gráfico actúe sobre una imagen, en la práctica es posible aplicar efectos sobre cualquier componente FireMonkey visible en un formulario.

Una vez asociados control destinatario del efecto gráfico y componente encargado de generarla, no hay más que usar las propiedades de este último para configurar el resultado. Los parámetros dependerán del tipo de efecto, como es lógico, y aparte de esto solamente usaremos las citadas propiedades Enabled y Trigger para activar/desactivar el efecto a demanda o asociarlo a un cierto evento.

La mayoría de las propiedades que ajustan el efecto gráfico producido por los componentes derivados de `TEffect` pueden tener asociada una animación, lo cual abre las puertas a crear composiciones realmente llamativas. Un ejemplo muy sencillo es el que puede verse en la imagen inferior, en el que hemos asociado un `TPixelateEffect` con el `TLayout` del proyecto de ejemplo del punto anterior. En lugar de asignar sin más un valor a la propiedad `BlockCount`, para establecer el grosor del mosaico obtenido, hemos colocado como hijo del `TPixelateEffect` un `TFloatAnimation`, de forma que el contenido de `BlockCount` cambie desde `10` a `1000` a lo largo de 5 segundos de manera exponencial, es decir, en principio los recuadros del mosaico serán muy grandes e irán refinándose poco a poco, de manera acelerada en el tiempo, hasta llegar a obtener la visualización por defecto.



Al ejecutar el programa la animación se pondrá en marcha, dado que hemos asignado el valor `True` a su propiedad `Enabled`, y el contenido de la ventana aparecerá como puede verse en la imagen inferior. La visualización irá haciéndose paulatinamente más nítida, pero en esa transición podemos comprobar que la interfaz sigue respondiendo como lo haría normalmente. El botón, por ejemplo, reproducirá la animación que lo balanceaba levemente al situar el puntero sobre él y responderá a la pulsación generando el habitual evento `OnClick`.



Como se comentó antes hay más de sesenta componentes en la FMX cuyo fin es generar efectos gráficos, cada uno de ellos con un conjunto diferenciado de propiedades para configurar el resultado. A pesar de que en la actual documentación de Delphi XE2[130] no es posible encontrar ayuda sobre estos componentes ni sus propiedades, en la mayoría de los casos nos bastará con insertarlos en un formulario y echar un vistazo al Inspector de objetos para deducir cómo funcionan.

---

130 Embarcadero está trabajando en actualizar la documentación electrónica que acompaña al producto, en especial la que afecta a todos los componentes FMX, por lo que es posible que en una futura actualización sí que encontremos en la ayuda toda la información relativa a los controles relacionados con la generación de efectos gráficos.

Interfaces 3D

Las animaciones, efectos visuales y transformaciones descritas en los puntos previos, que hemos aplicado a interfaces estándar, adquieren una mayor vistosidad cuando se usan en una interfaz en la que aparecen objetos tridimensionales[131]. Las posibilidades de FireMonkey en este sentido están casi a la altura de WPF[132] o Java3D, pudiendo definir objetos mediante mallas de puntos, aplicar materiales a las superficies, disponer luces de diferentes tipos y configurar cámaras para cambiar la vista de la escena, aunque con algunas limitaciones.

Una aplicación FireMonkey 3D precisa para su funcionamiento de Direct3D, en el caso de Windows, u OpenGL, cuando la plataforma es MacOS X o iOS. La aceleración hardware que aporta el alto paralelismo de la GPU, con capacidad para procesar cientos de vértices de manera simultánea, es fundamental al generar escenas 3D, especialmente si cuenta con animaciones.

> ADVERTENCIA
>
> Las aplicaciones FireMonkey HD pueden ser ejecutadas en máquinas sin una GPU moderna con algunas limitaciones, por ejemplo no pueden usarse los efectos basados en *shaders* ni ciertos componentes de visualización de objetos 3D. Para la ejecución de una aplicación FireMonkey 3D la GPU es imprescindible, si no se dispone de ella el programa fallará.

Sobre la composición de escenas 3D en general y su diseño mediante componentes FMX podría escribirse un libro monográfico. Únicamente los fundamentos relativos a los diferentes tipos de proyección, la aplicación de transformaciones mediante matrices o los tipos de iluminación (*shading*) de

---

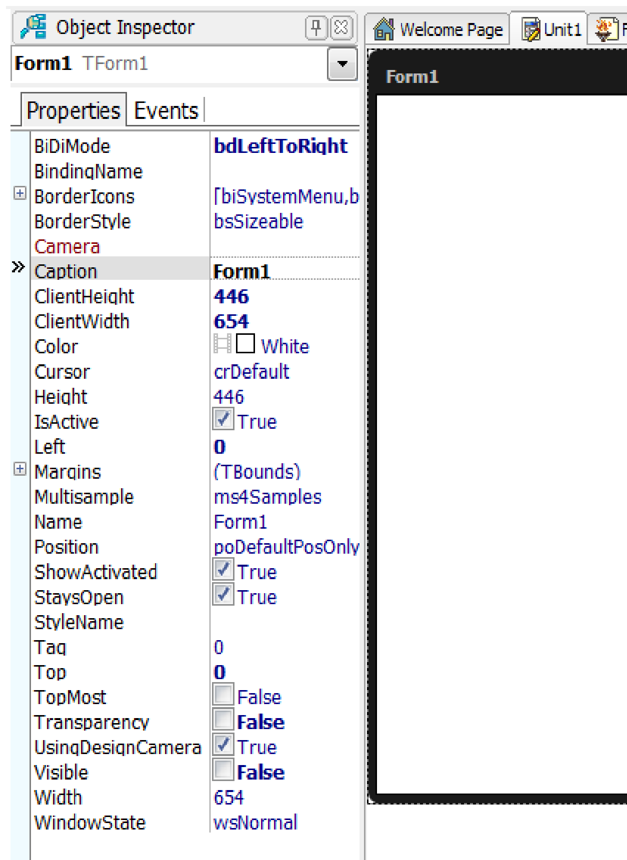
131 En realidad lo que veremos en la interfaz será una proyección bidimensional de una escena en tres dimensiones, no una estereografía generada mediante anaglifos, polarización o alguna otra técnica que permite realmente apreciar la profundidad.

132 *Windows Presentation Foundation.* Nombre que recibe la API gráfica introducida por Microsoft en Windows Vista y Windows 7, conjuntamente con el lenguaje XAML. Al igual que WPF, FireMonkey también usa Direct3D en Windows para aprovechar la aceleración hardware a la hora de trabajar con escenas tridimensionales.

los objetos darían para escribir cientos de páginas. En los puntos de este último apartado del capitulo solamente se ofrece una introducción muy breve centrada en los componentes FMX.

Estructura de una interfaz 3D

Las funciones 3D en una interfaz FMX parten de la interfaz `IViewPort3D`, definida en el módulo `FMX.Types3D` e implementada en la clase `TCustomForm3D`. De éste deriva la clase `TForm3D` que es el tipo de los formularios cuando se inicia un proyecto FireMonkey 3D. Si observamos el Inspector de objetos (véase imagen inferior) apreciaremos que estos formularios cuentan con muchas menos propiedades que los de las aplicaciones HD, ocurriendo lo mismo con los eventos.



El aspecto del formulario no denota sus especiales características, como el hecho de que incorpora una cámara que genera la visualización de la escena en la fase de diseño. La configuración de esa cámara es la que genera la proyección bidimensional a partir de los objetos 3D que se introduzcan en el formulario. El origen de coordenadas del formulario se encuentran en su centro y se corresponde a `X: 0, Y: 0, Z: 0`. Los ejes X e Y son los habituales y el eje Z sería perpendicular al plano que forman los anteriores, de forma que en su sentido positivo se alejaría de nosotros (según nuestra perspectiva al mirar la pantalla del ordenador) y en el negativo se acercaría.

> TRUCO
>
> Si queremos crear una escena en la que el contenedor global pueda responder a los eventos habituales, por ejemplo para interactuar con el ratón, y que contemple también las transformaciones habituales, podemos conseguirlo insertado en el formulario un componente `TLayout3D` y dando el valor `alClient` a su propiedad `Align`.

Los objetos de una escena 3D están formados por vértices, situados en el espacio mediante coordenadas 3D. Los vértices se conectan mediante aristas y éstas forman las caras de las entidades gráficas. Normalmente tendremos una cara visible[133] y otra oculta, situada en el interior del objeto o en la parte trasera. A las caras visibles se les aplican materiales que son los que generan la apariencia de las superficies.

Además de objetos con materiales y una cámara, en cualquier escena hace falta contar con una o más luces ya que de lo contrario no sería posible ver su contenido. Sería aproximadamente como entrar en una habitación a oscuras: no importa los objetos que haya ni sus colores, sin luz no podremos apreciarlos.

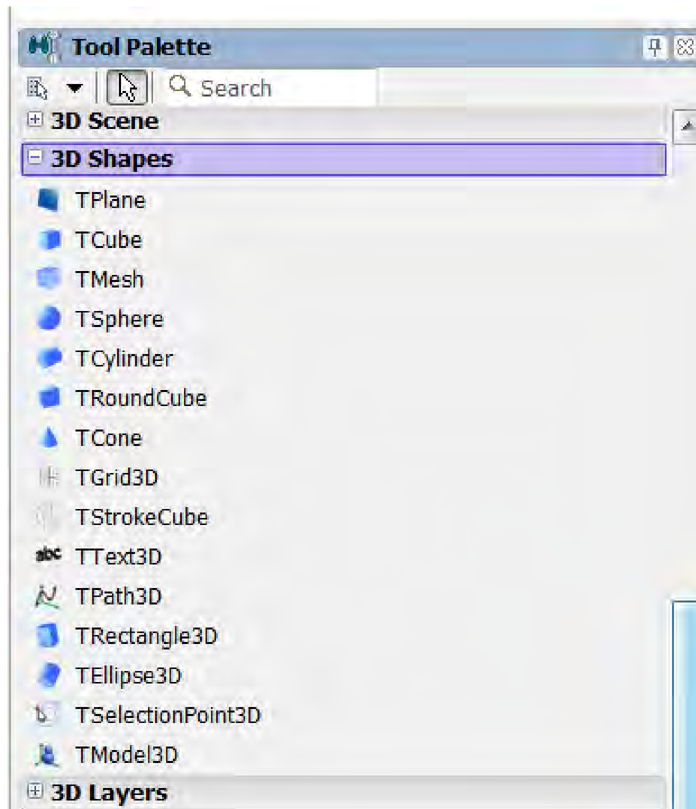
Todos los elementos mencionados forman lo que se conoce como un *grafo de escena*, un árbol en el que la raíz será el propio formulario y del que colgarán como hojas las cámaras, las luces y las entidades gráficas, pudiendo éstas contar con ramas con otros objetos. El orden de las entidades en dicho grafo determinará las relaciones entre ellas, como es habitual.

---

133 Las caras visibles pueden establecerse facilitando explícitamente el vector normal asociado o bien obteniendo dicho vector implícitamente a partir del orden en que se entregan los vértices.

Objetos tridimensionales

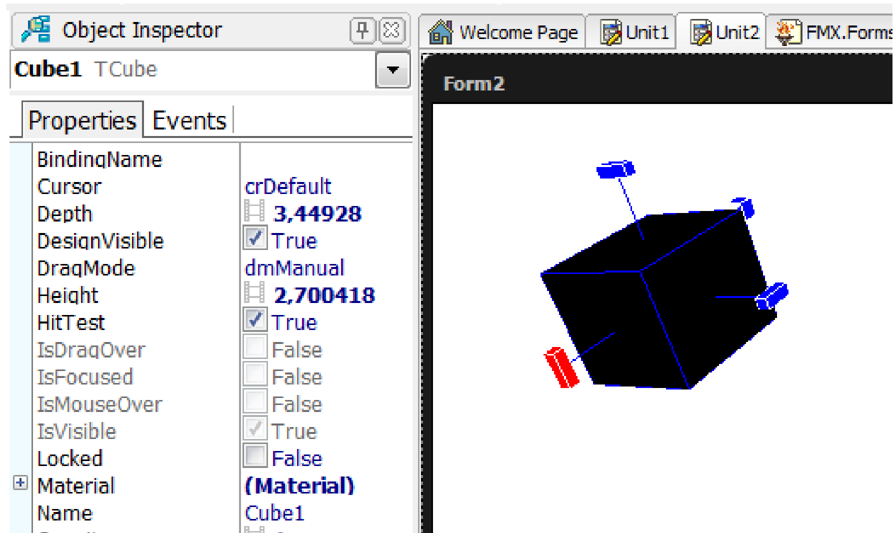
En la página 3D Shapes de la Paleta de herramientas la biblioteca FMX nos ofrece algunos objetos 3D clásicos, como son el cubo: TCube, la esfera: TSphere, el cilindro: TCylinder o el cono: TCone, junto con otros menos habituales como la elipse 3D: TEllipse3D o el cubo con esquinas redondeadas: TRoundCube. Algunas de estas entidades también cuentan con una versión en *alambre*, que es la denominación que reciben cuando se dibuja únicamente las aristas pero no se rellenan las superficies, como es el caso de TStrokeCube y TGrid3D. Esta última es la versión en alambre de TPlane.



Al insertar cualquiera de estos componentes en el formulario, asumiendo que aún no hemos introducido luz alguna, veremos una superficie oscura con la forma que corresponda. Además también aparecerán cuatro controles

enlazados al objeto, uno por cada eje más uno adicional que sirve para ajustar las dimensiones. Para habituarse al manejo de estos controles podemos insertar en el formulario un cubo, que es una entidad fácil de ajustar, y dedicar algo de tiempo a jugar con los giros y el tamaño en cada eje.

Como se aprecia en la imagen inferior, el control sobre el que esté situado el puntero del ratón aparece en un color distinto. Aunque apenas pueden distinguirse las caras del cubo, el dibujo de las aristas también en otro color nos sirve de ayuda. El control en forma de cubo pequeño que aparece adosado a uno de los vértices del cubo permite cambiar las dimensiones en los tres ejes posibles: alto, ancho y profundidad. Al mover el puntero del ratón sobre dicho control veremos que se ilumina una cara u otra, representando el eje sobre el que se actuará.

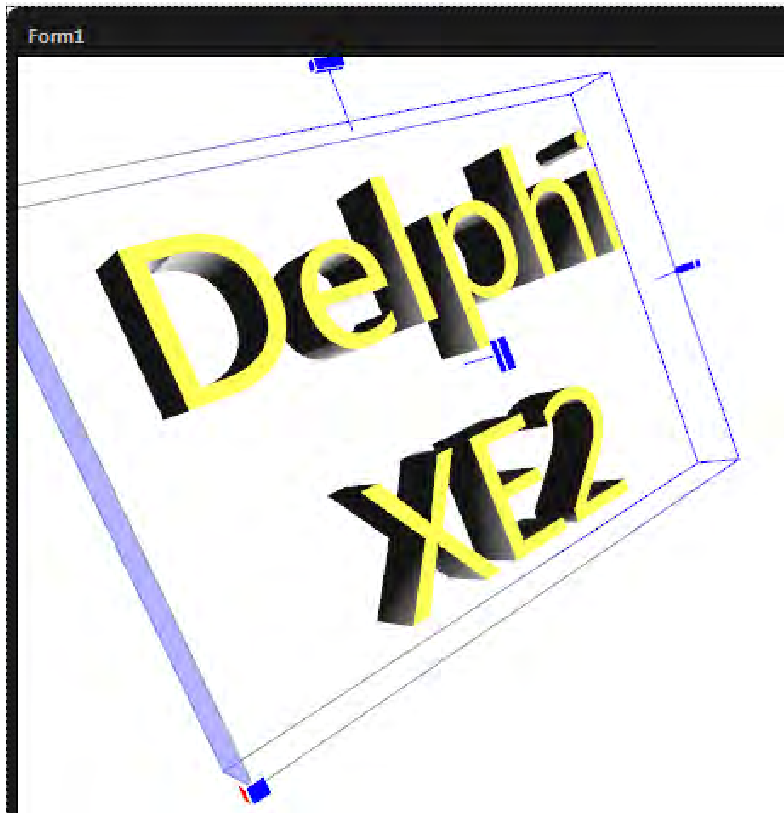


Los cambios que realicemos sobre el objeto 3D de manera interactiva se verán reflejados en varias de sus propiedades: `Position`, `RotationAngle`, `Width`, `Height` y `Depth`. Las dos primeras son de tipo `TPosition3D` y cuentan con tres miembros: `X`, `Y` y `Z`, fijando la posición[134] del objeto y su rotación en los tres ejes.

134 La posición del cubo es relativa al contenedor en que se encuentra, en este caso el formulario, y no se mide en píxeles, como tampoco se usa esta unidad de medida para las dimensiones.

Por defecto el tamaño de los objetos 3D suele ser de una unidad en cada uno de los ejes: ancho, alto y profundidad. De igual forma, su posición inicial siempre es `(0, 0, 0)`, apareciendo en el centro del contenedor en el que se hayan insertado.

Algunos objetos que no tienen naturaleza 3D, como es el caso del texto, adquieren la tercera dimensión mediante extrusión en el componente `TText3D`. Tal y como se aprecia en la imagen inferior podemos dar profundidad y rotar el texto en cualquiera de los tres ejes.



Si estuviésemos limitados a los objetos predefinidos que acaban de mencionarse, apenas media docena en total, componer una escena de cierta complejidad sería algo casi imposible. En realidad podemos generar cualquier otro objeto siempre que seamos capaces de facilitar información sobre sus vértices: la posición en que se encuentran en el espacio tridimensional, y la forma en que se unen.

El componente `TMesh` es todo lo que necesitamos para colocar una malla de triángulos en la escena. Partiendo de una nube de puntos indicaremos cómo unirlos mediante aristas y definir así las caras de cualquier objeto tridimensional en base a una serie de triángulos[135]. Las propiedad fundamental de `TMesh` es `Data`, un objeto de tipo `TMeshData` con los miembros siguientes:

- `Points`: Es una cadena con las coordenadas 3D de una colección de puntos en el espacio. Cada número (coordenada X, Y o Z) se separa del siguiente con un espacio y cada conjunto de tres, que definen la posición de un punto, también se separa del siguiente conjunto con un espacio. Estos puntos no serán dibujados en la escena, solo sirven como información para dar forma al objeto 3D. Es importante saber el orden de los puntos en esta colección: al primero le corresponde el índice `0`, al siguiente el `1` y así sucesivamente.
- `TriangleIndices`: Al igual que la anterior esta propiedad también contendrá una cadena con una colección de números, pero en este caso serán siempre números enteros, concretamente los índices de los puntos definidos en `Points`. Cada conjunto de tres índices define un triángulo y cada índice puede reutilizarse las veces que sea necesario, dando así conectividad a los triángulos. El orden en que se facilitan los índices de cada triángulo determina cuál será su cara visible.
- `Normals`: Por cada triángulo esta propiedad contendrá la definición de su vector normal, un dato que afecta sobre todo a la iluminación de las caras del objeto. Por defecto el vector normal de cada triángulo suele ser perpendicular al plano del que forma parte el triángulo. Esta propiedad obtiene una configuración por defecto establecida a partir de los índices de triángulos de la propiedad anterior.
- `TexCoordinates`: En caso de que vaya a aplicarse una textura a la superficie de los triángulos,en lugar de un color sólido, esta propiedad establecerá las coordenadas de textura para cada una de las caras del objeto.

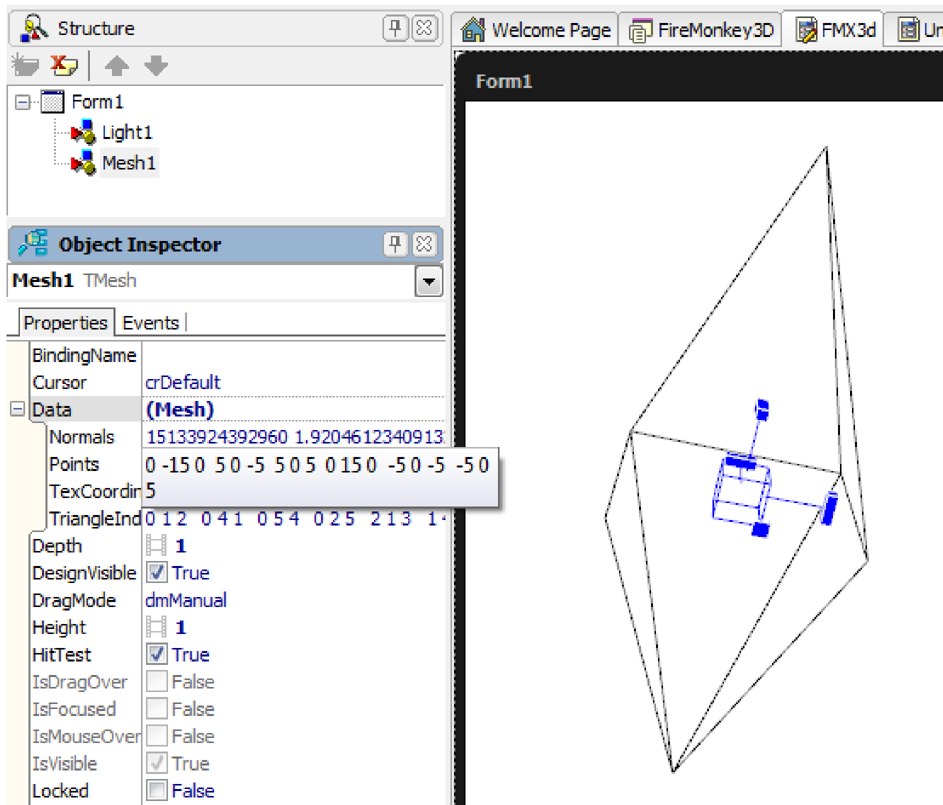
---

135 Cualquier superficie 3D puede ser descompuesta en una colección de triángulos que formarán una malla. A mayor complejidad del objeto y cuanto más detalle se requiera habrá que usar más triángulos de menor tamaño.

Usando exclusivamente las dos primeras propiedades podemos, por ejemplo, dibujar un octaedro asignando los valores siguientes:

```
with mesh.Data do begin
   Points := '0 -15 0  5 0 -5  5 0 5  0 15 0  -5 0 -5  -5
0 5'
   TriangleIndices := '2 0 1   0 4 1   0 5 4   0 2 5   2 1
3   1 4 3   4 5 3   2 3 5'
end;
```

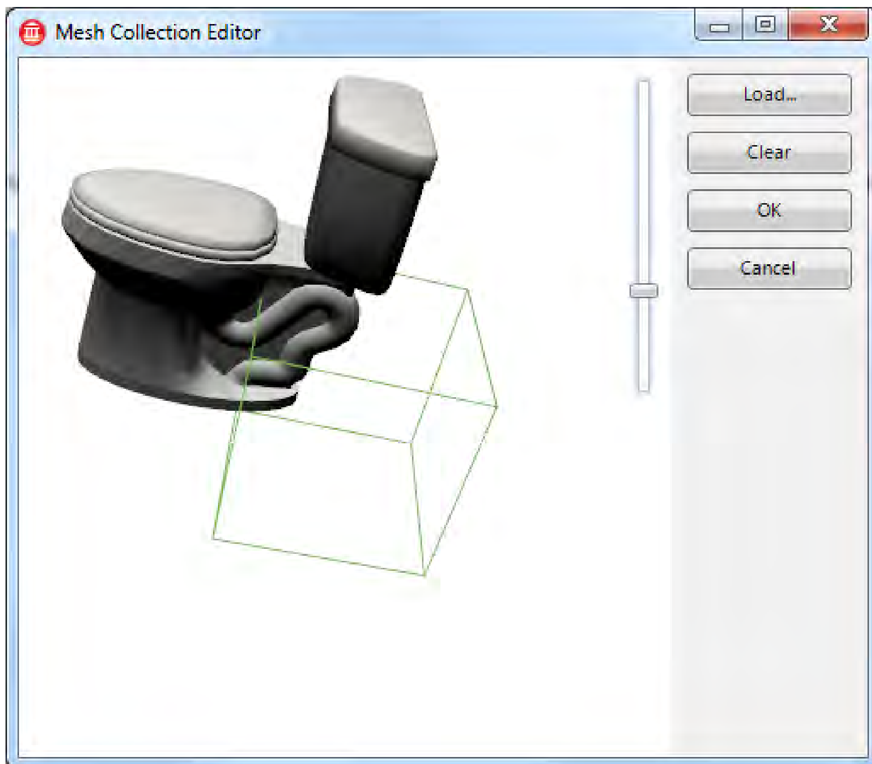
También podemos introducir las cadenas de puntos e índices de triángulos en la fase de diseño, mediante el Inspector de objetos, lo cual nos permite observar el resultado de manera inmediata como se aprecia en la imagen inferior.



Una vez definido el objeto 3D, mediante la malla de triángulos, sus propiedades no difieren de las de cualquier otra figura 3D predefinida de FireMonkey, contando con escala, posición y rotación, material, etc.

> NOTA
>
> Tanto TMesh como la mayor parte de los objetos 3D predefinidos, como TCube o TSphere, derivan de una clase ascendiente común: TCustomMesh. Ésta a su vez es una derivada de TShape3D, clase que sirve como nexo de unión con otras como TText3D o TRectangle3D.

Si bien es posible crear objetos compuestos de mallas de triángulos *a mano*, calculando los puntos y definiendo los índices de cada triángulo, para modelos realmente complejos lo habitual es recurrir a herramientas de diseño específicas. Una vez que se ha obtenido el modelo deseado éste se exporta a un archivo y, posteriormente, se recupera desde Delphi gracias al componente TModel3D. Éste cuenta con una propiedad MeshCollection, que tiene asociado el editor que aparece en la imagen inferior, con capacidad para almacenar varias mallas de triángulos (objetos TMesh), dando así lugar a escenas con más de un objeto 3D.



Con el componente TModel3D podemos recuperar modelos a partir de diferentes formatos de archivos 3D[136]. La arquitectura interna para hacer esto posible parte de la clase TModelImportServices, definida en el módulo FMX.Import, y está diseñada para ser extensible, por lo que podríamos agregar la posibilidad de trabajar con otros formatos.

Materiales

Todos los derivados de TShape3D, entre los que se cuentan los componentes mencionados en los apartados previos a excepción de TModel3D, cuentan con la propiedad Material, un objeto de clase TMaterial. Con sus propiedades se establecen las características que tendrán las partes visibles de los objetos: color, textura, reacción ante la luz, etc.

Por defecto la propiedad Texture de TMaterial es nula, por lo que la superficie del objeto será lisa: tendrá un color uniforme afectado únicamente por la iluminación. Podemos recuperar cualquier mapa de bits y aplicarlo como textura, ésta se *pegará* a la superficie del objeto según las coordenadas de textura que se hubiesen establecido. En el aspecto de la textura también influirá la iluminación del objeto, configurada con las siguientes propiedades:

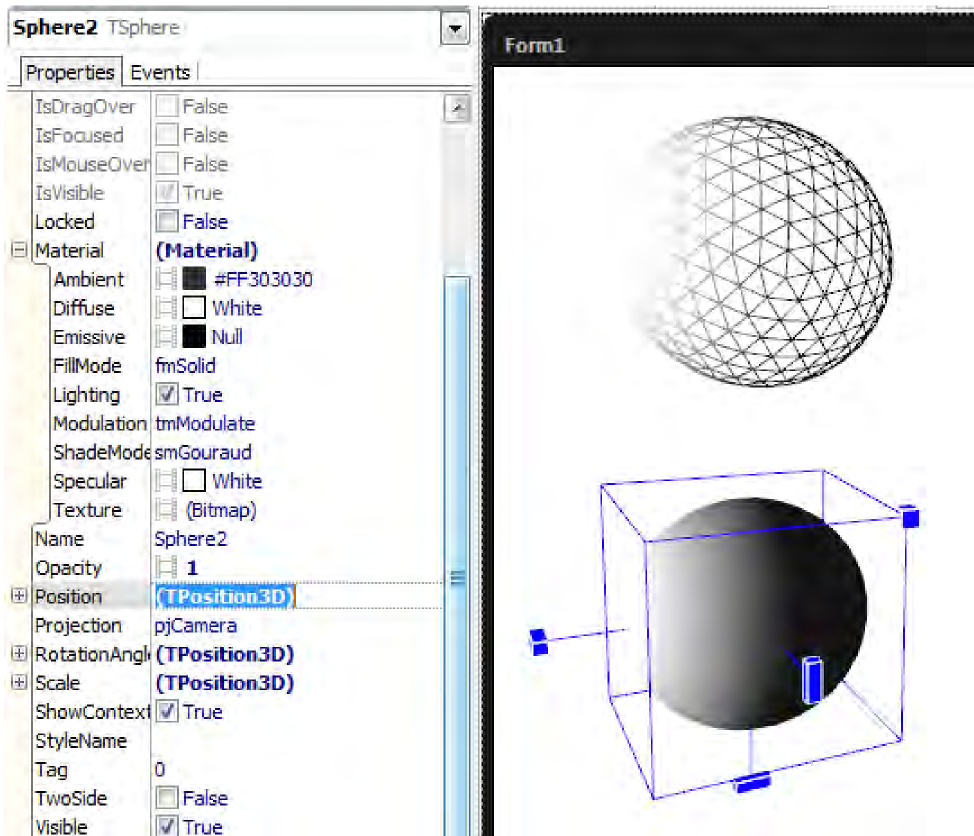
- ShadeMode: Determina cómo se llevará a cabo la iluminación de cada una de las caras del objeto 3D (cada triángulo), seleccionando el tipo de *shading* (sombreado) a utilizar. Existen dos opciones: smFlat y smGouraud. El primero aplica una iluminación plana, lo que significa que toda la superficie del triángulo tendrá el mismo color[137]. Con el segundo se simula el efecto progresivo de la iluminación sobre las superficies, sin saltos bruscos, por lo que el color de cada punto del triángulo se obtendrá mediante un proceso de interpolación entre sus vértices. Existen otros tipos de *shading* más avanzados, como el sombreado de Phong, capaces de dar una apariencia más realista, pero no están contemplado en FireMonkey.

---

136 En http://sketchup.google.com podemos encontrar multitud de modelos 3D. El ejemplo que aparece en la imagen de la página previa procede de dicho sitio.

137 Si probamos este tipo de sombreado en un objeto de cierta complejidad, como puede ser una esfera (TSphere), apreciaremos perfectamente cómo está compuesto de pequeños triángulos porque los bordes entre éstos, al tener cada uno un color plano uniforme, quedarán resaltados claramente.

- `Lighting`: De tipo `Boolean`, esta propiedad controla si se aplica o no iluminación al objeto en cuestión. Por defecto tiene el valor `True`, si la ponemos a `False` el objeto normalmente se volverá invisible.
- `FillMode`: Es una numeración con dos valores posibles: `fmSolid` y `fmWireframe`. El tomado por defecto es el primero y provoca que el objeto aparezca como un sólido, con el material aplicado a la superficie de cada triángulo. Si usamos el segundo únicamente se dibujará el modelo en alambre, como puede apreciarse en la parte superior de la imagen siguiente, en la que aparece el mismo objeto en los dos estados posibles. A pesar de dibujarse únicamente las aristas de los triángulos, en este caso de dos esferas, el color de las aristas también se verá afectado por la configuración de iluminación establecida por las cuatro propiedades siguientes.



- `Ambient`: Establece el color que reflejará el objeto ante la luz ambiental. Ésta es una iluminación general, que procede de todas direcciones con igual intensidad, por lo que el color elegido se distribuirá homogéneamente por la superficie del modelo.
- `Diffuse`: La iluminación difusa proviene de algún punto en el infinito, por lo que los imaginarios rayos de luz son todos paralelos. Es lo que ocurre, por ejemplo, con la luz procedente del sol. El color asignado a esta propiedad sería el que aparecería sobre la parte de la superficie del objeto en la que incidiese este tipo de luz, lo que habitualmente se traduce en la aparición de una parte más brillante. Por defecto el color elegido es el blanco y genera ese efecto.
- `Specular`: Es un tipo de luz similar a la anterior, ya que procede de un foco e incide sobre la superficie con una cierta dirección, provocando la aparición de partes brillantes en el objeto. Cuando se quiere obtener un objeto de un material similar al metal o el plástico se asignará a esta propiedad el color a reflejar.
- `Emissive`: Se utiliza para crear objetos que emiten luz, como puede ser una lámpara, sin que importe la configuración de iluminación que exista alrededor.

Nos ayudará a comprender la influencia de cada una de estas propiedades, en el aspecto final de un objeto, *jugar* un tiempo con ellas, por ejemplo colocando uno o más modelos predefinidos y usando en unos texturas y en otros colores lisos, alterando las cuatro propiedades anteriores y observando los cambios que se producen. Para ello, no obstante, tendremos que colocar una o más luces en la escena, ya que de lo contrario todos los objetos aparecerán planos, sin volumen.

> NOTA
>
> Si estás interesado en aprender más sobre modelos de sombreado e iluminación, así como sobre los tipos de materiales y luces, puedes recurrir a cualquier documentación sobre OpenGL. Los componentes 3D de FireMonkey utilizan dicha API gráfica (o Direct3D en el caso de Windows), sobre la cual es posible encontrar abundante información en la web.

Luces

Para que una escena sea algo más que una colección de sombras, distinguibles únicamente porque el fondo del formulario es blanco, tendremos que añadir una o más fuentes de luz. Cada una de ellas estará representada por un objeto de clase `TLight`, componente que encontramos en la página 3D Scene de la Paleta de herramientas.

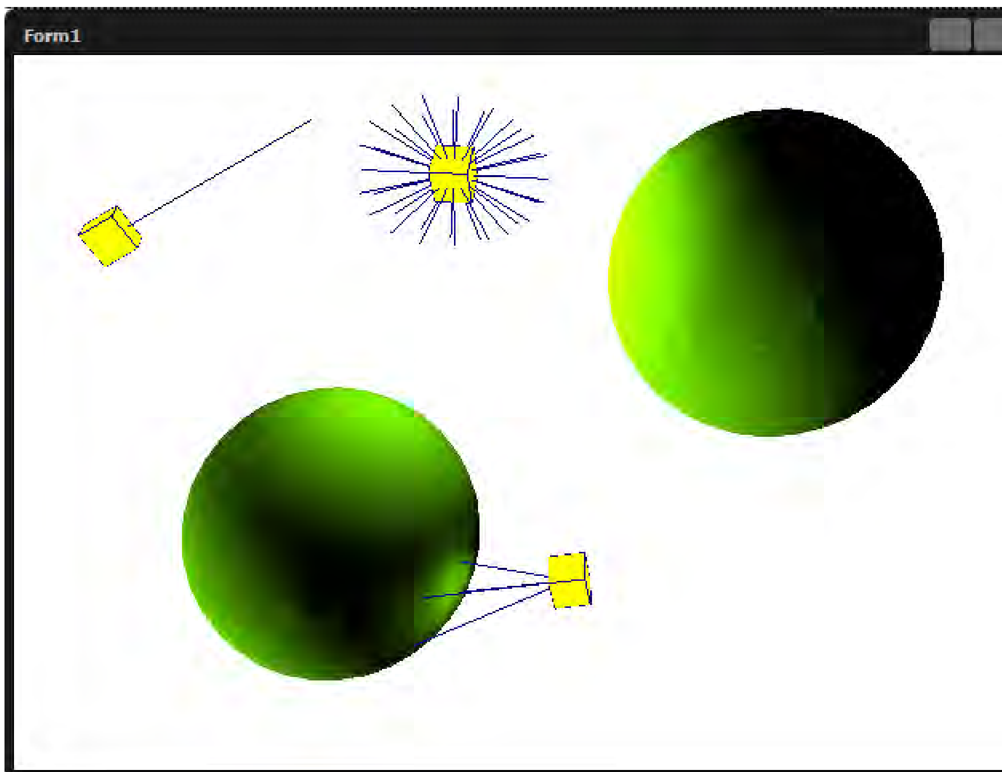
La propiedad fundamental de este componente es `LightType`, encargada de de determinar el tipo de luz a generar. Ésta puede ser de tres tipos diferentes que se corresponden con los tres valores de dicha propiedad:

- `ltDirectional`:El primer tipo corresponde a una luz con una posición en el infinito y cuyos rayos llegan todos paralelos, por lo que cambiando la propiedad `Position` no se altera su influencia, pero sí al modificar la propiedad `Rotation` ya que ésta es la que establece la dirección de incidencia. Podemos colocar una luz de este tipo en la escena, ajustar el ángulo con que incide en los objetos y después mover el componente `TLight` por el formulario para apreciar que la iluminación no cambia. Sería similar a la luz solar.
- `ltPoint`: Este tipo permite introducir un punto de iluminación omnidireccional, es decir, que emite luz en todas direcciones. Por ello la rotación aplicada a la fuente de luz no afecta a la forma en que incide en los objetos de la escena, pero sí la posición en que se coloque en la escena. Sería similar al comportamiento de una bombilla de tipo bulbo incandescente.
- `ltSpot`: El tercer tipo de luz es el de un foco que emite desde una posición y en una dirección concretos, por lo que iluminará las caras de aquellos objetos que se encuentran enfrentadas a esa dirección en un grado que dependerá de la distancia. Cuando más cerca esté la cara de la luz más concentrada será la iluminación. Sería similar al comportamiento de una típica lámpara de escritorio o lectura.

> NOTA
>
> Si revisamos las propiedades de un objeto `TLight` comprobaremos que no hay propiedades que sirvan para establecer el color de luz emitida. Se asume que ésta siempre es blanca, el color final de cada superficie dependerá principalmente del material usado para cada objeto.

La inclusión de múltiples fuentes de luz en una escena, cada una de ella con una configuración adecuada, permite crear efectos más realistas. Sobre todo cuando se combina con objetos formados por mallas de triángulos de mayor complejidad a los que se aplican texturas.

En la imagen inferior puede verse una escena sencilla en la que intervienen dos objetos idénticos: dos esferas del mismo tamaño y con el mismo material, solamente difieren en la posición que ocupan en los ejes X e Y. La posición en el eje Z es la misma, de forma que el tamaño obtenido tras la proyección 3D a 2D sea también idéntico. Las diferencias que percibimos, por tanto, se deben exclusivamente a la iluminación. En ésta intervienen tres puntos de luz: uno direccional que ilumina los objetos desde abajo, sin importar donde se encuentren; otro puntual que ilumina la parte superior de la esfera izquierda y el lateral izquierdo de la esfera derecha, y una luz focal que incide solamente en la parte inferior derecha de la esfera izquierda.



Cámaras

Una misma escena generará distintas proyecciones en el plano dependiendo de dónde situemos la cámara, es decir, cambiará según desde donde estemos mirando si asumimos que nuestros ojos hacen las veces de cámara. Si nos movemos, desplazándonos en cualquiera de los tres ejes, o si giramos la cabeza a izquierda-derecha o arriba-abajo, la imagen incidente en nuestros ojos es distinta.

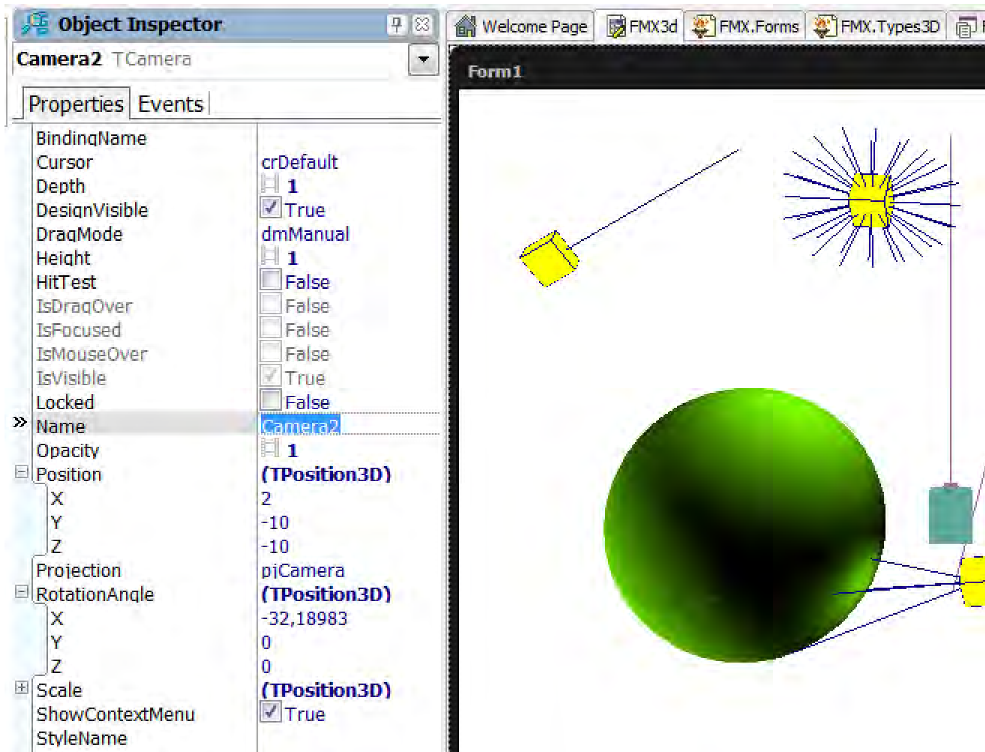
Las cámaras están representadas en FireMonkey[138] por objetos `TCamera` y toda escena cuenta con una cámara por defecto: la que genera la proyección que estamos viendo en la fase de diseño. Ésta será la cámara activa siempre que no desactivemos la propiedad `UsingDesignCamera` del formulario, cuyo valor por defecto es `True`. Si lo cambiamos a `False` la proyección de la escena, que generará la visualización final en la ventana, se realizará usando la cámara indicada por la propiedad `Camera` del formulario.

Durante la ejecución de un programa podemos tanto cambiar de cámara, alterando el contenido de esa propiedad, como modificar las propiedades de un objeto `TCamera`. Si ésta es la cámara activa la alteración de su configuración provocará una actualización inmediata de la proyección de la escena. Es algo que podemos probar de forma sencilla reproduciendo los pasos siguientes:

- Desmarcamos la propiedad `UsingDesignCamera` del formulario, de forma que la cámara activa dependa de la propiedad `Camera`.
- Agregamos a la escena del apartado anterior, que contenía las dos esferas, un objeto `TCamera`. Ésta se asigna automáticamente a la propiedad `Camera` como cámara activa. Mantendremos su posición y ángulos de rotación.
- A continuación insertamos un segundo objeto `TCamera`, una cámara secundaria que situaremos de forma que nos permita observar la escena desde arriba, eligiendo un punto ligeramente desplazado a la derecha. En la imagen de la página siguiente puede observarse la configuración de sus propiedades `Position` y `RotationAngle`.

---

138 En FireMonkey tenemos poco control sobre la cámara, ya que básicamente estamos limitados a configurar su posición y orientación, pero no el resto de parámetros como el ángulo de apertura, planos de recorte frontal y posterior y parámetros similares que sí podemos configurar al trabajar directamente con OpenGL o Direct3D.



- Usaremos el evento `OnClick` de la esfera que está más a la derecha para alternar entre las dos cámaras de la escena, usando para ello el siguiente código:

```
procedure TForm1.Sphere2Click(Sender: TObject);
begin
  if Camera = Camera1 then
     Camera := Camera2
  else Camera := Camera1;
end;
```

- Lo siguiente será añadir un componente `TTimer`, configurado para que genere un evento `OnTimer` cada décima de segundo, para lo cual modificaremos su propiedad `Interval` dándole el valor 100. El objetivo es modificar la posición en el eje Z y el ángulo de giro en el eje Y de la primera cámara que añadimos, para conseguir como resultado un efecto de animación en el que parezca que vamos alejándonos de la escena y girando la cabeza alternativamente a

izquierda y derecha. Cada vez que se genere el evento `OnTimer` se ejecutará el código mostrado a continuación. La variable `offsetAngulo` es un miembro privado de la clase, de tipo entero, que hemos inicializado con el valor 1.

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  with Camera1.Position do
    Z := Z - 0.1;

  with Camera1.RotationAngle do begin
    Y := Y + offsetAngulo;
    if Abs(Y) = 30 then offsetAngulo := offsetAngulo * -1;
  end;
end;
```

Ya podemos ejecutar el programa y apreciar el movimiento de la cámara, que se traduce en la animación de la escena. Hemos de hacer clic sobre la esfera de la derecha para alternar entre las dos cámaras, accediendo a la proyección estática desde arriba.

Objetos 2D en un mundo 3D

No es extraño que al desarrollar una aplicación en la que se trabaja con escenas tridimensionales también se tenga la necesidad de contar con una interfaz clásica, basada en controles 2D, que permita al usuario elegir opciones, cambiar la configuración de los objetos, etc. Los componentes que usamos en una interfaz FireMonkey HD, como pueden ser `TListBox`, `TLabel`, `TButton` y `TCheckBox`, no están preparados para hacerse visibles en un formulario de una aplicación FireMonkey 3D, ya que su propiedad `Position` es bidimensional, no 3D.

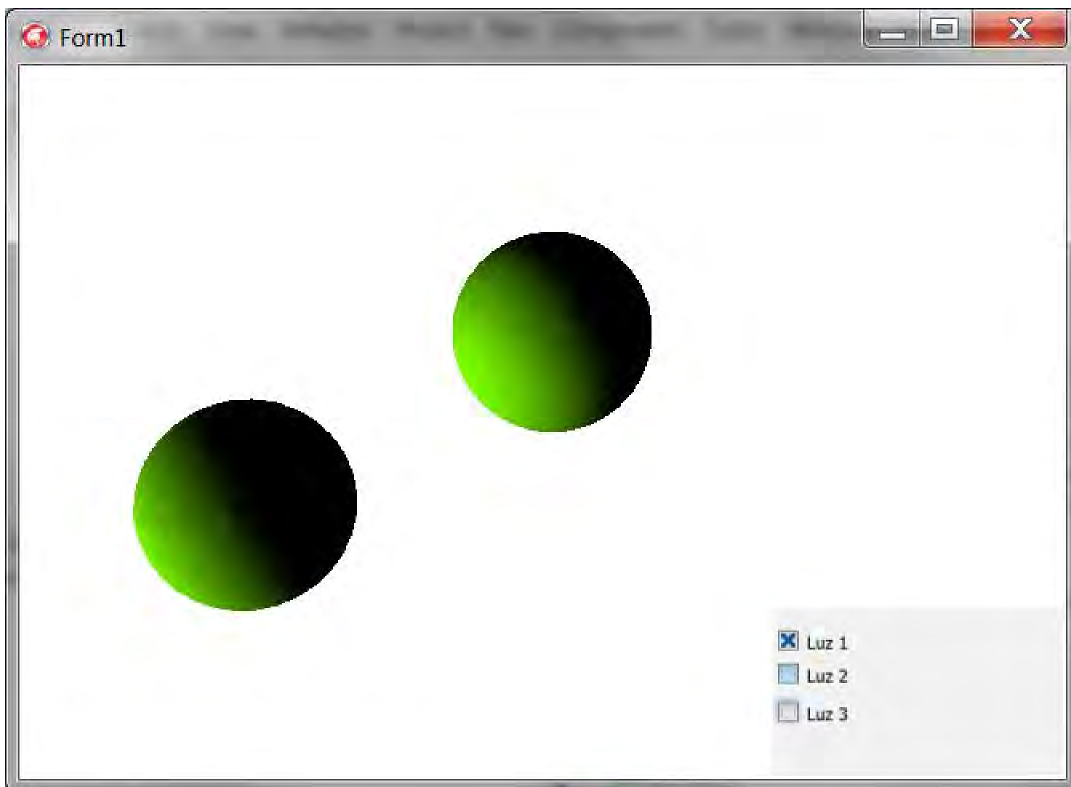
La solución pasa por colocar ese tipo de controles sobre una superficie (un contenedor) que resuelva su posición, dimensiones y orientación. Dicha superficie es el componente `TLayer3D` que, en cierta forma, actuará como un formulario 2D alojado en el formulario 3D.

Dicho componente tiene una posición 3D y también ángulos de giro para cada uno de los tres ejes pero, por regla general, desactivaremos todas esas características para conseguir una visualización independiente de la proyección de la escena. Para ello no hay más que cambiar el valor de la propiedad `Projection`, que por defecto es `pjCamera`, eligiendo `pjScreen`.

Partiendo del proyecto de los apartados anteriores, podríamos agregar a la escena un TLayer3D con la finalidad de alojar en él tres controles TCheckBox que sirvan para activar y desactivar selectivamente las luces aplicadas. Inicialmente la propiedad IsCheckqued de los tres sería True y el evento OnChange sería compartido, ejecutando un mismo método con el código siguiente:

```
procedure TForm1.CheckBox1Change(Sender: TObject);
begin
  Light1.Enabled := CheckBox1.IsChecked;
  Light2.Enabled := CheckBox2.IsChecked;
  Light3.Enabled := CheckBox3.IsChecked;
end;
```

En la imagen inferior puede verse el aspecto del programa en ejecución, tras haber desactivado dos de los puntos de luz existentes. De manera similar podríamos agregar una lista desplegable para elegir entre las cámaras existentes, controles para mover la cámara activa o cambiar su orientación, etc.



Conclusión

En este extenso capítulo hemos conocido muchas de las características del diseño de interfaces de usuario con la biblioteca de componentes FireMonkey, tanto clásicas (2D) como conteniendo escenas 3D. Hemos aprendido a usar muchos de los componentes de dicha biblioteca, tanto visuales como aquellos no visibles pero que influyen en el comportamiento y aspecto de los que sí lo son.

Se ha descrito el funcionamiento de los estilos visuales, los libros de estilos y la forma en que se asocian con los formularios, y hemos creado un estilo a medida para un tipo de control, explicándose también la aplicación de animaciones y de efectos gráficos.

La visión global que hemos obtenido de FireMonkey nos permitirá aprovechar las características de esta potente biblioteca, algo que nos interesa básicamente por dos causas: es multiplataforma y puede ser usada para desarrollar aplicaciones en OS X e iOS, algo que no puede decirse de la VCL, y está llamada a ser la biblioteca de controles principal en futuras versiones de Delphi, a pesar de que el desarrollo de la VCL no se abandone.

A continuación

Una vez que conocemos las posibilidades de FMX llega el momento de comprobar cómo usar esta biblioteca a la hora de desarrollar aplicaciones para otros sistemas operativos.

El primer objetivo es MacOS X, del que nos ocuparemos en el capítulo siguiente. En el posterior abordaremos la creación de proyectos para iOS, el sistema operativo de los iPhone y iPad.

Capítulo 12: Delphi XE2 y MacOSX

Junto con la nueva biblioteca de componentes multiplataforma FMX, que hemos conocido a fondo en el capítulo previo, seguramente la novedad de mayor interés de Delphi X2 es la incorporación de un compilador Delphi capaz de generar código nativo para MacOS X. La unión de este compilador, las versiones para MacOS X de la RTL y la FMX nos permiten desarrollar con Delphi aplicaciones para el sistema operativo de Apple con una gran sencillez.

En este capítulo se describe el proceso que hemos de seguir para generar una versión para MacOS X de nuestro proyecto, cómo desplegarlo en la máquina de destino y los detalles que hemos de tener en cuenta a la hora de distribuir la versión final de la aplicación.

El compilador para MacOS X

Como ya sabemos Delphi, el entorno de desarrollo en sí, es una aplicación para Windows de 32 bits, por lo que no podemos usarlo sobre MacOS X. Al menos no directamente, aunque podríamos recurrir a una máquina virtual instalada en el Mac. El compilador para MacOS X incluido en Delphi XE2 tiene esa misma naturaleza: es una aplicación para Windows. Sin embargo es capaz de generar ejecutables para MacOS X. Es lo que se denomina un compilador cruzado.

Es una situación muy parecida a la descrita en el décimo capítulo, al tratar la creación de aplicaciones para Windows de 64 bits, que nos forzará a trabajar con dos máquinas: una en la que se ejecuta Delphi XE2, que asumiremos como local, y otra en la que está instalado MacOS X, a la que llamaremos máquina remota.

El compilador para MacOS X, llamado DCCOSX.EXE, es equivalente al DCC32.EXE estándar de Delphi y, como éste, únicamente puede generar código de 32 bits. Esto significa que incluso si tenemos una versión de MacOS X de 64 bits[139], y un ordenador con 4 gigabytes o más de memoria, nos encontraremos con las limitaciones que se describieron en el décimo capítulo.

La integración entre el compilador Delphi y algunas de las características de OS X, como la gestión de excepciones Mach[140], se ve asistida por código a medida introducido en la RTL en su versión para Mac. Lo encontraremos en el módulo `System.Internal.MachExceptions`. Gracias a dicho código podemos seguir usando las excepciones habituales de Delphi.

Asimismo el requerimiento de OS X de que cuando se invoca a una función la pila esté alineada en una dirección múltiplo de 16, algo poco habitual en sistemas con arquitectura Intel x86, queda en manos del compilador. No obstante es un detalle que hemos de tener en cuenta[141] en caso de que introduzcamos código en lenguaje ensamblador en nuestro proyecto.

---

139 Se espera que una futura versión de Delphi, posiblemente el próximo XE3, cuente con la capacidad de generar aplicaciones MacOS X de 64 bits igual que para Windows de 64 bits.

140 Mach es el nombre por el que se conoce al núcleo del sistema operativo MacOS X, evolución del que incorporó NeXTStep y que procede originalmente de un desarrollo de investigación de la Universidad Carnegie Mellon.

Configuración de la máquina remota

A fin de facilitar el despliegue y depuración de aplicaciones en MacOS X, desde el entorno de Delphi XE2, tendremos que instalar en la máquina remota un software, llamado `paserver`, que actuará a modo de servidor respecto al entorno de Delphi. Éste, operando como cliente, enviará solicitudes a través de TCP/IP con comandos para la copia de archivos, disposición de puntos de parada, evaluación de expresiones y, en general, todas las operaciones habituales de depuración.

El servidor ha de encontrarse en marcha antes de que podamos, desde el entorno de Delphi, definir el perfil remoto para generar el proyecto en su versión para MacOS X. Por tanto el primer paso será instalar este software e iniciarlo en la máquina remota. Son las tareas que se describen en los siguientes apartados de este punto.

Instalación de paserver

En la subcarpeta `RAD Studio\9.0\PAServer` de la carpeta donde hubiésemos instalado Delphi, que por regla general suele ser `C:\Program Files\Embarcadero`, encontraremos un archivo llamado `setup_paserver.zip`. Hemos de copiarlo a la máquina remota[142], por ejemplo a nuestro escritorio en MacOS X. A continuación lo desempaquetamos y hacemos doble clic sobre el archivo `setup_paserver` para lanzar la instalación.

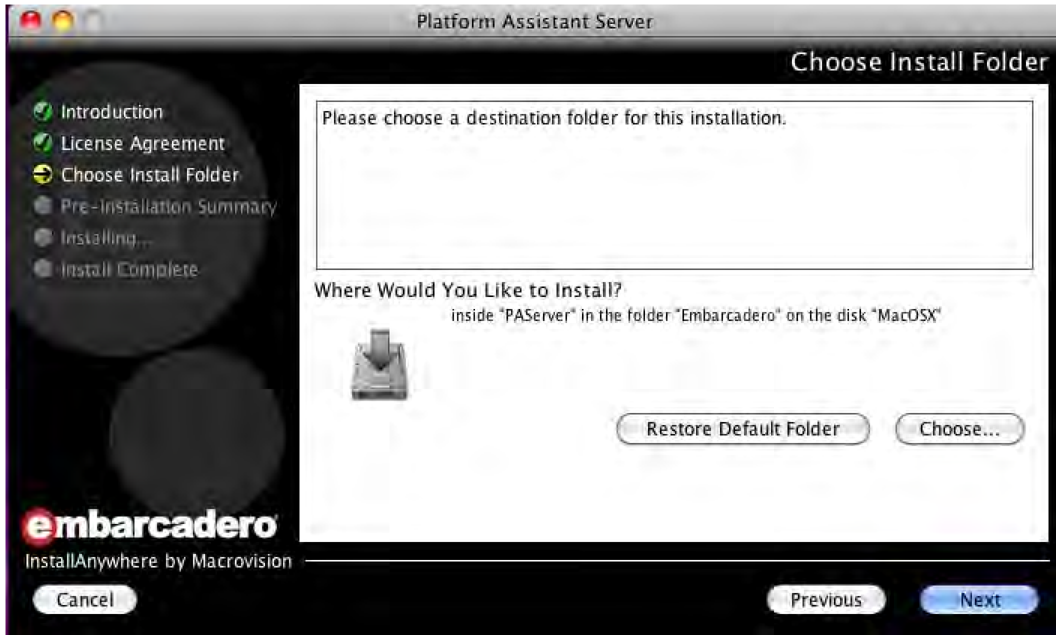
El proceso de instalación es prácticamente idéntico al descrito en el décimo capítulo, al instalar `paserver` en la máquina de 64 bits. Tras aceptar la licencia de uso deberemos indicar la carpeta donde se instalará el software,

---

141 En caso de que necesitemos incluir código en ensamblador en un proyecto que va a compilarse para MacOS X es recomendable leer las sugerencias dadas en http://blogs.embarcadero.com/eboling/2009/05/20/5607 .

142 Si tenemos activado en MacOS X el servicio Compartir Archivos, en Preferencias del Sistema>Compartir, podremos transferir archivos directamente desde la máquina Windows con el Explorador de Windows. Otra alternativa sería usar el servicio Sesión Remota y conectar mediante SSH.

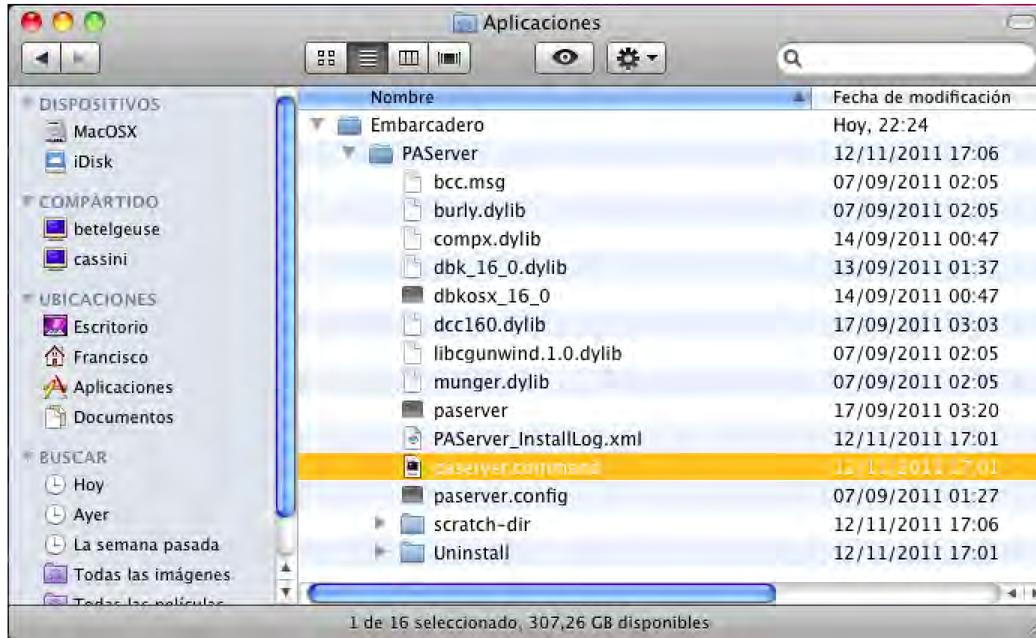
pudiendo aceptar la ruta por defecto o bien pulsar el botón Choose para seleccionar otra distinta (véase la imagen inferior).



Un clic en el botón Next nos llevará al paso en que se revisan los parámetros de instalación y un nuevo clic en el mismo botón al inicio del proceso de copia de archivos.

Una vez haya finalizado la instalación, y siempre que no hubiésemos cambiado la ruta de instalación, encontraremos en la carpeta `Applications` de nuestro usuario una subcarpeta llamada `Embarcadero/PAServer`, tal y como se aprecia en la imagen de la página siguiente. Dentro de ella está la carpeta `Uninstall`, con la utilidad `Uninstall PAServer` que facilita la desinstalación del software, y también la carpeta `scratch-dir` que será la que se use por defecto para la copia de archivos temporales desde Delphi.

> NOTA
>
> En caso de que surjan problemas durante la instalación podemos encontrar los detalles en el archivo `PAServer_InstallLog.xml`.



Puesta en marcha de paserver

Siempre que desde Delphi XE2 vayamos a realizar pruebas, depurar o desplegar un proyecto para MacOS X, deberemos tener en funcionamiento en la máquina remota el servidor `paserver`.

Los parámetros de configuración son exactamente los mismos que se explicaron en el décimo capítulo, por lo que no vamos a repetirlos aquí: puerto en el que se quedará a la escucha, directorio para el almacenamiento temporal, etc. A continuación nos ocupamos únicamente de los aspectos específicos.

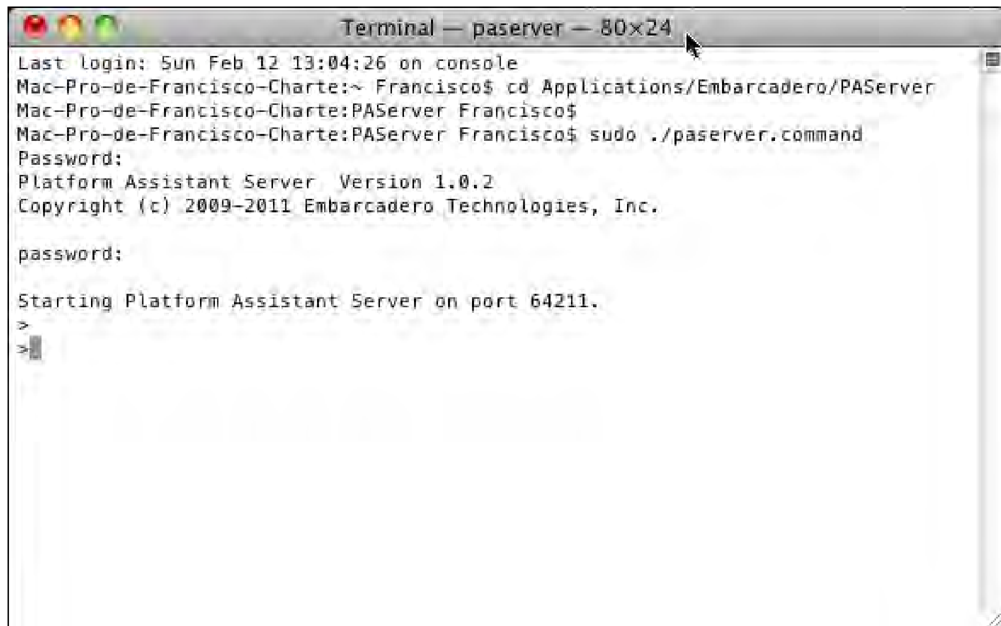
El archivo `paserver.config` almacena la configuración por defecto inicial, en la que se establece que el puerto será el `64211`. Podemos modificar este archivo si no queremos entregar en cada ejecución los parámetros en la línea de comandos. De hecho en la mayoría de los casos será lo que más cómodo nos resulte.

Antes de ejecutar `paserver` hemos de tener en cuenta dos detalles importantes:

- Es necesario actualizar la variable de entorno `DYLD_LI BRARY_PATH` para incluir en ella la ruta en la que se ha instalado `paserver`, a fin de que el sistema pueda localizar las bibliotecas de enlace dinámico. Sin esta información el servidor no podrá funcionar.
- Para poder acceder a operaciones de depuración en MacOS X es necesario contar con privilegios de administrador. Podemos otorgarlos desde un principio, ejecutando `paserver` como superusuario, o bien esperar a que se nos solicite la contraseña de administrador al inicio de la sesión de depuración.

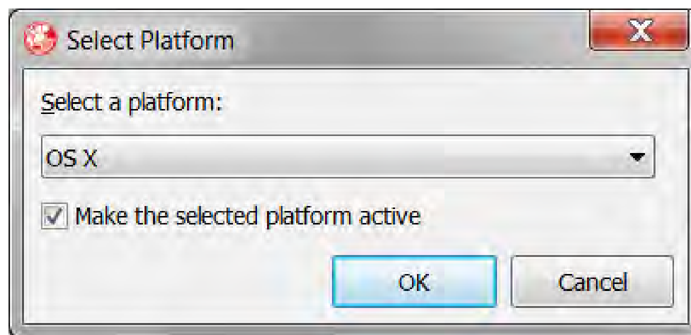
En la carpeta de instalación encontraremos un guión, con el nombre `paserver_command`, que se encarga de actualizar la variable de entorno `DYLD_LI BRARY_PATH` y a continuación ejecutar paserver. Podemos actualizar este *script*, incluyendo las opciones que necesitemos, y usarlo sistemáticamente cada vez que vayamos a trabajar con un proyecto en MacOS X.

En la imagen inferior puede verse cómo se utiliza el mencionado guión y cómo `paserver` solicita la contraseña de acceso al servidor.

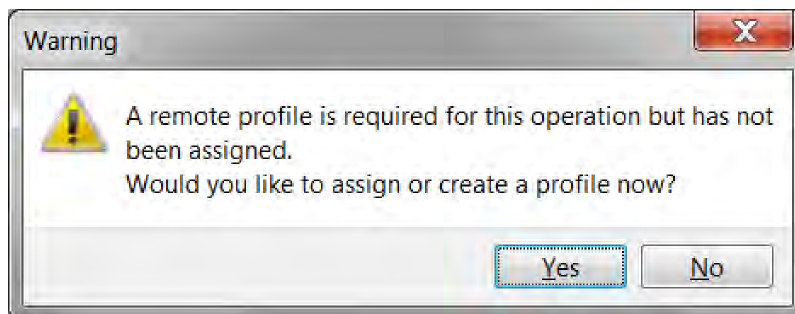


Configuración en la máquina local

Asumiendo que tenemos en desarrollo un proyecto para MacOS X que queremos ir probando y depurando, lo primero que hemos de hacer es agregar la plataforma OS X a la rama Target Platorms, en el Gestor de proyectos. Solamente tenemos que abrir el menú contextual de la citada rama, elegir la opción Add Plataform y seleccionar la lista desplegable del cuadro de diálogo Select Platform el elemento OS X (véase la imagen inferior).

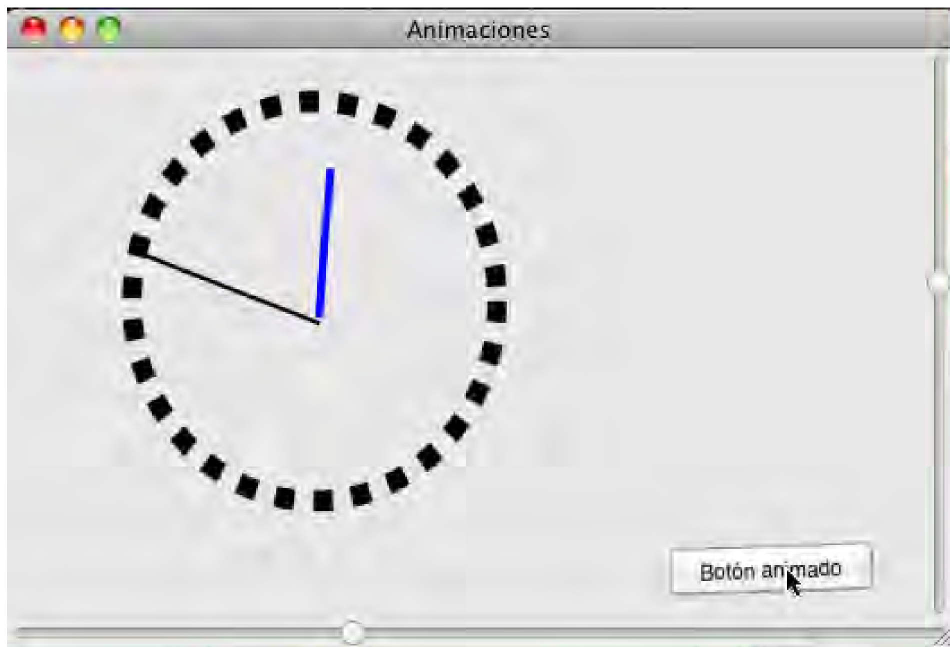


Si intentamos ejecutar el proyecto tras agregar la nueva plataforma Delphi nos informará de que no es posible, con un mensaje como el de la imagen siguiente, ya que no dispone de los datos necesarios para poder conectar con el servidor `paserver` que está ejecutándose en el Mac. Si hacemos clic en el botón Yes se abrirá el cuadro de diálogo de selección de perfil remoto, en el que podremos elegir uno existente o bien crearlo. El proceso es idéntico al descrito en el décimo capítulo.



Definido el perfil remoto que se usará, y siempre que la conexión entre Delphi XE2 y el servidor `paserver` no se encuentre con problemas (podríamos necesitar configurar el software cortafuegos que usemos en ambas máquinas), la ejecución del proyecto provocará su despliegue automático. Se transferirán al Mac los archivos necesarios, que se alojarán en la carpeta indicada por el parámetro `scratch-dir`, y se iniciará la aplicación. Veremos aparecer en el Dock de MacOS X el habitual icono de las aplicaciones Delphi (o el que hayamos configurado si lo hemos cambiado) y a continuación abrirse la ventana inicial del programa.

En la imagen inferior puede apreciarse el ejemplo del capítulo anterior que usaba varias animaciones y transformaciones. El efecto gráfico de mosaico inicial, el balanceo del `TButton`, la escala controlada por los `TTrackBar` y el movimiento de las manecillas del cronómetro son idénticos a los que teníamos en el proyecto original sobre Windows.



Tenemos un programa multiplataforma que podemos distribuir para los dos sistemas operativos más difundidos en la actualidad, partiendo de un mismo código fuente y sin tener que recurrir a componentes diferenciados. Es uno de los aspectos más interesantes de la biblioteca FireMonkey.

Despliegue de módulos redistribuibles

En caso de que el proyecto que vamos a desplegar en MacOS X utilice acceso a datos con dbExpress, o actúe como cliente DataSnap, además de los módulos del proyecto en sí también será necesario incluir aquellos redistribuibles que se precisen. Ésta es una tarea que ha de configurarse manualmente, ya que Delphi XE2 no la realiza de forma automática.

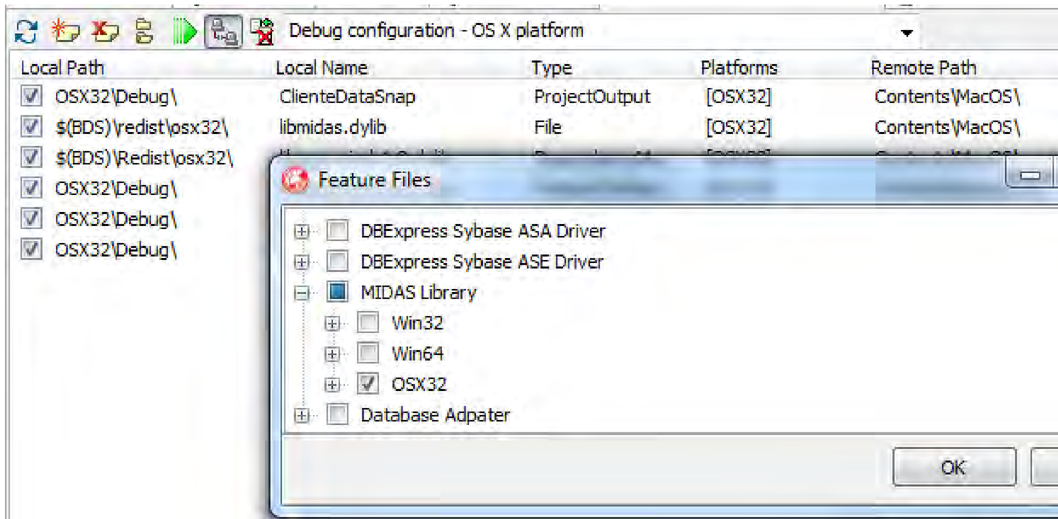
Tomemos la pareja de proyectos desarrollados inicialmente en el capítulo dedicado a DataSnap, el primero actuaba como servidor y mostraba las conexiones establecidas mientras que el segundo lo hacía como cliente con un `TStringGrid` en el que aparecía la información de la base de datos `MicroClassic`, aparte de un botón y un recuadro de texto que facilitaban el acceso a dos de los métodos expuestos por el servidor. Damos los siguientes pasos:

- Lanzamos la ejecución del servidor en la máquina donde estamos trabajando con Delphi, sin entrar en modo de depuración para poder continuar usando el entorno de desarrollo.
- Tomamos la dirección IP del servidor y la introducimos en la propiedad `Params` del componente `TSQLConnection`, a fin de que el cliente pueda localizar al servidor DataSnap.
- Agregamos al cliente la plataforma `OS X` y seleccionamos el perfil remoto correspondiente al Mac (o lo creamos si no teníamos definido ninguno).
- Lanzamos la ejecución del cliente que, tras ser compilado, desencadenará la copia de módulos a la máquina remota y el inicio de la ejecución.

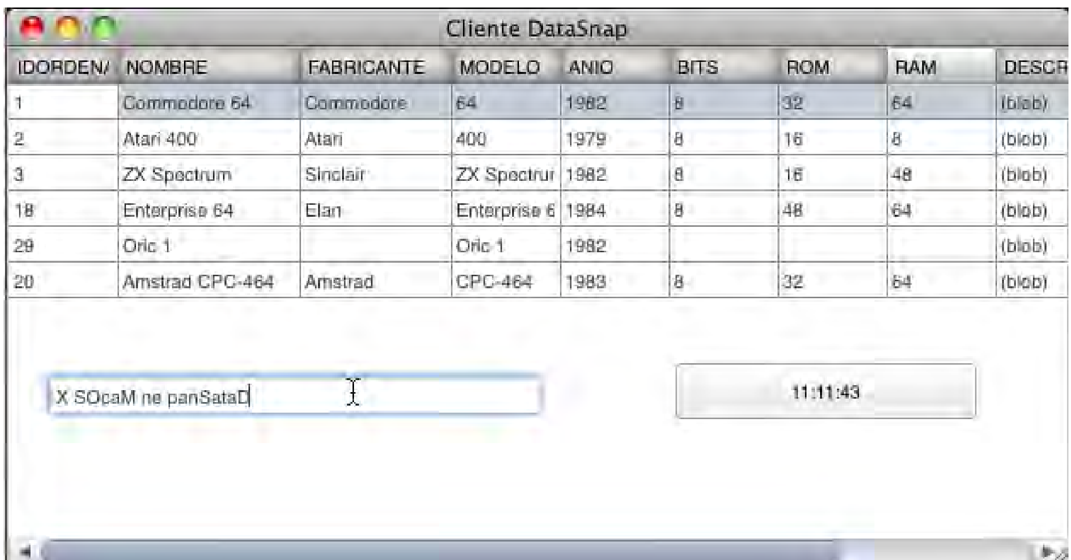
Casi de manera inmediata veremos aparecer mensajes de error y excepciones en el entorno de Delphi, mientras que en el Mac la ventana de la aplicación no llega a abrirse. Lo que ocurre es que la aplicación no puede encontrar en la máquina en que está ejecutándose las bibliotecas externas necesarias, en este caso concreto MIDAS. Lo mismo ocurriría al usar dbExpress para acceder directamente a cualquier RDBMS, aparte del software cliente también tendríamos que incluir los archivos redistribuibles correspondientes.

Usaremos la opción Project>Deployment para abrir la página de configuración de despliegue. Ésta fue descrita en el décimo capítulo, junto

con las opciones que ofrece. En nuestro caso (véase la imagen inferior) haremos clic sobre el botón Feature Files y marcaremos la versión para OSX32 de MIDAS Library.



Hecho este cambio en la configuración podremos ejecutar el proyecto sin problemas y, como puede verse en la siguiente imagen, usar el cliente DataSnap para acceder a los datos y servicios ofrecidos por el servidor.



Conclusión

La inclusión de un compilador específico para MacOS X en Delphi XE2, la creación de una versión multiplataforma de la RTL y el hecho de FMX también sea una biblioteca multiplataforma hace posible desarrollar aplicaciones para el sistema operativo de los Mac con gran sencillez, tal y como se ha mostrado en el presente capítulo.

No tenemos que aprender nada nuevo respecto a lo que habíamos aprendido en capítulos previos. El aspecto fundamental estriba en conocer la FMX a fin de poder diseñar interfaces de usuario que son independientes del sistema operativo. Hemos de tener en cuenta que ciertos tipos de proyectos existentes en Delphi, como los servidores DataSnap y aplicaciones web, no pueden ejecutarse en MacOS X, son soluciones pensadas únicamente para Windows, al menos por el momento, dada su dependencia de la VCL y los servicios del sistema operativo de Microsoft.

Se echa de menos que el compilador sea capaz de generar código de 64 bits para esta plataforma, especialmente cuando la última versión de MacOS X únicamente existe en versión de 64 bits. Es de esperar que sea uno de los objetivos satisfechos en una posterior actualización de Delphi.

A continuación

Tras el desarrollo para Win32, Win64 y MacOS X, en el próximo capítulo, último de esta cuarta parte del libro, abordaremos de manera más breve la creación de aplicaciones para otras plataformas.

Nos ocuparemos especialmente de iOS, el sistema operativo usado en productos de Apple como iPhone, iPad e iPod, introduciéndose también el acceso a servidores DataSnap desde aplicaciones para dispositivos móviles. Con ello concluiremos el recorrido por las capacidades multiplataforma de Delphi XE2.

Capítulo 13: Aplicaciones para otras plataformas

Este último capítulo de la cuarta parte del libro, dedicada al desarrollo de software multiplataforma, está en su mayor parte centrado en describir el proceso a seguir para crear aplicaciones para iOS, el sistema operativo más vendido de Apple muy por encima de OS X. El último punto explica, de manera más breve, cómo desarrollar software que, ejecutándose en otros sistemas operativos, consuma servicios implementados con Delphi.

Desarrollo para iOS en Delphi

Solamente en el año 2011 Apple ha vendido más dispositivos con el sistema operativo iOS que ordenadores con MacOS X en toda su historia. iOS representa una plataforma de altísimo interés para cualquier desarrollador de software, ya que a través de ella puede llegar a un mercado al que cada año se suman millones de usuarios.

Con Delphi XE2, y gracias a la naturaleza multiplataforma de la biblioteca de componentes FMX, podemos crear software para iOS. La integración de Delphi con dicho sistema, no obstante, es menor que con MacOS X y aun está muy lejos de la que existe con Windows. Es de esperar que este aspecto se mejore en futuras versiones de Delphi.

Compilación de código Delphi para iOS

Delphi XE2 incluye, tal y como se ha indicado en capítulos previos, un compilador para Win32, otro para Win64 y uno más para OS X. No existe, sin embargo, un compilador propio capaz de generar código para iOS. En realidad el problema no es el sistema operativo, sino la plataforma hardware. Win32, Win64 y OS X se ejecutan sobre una arquitectura x86/x64, la de los procesadores de Intel, pero los dispositivos con iOS tienen un procesador propio basado en una arquitectura ARM, muy distinta a la de Intel.

Por tanto para compilar el código de nuestros proyectos Delphi tendremos que recurrir a otro compilador, concretamente a una versión específica de FPC[143] que se distribuye junto con Delphi. No se trata de un compilador cruzado Windows-iOS sino MacOS X-iOS, es decir, que no compilaremos desde Windows para transferir después el código objeto a iOS sino que el compilador se instala en un Mac.

También hay disponible una versión de FPC que, ejecutándose sobre MacOS X, genera código para este sistema operativo. Podría utilizarse de forma conjunta con la FMX para desarrollar software para ese sistema, sin Delphi.

---

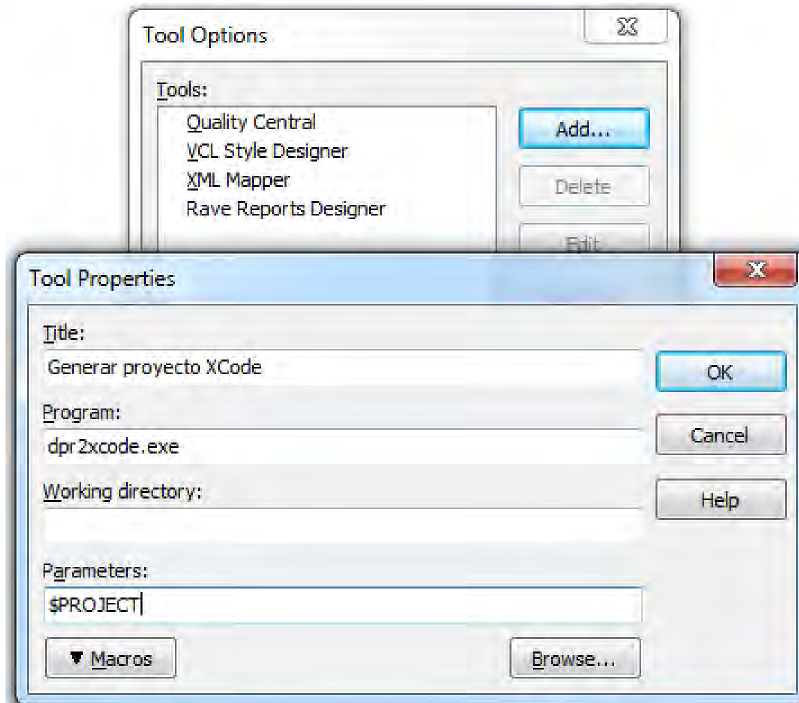
143 *FreePascal Compiler*. Es un compilador de código abierto disponible para múltiples plataformas y que reconoce la mayor parte de la sintaxis de Delphi. El sitio oficial del proyecto es http://sourceforge.net/projects/freepascal.

Conversión de proyectos Delphi a Xcode

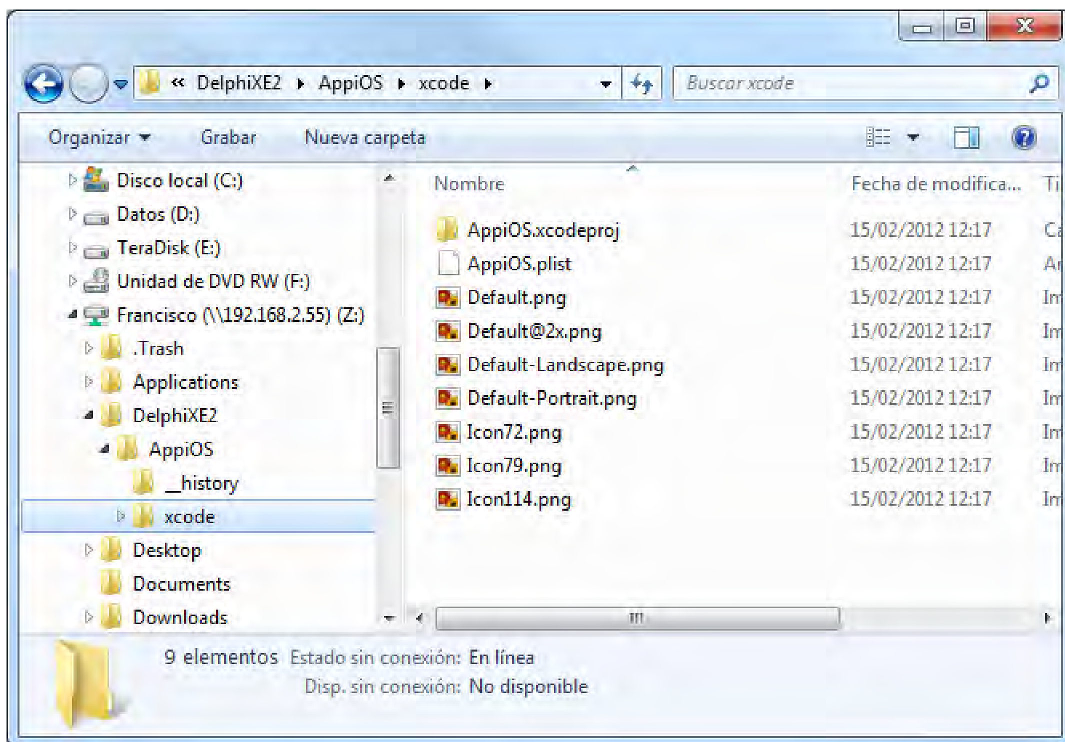
Puesto que vamos a tener que compilar en MacOS X, normalmente varios módulos a los que se agregarán la biblioteca FMX y posiblemente otras, lo más cómodo será recurrir a una herramienta de desarrollo específica para MacOS X. El entorno por excelencia para este sistema es Xcode, de la propia Apple, que deberemos instalar tal y como se indica más adelante.

Con el fin de hacernos la transición más suave, Delphi incorpora una utilidad que se encarga de convertir los proyectos Delphi XE2 al formato de Xcode, introduciendo además los cambios necesarios en el código de los módulos para adaptarlos a los aspectos específicos de iOS. Dicha utilidad se llama `dpr2xcode.exe`.

Aunque podríamos ejecutar esta utilidad desde la línea de comandos cada vez que lo necesitemos, nos será mucho más cómodo agregarla como una opción más al menú Tools de Delphi. Para ello abrimos el cuadro de diálogo Tool Options (véase imagen inferior), hacemos clic en Add y configuramos las propiedades incluyendo la variable `$PROJECT` como parámetro.



Ahora podremos generar a partir de un proyecto Delphi el correspondiente proyecto Xcode sin más que elegir la nueva opción. En ese momento se creará, dentro de la carpeta del proyecto, una subcarpeta con el nombre `xcode`. En ella, como puede apreciarse en la imagen siguiente, se incluyen una serie de iconos, el archivo de proyecto Xcode y una subcarpeta más con los módulos de código.



> NOTA
> Normalmente tendremos que usar la opción que ejecuta `dpr2xcode.exe` la primera vez que vayamos a desplegar el proyecto en iOS, obteniendo el proyecto Xcode a abrir en el Mac, repitiendo la operación únicamente cuando se agreguen o eliminen módulos del proyecto. No es necesario si lo único que hemos hecho es cambiar elementos de la interfaz de usuario o modificar el código de implementación propio.

Preparación de la máquina MacOS X

Como acaba de indicarse, para crear software para iOS con Delphi vamos a precisar una máquina con MacOS X en la que, además, hemos de instalar varias herramientas adicionales. Veamos cuáles serían los pasos a seguir para tener nuestro Mac con todo lo necesario.

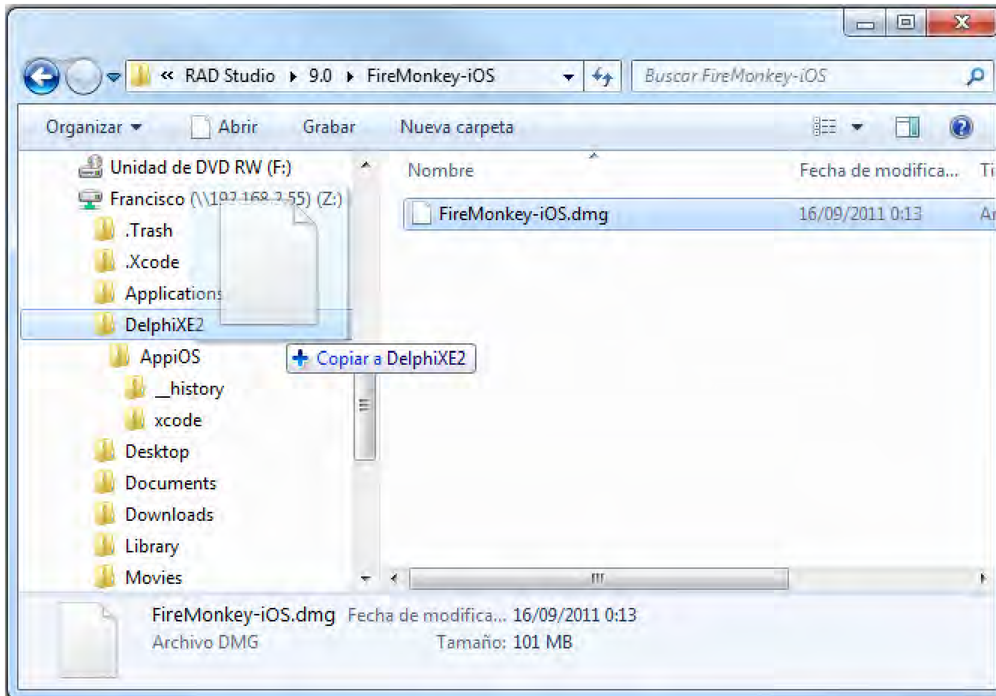
Lo primero será descargar e instalar Xcode, producto que encontraremos tanto en la Mac App Store como en el sitio de Apple para desarrolladores (véase la imagen inferior). La versión actual de Xcode es la 4.X, pero también nos serviría la última actualización de la rama 3.X.

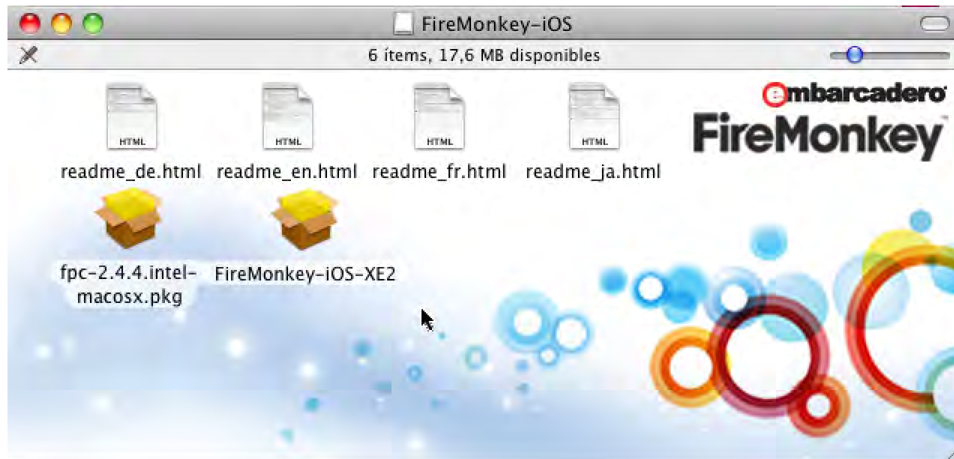
El paquete de instalación de Xcode también incluye el iOS SDK o paquete de desarrollo para iOS que, obviamente, necesitaremos para poder generar proyectos para dicho sistema operativo. Una vez descargada la imagen, el proceso de instalación será, como es habitual en OS X, muy simple y

básicamente no habrá más que aceptar la licencia de uso e indicar el destino de instalación, tras lo cual podremos elegir entre instalar o no ciertos componentes como el SDK para Mac que, en este caso concreto, no nos interesa. El software precisará aproximadamente unos 10 gigabytes de espacio en disco.

A continuación debemos instalar el compilador FPC y la biblioteca FMX en su versión para iOS. Ambos elementos están alojados en un archivo que encontraremos en la subcarpeta `FireMonkey-iOS` de la carpeta donde tuviésemos instalado Delphi XE2. Es un archivo imagen, de tipo `.dmg`, que copiaremos al Mac tal y como se muestra en la primera figura de la página siguiente.

Al abrir esta imagen en MacOS X veremos un conjunto de elementos (véase la segunda imagen de la página siguiente), entre ellos dos con extensión `.pkg`. El primero es el instalador de FPC y el segundo el de FireMonkey. Hacemos doble clic sobre el primero y, finalizado el asistente de instalación, repetimos operación con el segundo.





Con esto ya tenemos instalado en MacOS X todo lo necesario para crear aplicaciones para iOS mediante FPC y FMX.

Compartir carpetas y escritorio

Al crear aplicaciones para iOS con Delphi XE2 y Xcode tendremos que trabajar casi simultáneamente con ambas herramientas. Con la primera diseñaremos las interfaces de usuario y escribiremos el código, pudiendo también realizarse pruebas y depuración pero ejecutando el proyecto como si fuese una aplicación Windows. Con la segunda compilaremos y depuraremos en un entorno más real, ya sea mediante un emulador de iPhone/iPad o directamente sobre un dispositivo físico.

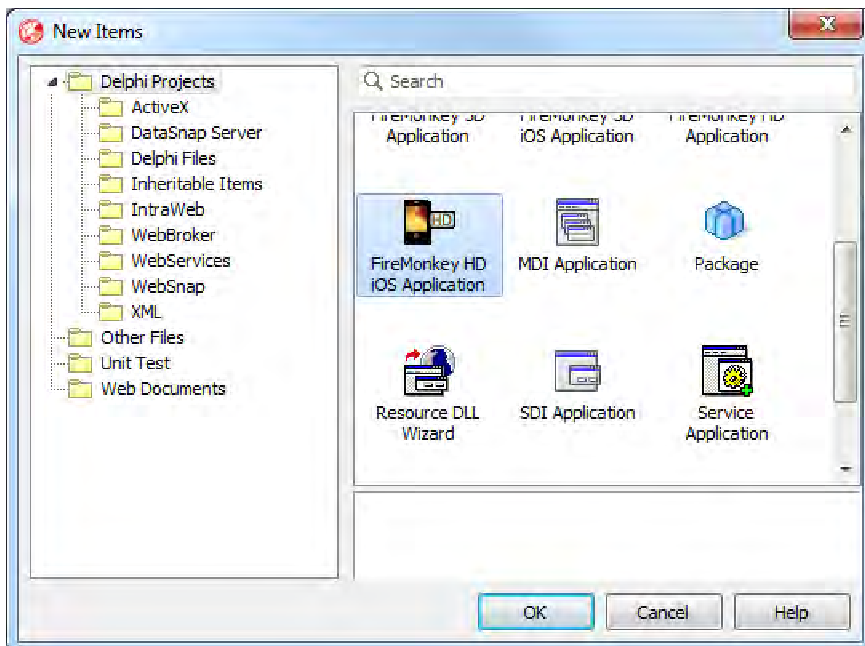
La tarea nos resultará más sencilla si activamos en MacOS X las opciones Compartir Archivos y Compartir Pantalla. Ambas se encuentran en el cuadro de diálogo Compartir mostrado en la imagen inferior. Tomaremos nota del nombre de la máquina o de su dirección IP, datos necesarios para acceder desde la máquina Windows en que se ejecuta Delphi XE2.

Habilitadas esas funciones en MacOS X, en la máquina Windows buscaremos la carpeta compartida y la conectaremos a una unidad de red. De esta forma podremos usarla como si fuese una unidad de almacenamiento más, lo cual nos permitirá guardar los proyectos de Delphi en el Mac de manera inmediata.

Por último instalaremos en Windows el software TightVNC o similar, un cliente de VNC mediante el que accederemos al escritorio de MacOS X desde la máquina de desarrollo, evitando así que tengamos que cambiar continuamente entre una y otra.

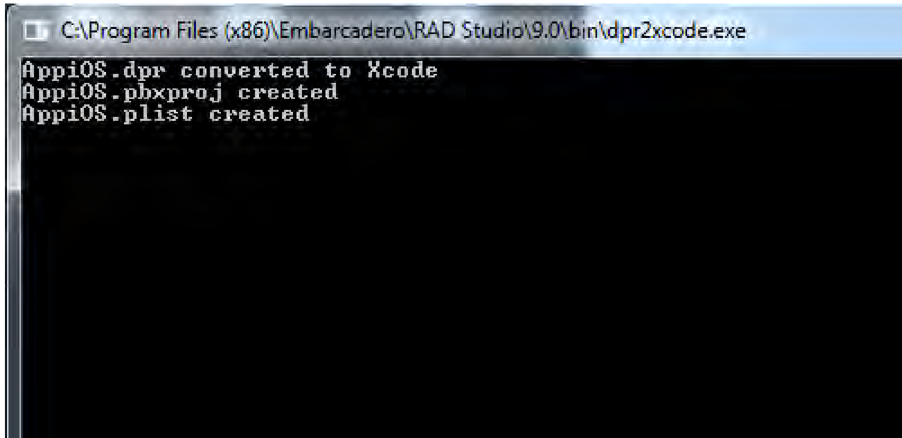
Creación de un proyecto para iOS en Delphi

Teniendo el Mac ya preparado estamos en disposición de crear un proyecto para iOS con Delphi XE2. Para ello usaremos la opción FireMonkey HD iOS Application si queremos un programa con una interfaz de usuario clásica, como se ve en la imagen inferior, o bien la opción 3D si pretendemos generar escenas 3D como las descritas en el undécimo capítulo.



Lo primero que apreciaremos será que el aspecto del formulario inicial tiene un aspecto peculiar, con unas proporciones y un fondo que recuerdan a la pantalla del iPhone, incluyendo el botón en la parte inferior. Vamos a insertar en él un `TImage` con un efecto de pixelización y una animación asociados, una lista desplegable que controlará la intensidad de uno de esos efectos y un `TButton` que servirá para iniciar/detener el `TFloatAnimation`. Éste se vinculará a la propiedad `RotationAngle` del `TImage`. En la imagen inferior puede apreciarse el aspecto de la interfaz en la fase de diseño, incluyendo las jerarquía de componentes.

A la hora de guardar el proyecto seleccionaremos como destino una carpeta en la unidad compartida con el Mac, de esta forma la herramienta `dpr2xcode.exe`, que hemos de ejecutar a continuación, almacenará el proyecto de Xcode en la ruta adecuada. La misma utilidad nos indicará las acciones que ha llevado a cabo, tal y como se muestra en la imagen siguiente.



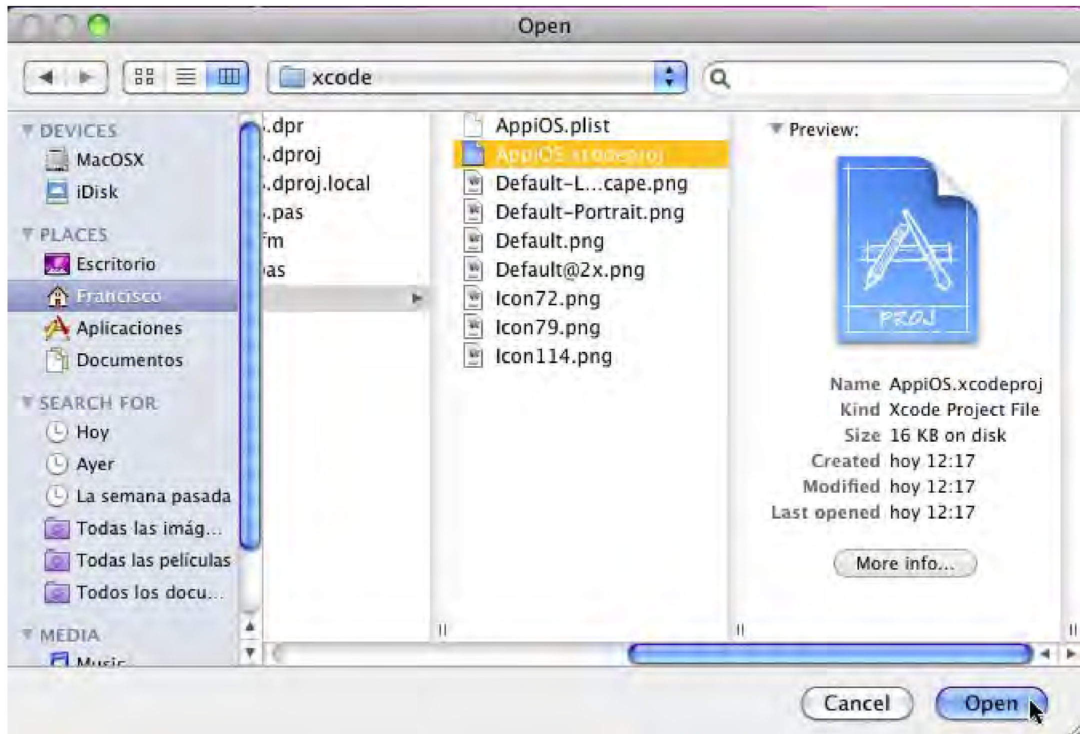
Podemos ejecutar el proyecto directamente desde Delphi a fin de comprobar que su funcionamiento es el que esperamos:

- La imagen que ocupa la parte central de la ventana gira continuamente, giro que podemos detener y reanudar con el botón dispuesto debajo. Éste sencillamente cambia la propiedad `Enabled` del objeto `TFloatAnimation`.
- Al seleccionar uno de los valores de la lista desplegable, que son 10, 50, 100, 500 y 1000, se modifica el grado de pixelización de la imagen. Esto se consigue asociando al evento `OnChange` del `TComboBox` una simple sentencia de asignación, convirtiendo el texto del elemento elegido en la lista en un número que podamos asignar a la propiedad `BlockCount` del componente `TPixelateEffect`.

Este comportamiento debería ser exactamente el mismo que obtuviésemos al compilar el código con FPC, con la versión de FireMonkey para iOS, tanto si usamos un emulador en Xcode como si desplegamos el código objeto resultante en un dispositivo real.

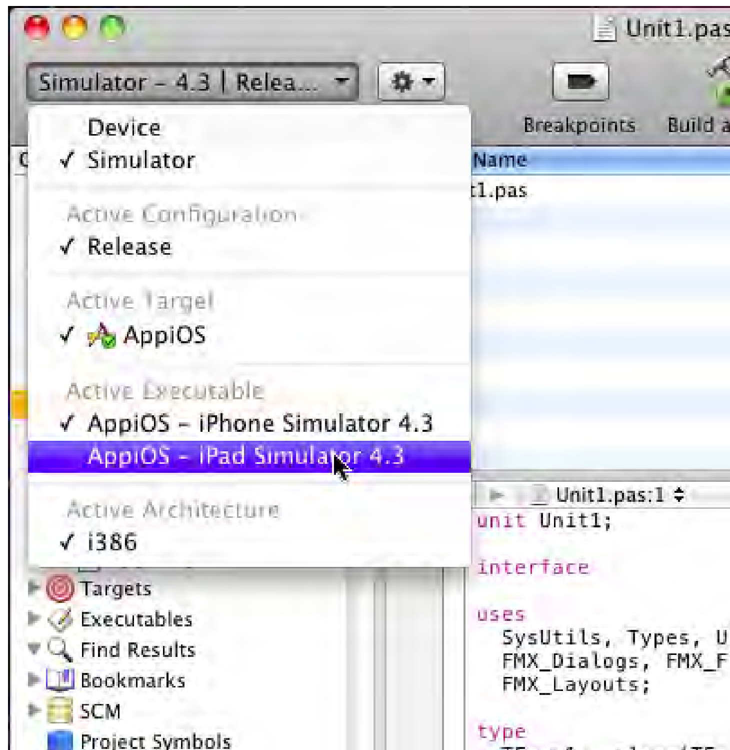
Ejecución del proyecto en Xcode

Asumiendo que el proyecto se ha almacenado en la ruta adecuada, lo encontraremos en el Mac y en la carpeta hallaremos un archivo con el nombre original del proyecto y extensión `.xcodeproj` (véase la imagen inferior). Hacemos doble clic sobre él para abrirlo en Xcode. También podemos ejecutar esta herramienta y a continuación usar el botón Open Other, con el mismo resultado.



Aunque desde Xcode podemos editar el código de los módulos del proyecto, y dado que están almacenados en una carpeta compartida Delphi obtendría los cambios de manera inmediata, generalmente nos será más cómodo hacerlo desde Delphi ya que su editor es mucho más potente y adaptado a un lenguaje de programación específico. Además en Xcode no tenemos el diseñador de formularios, por lo que cualquier cambio en la interfaz habría que realizarlo de forma manual. Con Delphi podemos hacerlo en el diseñador y simplemente guardar los cambios, obteniéndolos en Xcode.

Habitualmente el proceso de prueba y depuración en Xcode lo haremos con un emulador. En la parte superior izquierda del entorno hay una lista desplegable que, como puede verse en la imagen siguiente, nos permite elegir el dispositivo objetivo. Las dos primeras opciones: Device y Simulator, determinarán el objetivo del ejecutable.



> ADVERTENCIA
>
> Para poder ejecutar en un dispositivo físico y subir nuestra aplicación al App Store tendremos que registrarnos en el *iOS Developer Program*. Éste tiene un coste de 99 dólares anuales. Puedes encontrar más información en https://developer.apple.com/programs/ios.

Seleccionado el dispositivo de destino no tenemos más que lanzar la ejecución, con o sin depuración, y ver el resultado. En la imagen de la página siguiente puede verse el proyecto funcionando en el emulador de iPhone.

Carrier
1:09 PM
Ajuste mosaico
100
10
50
100
500
1000
Rotar imagen

Desarrollo para otras plataformas

Aparte de las tratadas en el anterior punto de este capítulo y en capítulos previos, que son iOS, MacOS X, Win32 y Win64, Delphi XE2 no está preparado para desarrollar software para ninguna otra plataforma[144]. Al menos por el momento.

Sí que lo está, sin embargo, para ayudarnos a crear aplicaciones que desde otros sistemas operativos accedan a un servidor DataSnap creado con Delphi XE2, concretamente un servidor que exponga sus servicios mediante las técnicas REST descritas en el noveno capítulo. Ése es el objetivo de los denominados *Mobile Connectors* y una herramienta llamada *proxy dispatcher* que puede incorporarse en el servidor DataSnap o bien ser ejecutada de manera independiente.

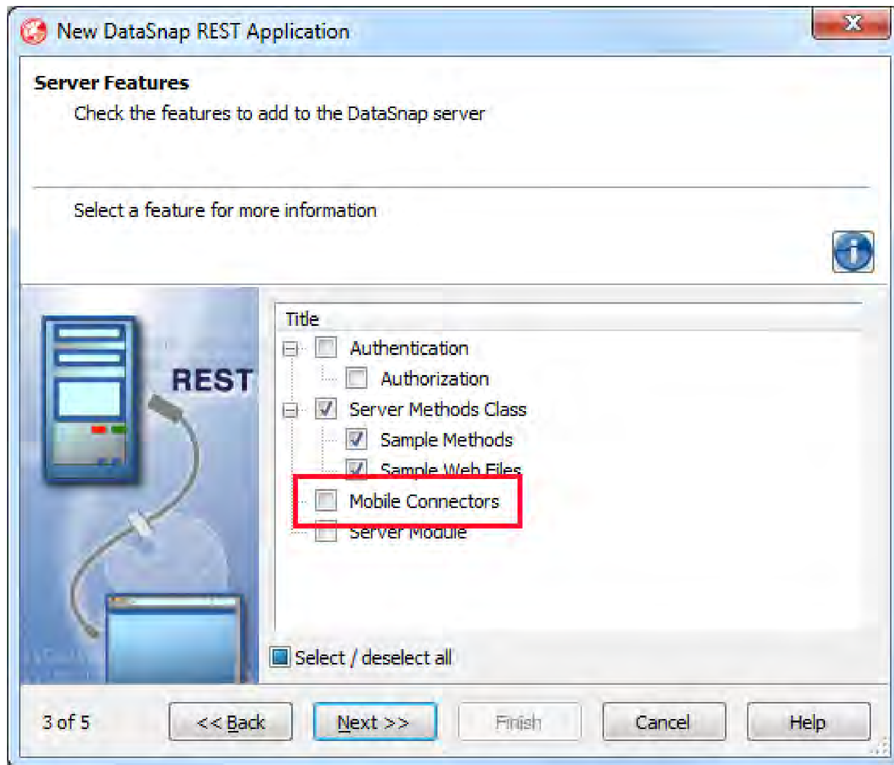
Dependiendo de la plataforma objetivo tendremos que usar un lenguaje u otro y elegir una herramienta o entorno de desarrollo. Para Android, por ejemplo, lo habitual es trabajar con Java desde Eclipse, mientras que para Windows Phone el lenguaje será C# y el entorno Visual Studio. No vamos a entrar aquí en las peculiaridades de cada caso: lenguaje de programación, entorno de desarrollo, bibliotecas de clases y componentes, etc., detalles todos ellos que encontraremos en la bibliografía específica para cada plataforma. Nos ocuparemos exclusivamente del proceso que nos permitirá obtener el *proxy* necesario para acceder al servidor DataSnap desarrollado con Delphi.

DataSnap Mobile Connectors

Al iniciar un nuevo proyecto de tipo servidor DataSnap REST el asistente, en uno de sus pasos, nos permite seleccionar los componentes que deseamos incluir. Una de las opciones existentes es Mobile Connectors (véase imagen de la página siguiente), desmarcada por defecto, cuya función es agregar los elementos necesarios para obtener del servidor los módulos de código que faciliten el acceso a sus servicios desde diferentes plataformas móviles.

---

144 Sí que es posible hacerlo con otros productos de la familia XE2, como RadPHP XE2 para la creación de aplicaciones Android o Prism XE2 que contempla el desarrollo de proyectos para Windows Phone a través de la plataforma .NET.



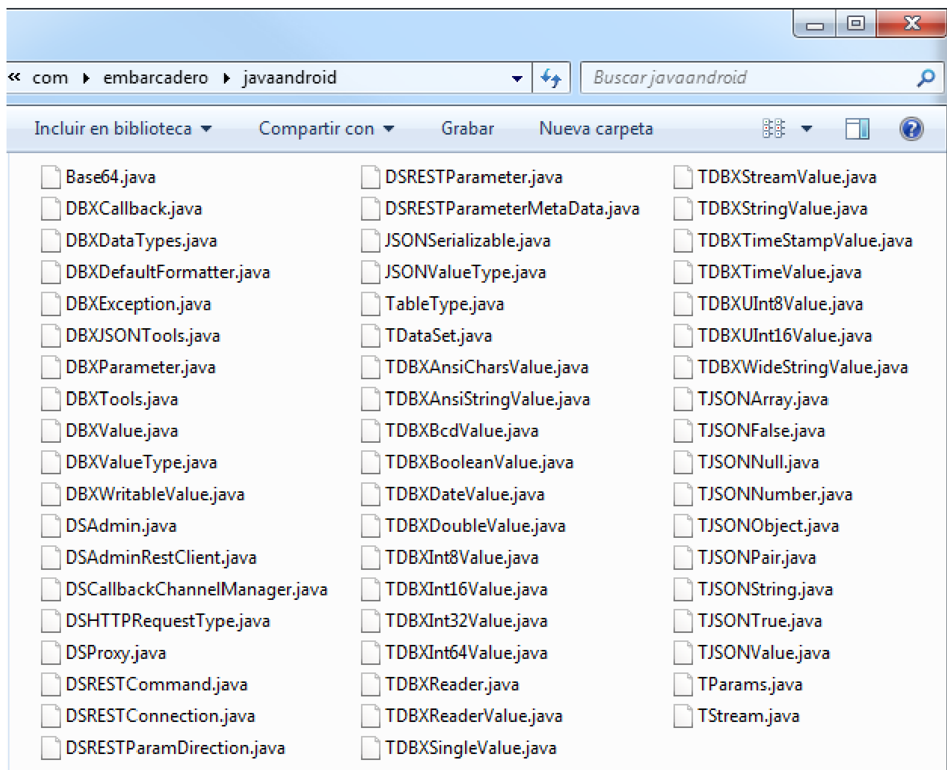
Si activamos dicha opción, se agregarán a la cláusula `uses` del módulo derivado de `TWebModule` una referencia a varios módulos con nombres del tipo `Datasnap.DSProxyLengPlat`, siendo `Leng` un lenguaje de programación: Java, ObjectiveC o Csharp y `Plat` una plataforma: Android, BlackBerry, iOS o Silverlight (Windows Phone). Así tenemos, por ejemplo, el módulo `Datasnap.DSProxyJavaAndroid`, encargado de facilitar el conector para Android en lenguaje Java.

En el mismo módulo del proyecto se han introducido además los componentes `TDSProxyGenerator` y `TDSProxyDispatcher`, cuya finalidad se describió también en el noveno capítulo. El primero genera el *proxy* en el lenguaje y para la plataforma apropiada, usando para ello objetos definidos en los módulos `Datasnap.DSProxyXXX`, mientras que el segundo se encarga de facilitarlos a los clientes que los soliciten. A los contextos `rest/`, `cache/` y `datasnap/` del `TDSHTTPWebDispatcher` (véase el capítulo 9) estos componentes añaden otro con la ruta `proxy/`.

Obtención de un conector concreto

Teniendo el servidor DataSnap en funcionamiento, podemos ejecutarlo desde el entorno de Delphi, podemos obtener el conector para la aplicación móvil directamente desde el navegador. Desde la misma máquina usaríamos el URL `http://localhost:8080/proxy/CONECTOR.zip`, cambiando `CONECTOR` por una de las siguientes cadenas: `java_android`, `java_blackberry`, `csharp_silverlight` u `objective_ios42`.

Lo que obtenemos es un archivo comprimido en el que están alojados todos los módulos de código que componen el conector. En la imagen inferior aparecen los correspondientes al conector Java para Android.



Es el componente `TDSProxyDispatcher` el que se encarga de combinar los módulos comunes del conector con los específicos para acceder a los servicios de nuestro servidor DataSnap, empaquetarlos en el archivo `.zip` y entregarlos como respuesta a la solicitud recibida.

La mayor parte de los módulos de código del conector son definiciones de clases que reproducen los tipos de datos habituales en Delphi, por ejemplo los correspondientes a DBX, a fin de facilitar la transmisión de información entre el servidor DataSnap y la aplicación que se ejecutará en el dispositivo móvil. También hay clases que tienen una función similar a la de los componentes Delphi que usaríamos para crear un cliente DataSnap, como `DSRESTConnection` o `DSRESTCommand`.

En lugar de usar el navegador para obtener el paquete con el código del conector, posiblemente sea la manera más fácil de hacerlo, podemos recurrir a una utilidad que se utiliza desde la línea de comandos y de la cual se facilitan dos versiones: una en forma de ejecutable Windows y otra para la plataforma Java. Esta última puede ser ejecutada en cualquier sistema operativo en el que tengamos instalado el JRE de Java. Ambas las encontraremos en la subcarpeta `bin` de la carpeta donde tengamos instalado Delphi XE2, la primera con el nombre `Win32ProxyDownloader.exe` y la segunda `JavaProxyDownloader.jar`. Las dos aceptan los mismos parámetros:

- `-host`: Indica el nombre o dirección IP donde está ejecutándose el servidor DataSnap, incluyendo el puerto. Por defecto toma el valor `localhost:8080`, por lo que puede omitirse si el servidor está en la misma máquina.
- `-language`: Selecciona la plataforma para la que se quiere obtener el conector. Será uno de los siguientes cuatro valores: `java_android`, `java_blackberry`, `csharp_silverlight` u `objective_ios42`.
- `-output`: Establece la ruta de destino en la que se almacenará el conector obtenido desde el servidor. Si se omite se usa la carpeta actual.

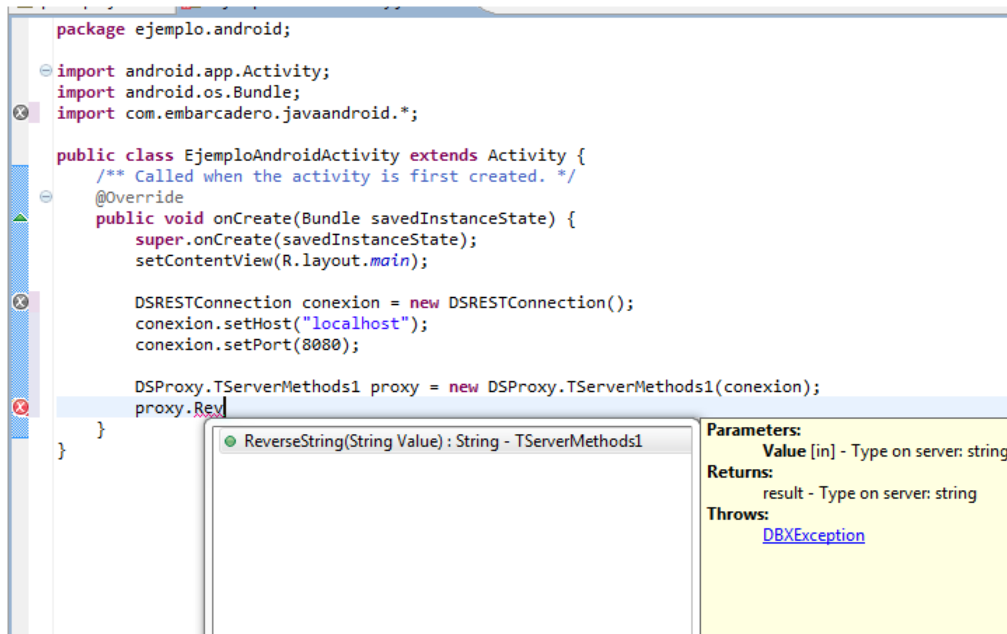
> NOTA
>
> A diferencia de lo que ocurre al solicitar el conector desde el navegador, obteniéndose un archivo `.zip`, estas herramientas se encargan de desempaquetar el contenido del archivo comprimido y alojarlo en la ruta indicada.

Uso del conector

Obtenido el paquete con el código del conector, lo agregaremos al proyecto de aplicación para el dispositivo móvil que corresponda y usaremos sus clases para configurar la conexión de acceso al servidor DataSnap, obtener una referencia al *proxy* que facilita el acceso a sus servicios e invocar a los métodos que precisemos.

Para la conexión crearemos un objeto `DSRESTConnection` que configuraremos con métodos como `setHost` y `setPort`. En el módulo `DSProxy` encontraremos las clases que actúan como *proxy* y que exponen los servicios implementados en el servidor, por ejemplo `TServerMethods1.ReverseString`.

La imagen inferior, correspondiente al editor de la herramienta de desarrollo Eclipse, muestra cómo se ha incluido una referencia al contenido del paquete `com.embarcadero.javaandroid` para poder usar las clases que acaban de citarse.



Conclusión

Como hemos comprobado en éste y los capítulos previos de la cuarta parte del libro, Delphi XE2, a pesar de seguir siendo un entorno que se ejecuta exclusivamente en Windows, cuenta con todo lo necesario para facilitar el desarrollo de aplicaciones multiplataforma: compiladores específicos para Win32, Win64 y OS X, una biblioteca de componentes multiplataforma y herramientas adicionales como el conversor de proyectos Delphi a Xcode o los conectores DataSnap que permiten consumir servicios DataSnap desde aplicaciones móviles.

Ofrecer un producto con esta capacidad abre a Delphi otros segmentos de mercado que antes le estaban vetados, con desarrolladores que tienen como objetivo no solamente Windows sino también MacOS X y los cada vez más difundidos sistemas operativos de los dispositivos móviles. Es una vía en la que seguramente se profundice en versiones posteriores de Delphi, con nuevos compiladores y una mayor integración de las distintas funciones de desarrollo respecto a esas otras plataformas.

A continuación

El próximo capítulo da paso a la quinta y última parte de este libro, compuesta únicamente de dos capítulos, y en él se tratará la confección de informes mediante FastReport, otra de las novedades incluidas en Delphi XE2.

Apartado V: Informes y documentación

La quinta y última parte del libro, la más breve también, describe algunas funciones nuevas incorporadas en Delphi XE2 pero que no forman parte del producto en sí sino que son complementos de terceros, como son el diseño de informes con FastReport o la documentación de código mediante Documentation Insight.

- **Capítulo 14: Informes con FastReport**
- **Capítulo 15: Documentation Insight**

Capítulo 14: Informes con FastReport

A lo largo de su historia Delphi ha ido incorporando distintos grupos de componentes y herramientas con el mismo objetivo: facilitar la generación de informes desde una aplicación. En ocasiones eran productos del propio fabricante y otros de terceros. Algunos ejemplos de ello son QuickReports, ReportSmith, ReportBuilder o Rave Report.

Delphi XE2 incluye una nueva opción a añadir a la lista anterior: FastReport. Se trata de un producto desarrollado por otra empresa, como QuickReports o Rave Report, y a pesar de contar con múltiples ventajas, especialmente respecto a QReport, también tiene algunas limitaciones. En este capítulo se abordan los aspectos más destacables de FastReport y se explica cómo usar estos componentes para crear un informe sencillo.

Componentes FastReport

La biblioteca de componentes FastReport está diseñada para trabajar conjuntamente con la VCL, ya sea en aplicaciones Win32 o Win64. Ésta es seguramente la principal limitación: no podemos incluir componentes FastReport en una aplicación FMX, por lo que la generación de informes para MacOS X no resulta posible y tendríamos que recurrir a otra solución.

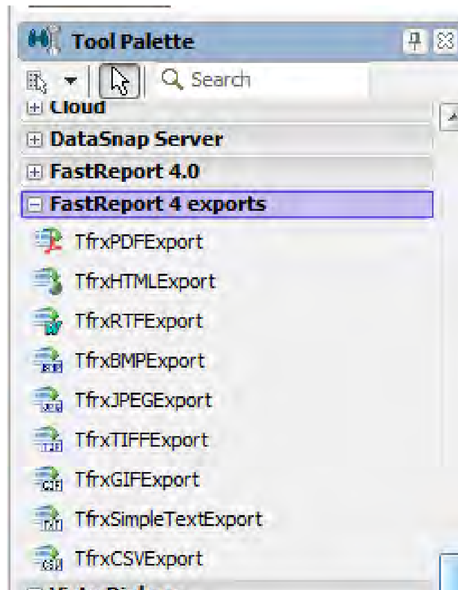
> ADVERTENCIA
>
> A pesar de esta limitación las páginas correspondientes a componentes FastReport en la Paleta de herramientas no se ocultan automáticamente al trabajar en un proyecto para MacOS X o iOS, y podemos incluso introducirlos en el formulario. Al intentar compilar, sin embargo, surgirán los problemas.

En la página FastReport 4.0 de la Paleta de herramientas están los componentes fundamentales, entre ellos el encargado de almacenar la configuración del informe y facilitar su edición y generación: `TfrxReport`; el que actúa como intermediario de acceso a datos: `TfrxDBDataset`, y el que facilita la composición de previsualizaciones a medida: `TfrxPreview`.

Todo lo necesario para componer un informe, guardarlo en el proyecto o un archivo externo, mostrarlo en pantalla e imprimirlo es el componente `TfrxReport`. En él se incluye la herramienta que usaremos en la fase de diseño para configurar informes, así como varios asistentes para su creación. Mediante métodos como `ShowReport` y `Print` podemos mostrar e imprimir el informe a demanda, desde nuestro código, ajustando los parámetros necesarios con las propiedades `ReportOptions`, `PrintOptions` y `PreviewOptions`.

Habitualmente los datos a usar en un informe procederán de una base de datos y los obtendremos de un `TClientDataSet`, `TSQLDataSet` o similar. En cualquier caso usaremos un componente `TfrxDBDataSet` para enlazar esos datos con el `TfrxReport`. La propiedad `DataSet` del primero apuntará al componente de base de datos, mientras que la propiedad del mismo nombre del `TfrxReport` hará referencia al `TfrxDBDataSet`. En caso de que los datos procedan de otra fuente distinta normalmente usaremos un `TfrxUserDataSet`, respondiendo a eventos como `OnGetValue` y `OnNext`.

Además de mostrarlo en pantalla o imprimirlo, un informe también puede ser exportado a diferentes formatos de archivo. Los componentes de la página FastReport 4 exports que pueden verse en la imagen inferior son los encargados de ocuparse de esa tarea.



El proceso es muy sencillo y se reduce a seleccionar el componente adecuado, por ejemplo `TfrxPDFExport` para exportar en formato PDF, configurar mediante sus propiedades el resultado que quiere obtenerse y, finalmente, invocar al método `Export` de `TfrxReport` facilitando como parámetro el componente de exportación que puede ser cualquiera de los de la página indicada antes.

NOTA

En la Paleta de herramientas también encontraremos una página llamada FastScript, con componentes desarrollados por la misma empresa que FastReport pero cuya finalidad es dotar a cualquier aplicación de un núcleo de *scripting*, de forma que puedan crearse guiones en distintos lenguajes para controlar la aplicación de manera programática.

Diseño de un informe

Teniendo una visión general de los componentes fundamentales, veamos cuáles serían los pasos a seguir para diseñar un informe con FastReport, usando información procedente de la base de datos MicroClassic, y mostrarlo desde una aplicación facilitando su impresión.

El objetivo es obtener un listado con los datos básicos de los ordenadores existentes en la colección: nombre, año de lanzamiento, memoria RAM y ROM y estado de conservación, agrupándolos según el fabricante a que pertenecen.

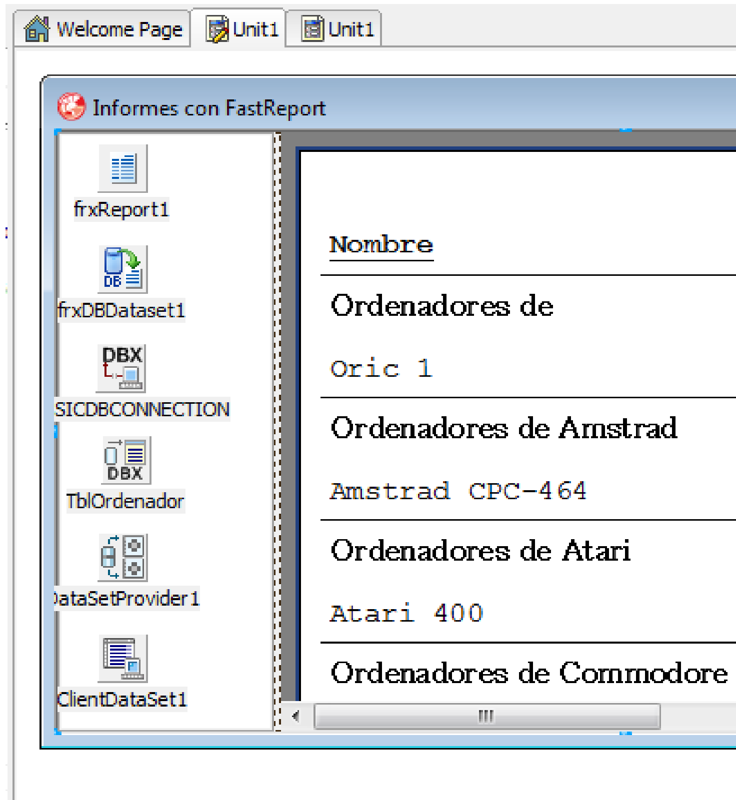
Conexión con los datos a usar

El primer paso será incluir en el formulario todos los componentes precisos para facilitar la conexión entre el informe y los datos que van a utilizarse. La lista de componentes, y sus relaciones, será la siguiente:

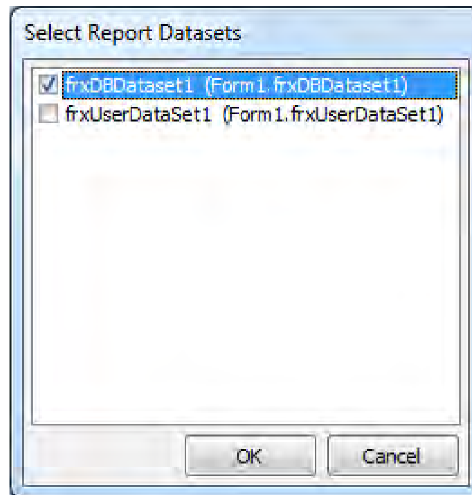
- `TSQLConnection`: Modificamos su propiedad `ConnectionName` para usar la conexión con Firebird correspondiente a la base de datos creada en capítulos previos, al tratar los componentes dbExpress.
- `TSQLDataSet`: Lo conectamos con el anterior, seleccionamos `ctTable` en la propiedad `CommandType` e introducimos en la propiedad `CommandText` el nombre de la única tabla existente: `TblOrdenador`.
- `TDataSetProvider`: La generación del informe no puede realizarse sobre un *dataset* unidireccional, por lo que necesitamos un proveedor y un `TClientDataSet` para almacenar los datos localmente y facilitar la navegación. Asignamos a su propiedad `DataSet` una referencia al `TSQLDataSet`.
- `TClientDataSet`: Lo conectamos con el anterior mediante la propiedad `ProviderName`.
- `TfrxDBDataset`: Es el eslabón que enlaza los componentes de acceso a datos con el informe. Modificamos su propiedad `DataSet` para que apunte al `TClientDataSet`.

- `TfrxReport`: Finalmente incluimos el componente que representa al informe y cambiamos su propiedad `DataSet` para enlazarlo con el `TfrxDBDataSet`.

En la imagen siguiente puede verse el conjunto completo de componentes necesarios ya insertados en el formulario.



Si hacemos doble clic sobre el componente `TfrxReport`, a fin de abrir el diseñador de informes, seguramente en la página Data dispuesta en el margen derecho no veamos aparecer las columnas de la tabla de ordenadores. Para que el diseñador obtenga la información que necesita de los datos, y nos permita usarlos como campos del informe, hemos de usar la opción Report>Data del diseñador. Ésta da paso a un cuadro de diálogo como el mostrado en la imagen de la página siguiente, en el que marcaremos de los `TfrxDBDataset` disponibles aquellos que queremos utilizar.



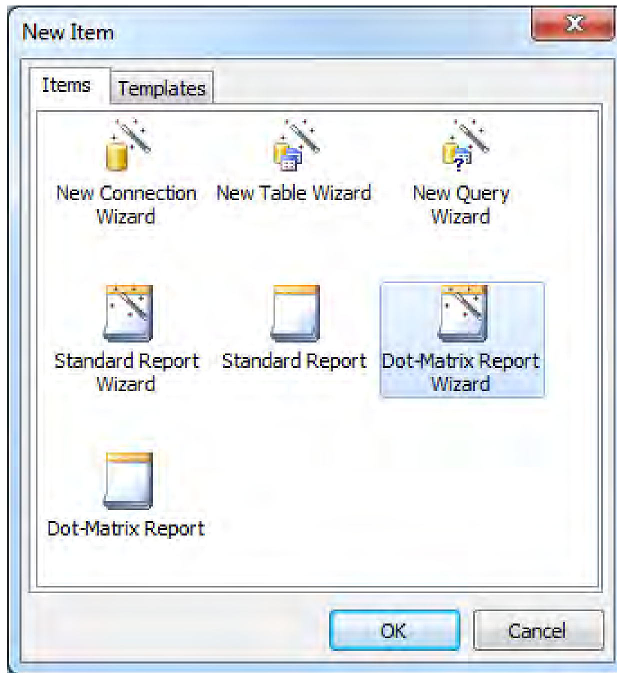
Asistentes y plantillas para informes

El diseñador de FastReport nos ofrece, como el de Delphi, una paleta con componentes que pueden incluirse en un informe, los propios campos de datos y expresiones predefinidas; un editor de propiedades que permite personalizar cualquiera de esos objetos y una superficie de trabajo en la que el contenedor es una página del informe.

Podríamos diseñar el informe arrastrando y soltando componentes y editando sus propiedades, definiendo las distintas secciones: encabezado, título de grupo, cuerpo y pie, colocando en cada una de ellas los textos y campos que nos interesen, estableciendo su posición y atributos visuales. Es un proceso que solamente requiere por nuestra parte que nos familiaricemos con el editor y sus opciones.

Otra posibilidad es usar las plantillas y asistentes para informes con que cuenta FastReport, una alternativa que puede ahorrarnos gran parte del trabajo más tedioso: la inserción de todos los componentes, permitiéndonos centrarnos exclusivamente en el contenido del informe y en ajustar su apariencia final.

La opción File>New del diseñador de FastReport da paso al cuadro de diálogo New Item (véase la imagen inferior), en el que podremos elegir entre varios asistentes y plantillas.

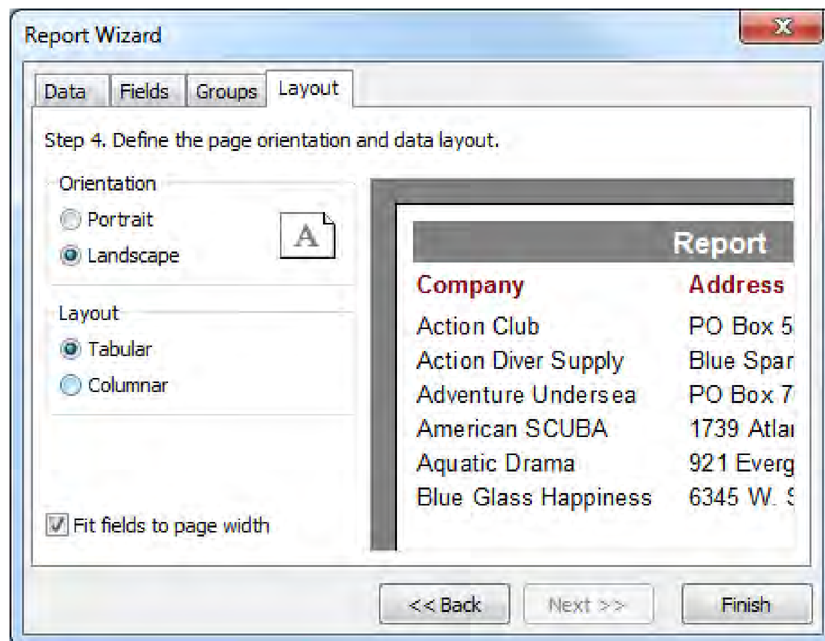


En nuestro caso vamos a usar el Dot-Matrix Report Wizard para establecer la configuración inicial del informe que queremos realizar. Se trata de un asistente en cuatro pasos que se corresponden con las cuatro páginas de la ventana Report Wizard. Los botones Back y Next situados en la parte inferior del asistente se limitan a cambiar la página activa, algo que podemos hacer directamente mediante un clic en la pestaña correspondientes. Las cuatro páginas, y su finalidad, son las siguientes:

- Data: Ofrece una lista desplegable de la que podemos elegir el `TfrxDBDataset` del que se obtendrán los datos, así como dos opciones adicionales para recuperar datos de una tabla a partir de una conexión o bien definir una consulta.
- Fields: De las columnas que existan en el conjunto de datos, ya sean las de una tabla o seleccionadas en una consulta, en el informe podemos usarlas todas o un subconjunto de ellas. En esta página del

asistente se ofrecen dos listas: una a la izquierda en la que aparecen todas las disponibles y otra a la derecha, inicialmente vacía, a la que agregaremos aquellas a incluir en el informe.

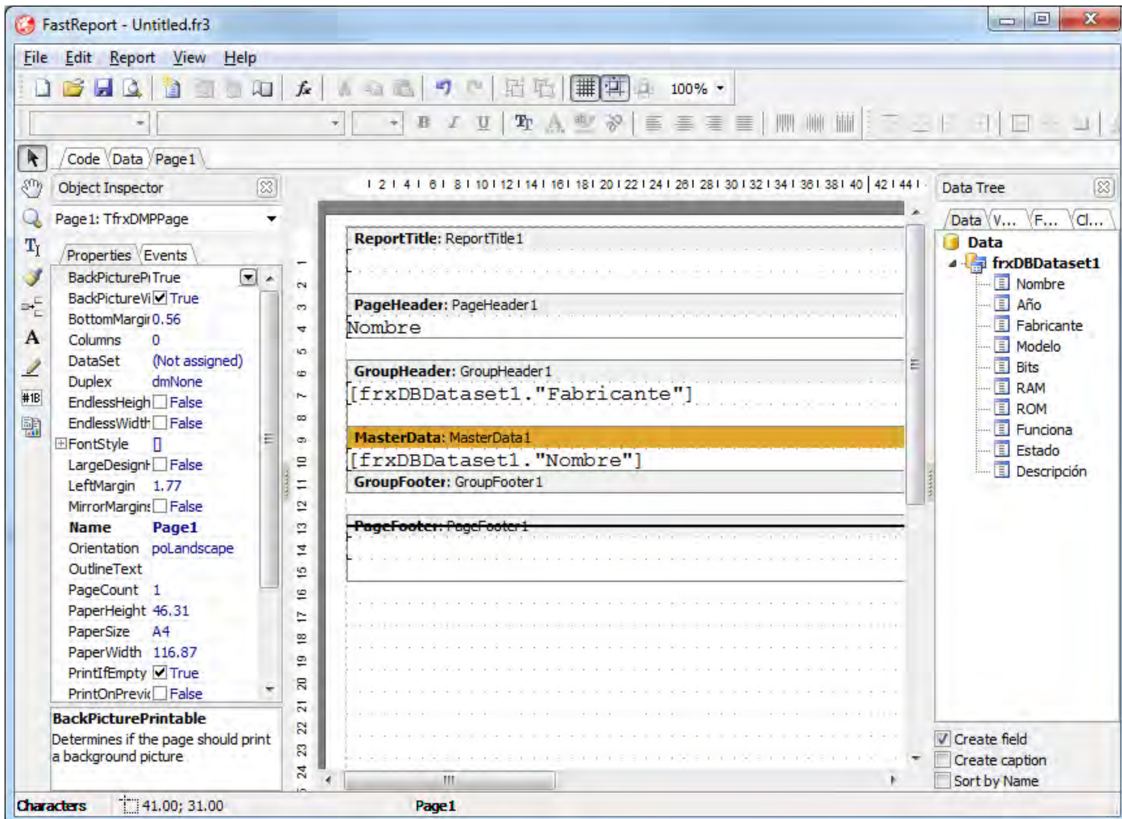
- Groups: En caso de que deseemos crear grupos de filas en el informe, en esta página seleccionaremos las columnas cuyo cambio de valor determinará el fin de un grupo y el inicio de otro.
- Layout: Finalmente configuraremos la estructura del informe, eligiendo la orientación del papel y la distribución de los datos entre dos opciones posibles: tabulado o por columnas. Al cambiar la opción elegida se ofrecerá una vista previa en el panel derecho (véase la imagen inferior).



Cuando hayamos establecido todos los parámetros en las distintas páginas del asistente, a las que podemos volver tantas veces como necesitemos hasta obtener el resultado deseado, un clic en el botón Finish generará el informe y lo abrirá en el diseñador.

En nuestro caso la página obtenida constará de un título de informe, un encabezado a repetir en cada página, el contenido del encabezado de grupo,

el cuerpo del informe y el pie de página. En la imagen inferior puede verse el aspecto que tendría al cerrar el asistente.



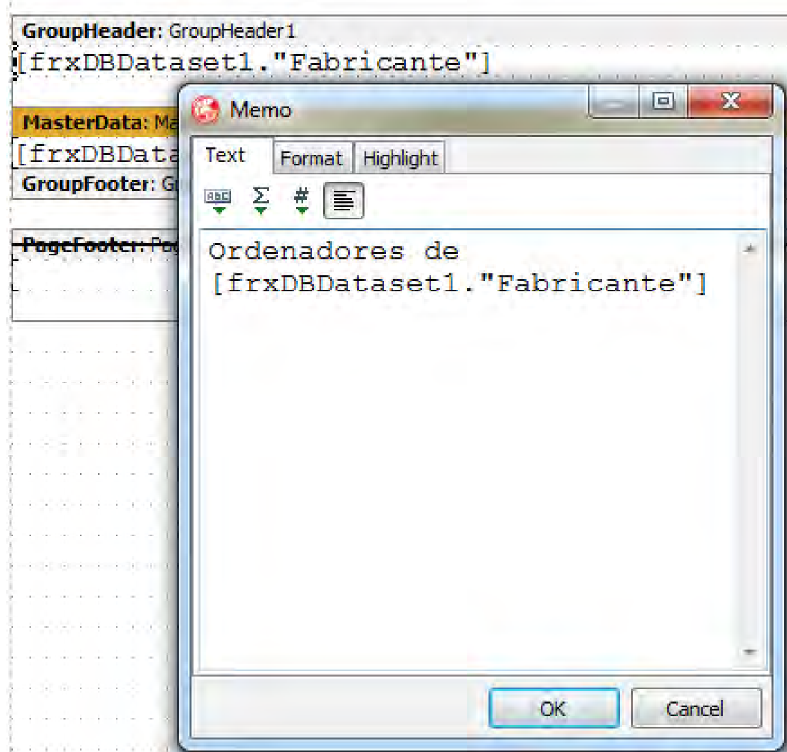
Personalización del informe

Tanto si hemos partido de un informe generado por un asistente, como en el punto anterior, como si lo hemos hecho desde una página vacía en la que hemos insertado las columnas de datos arrastrándolas desde la página Data y colocándolas en la sección deseada, el procedimiento para personalizar el informe es siempre el mismo.

Un clic sobre cualquier objeto del informe lo selecciona, lo cual nos permite modificar sus propiedades en el Inspector de objetos. De esta forma podemos, por ejemplo, establecer el tipo y estilo de letra de una sección:

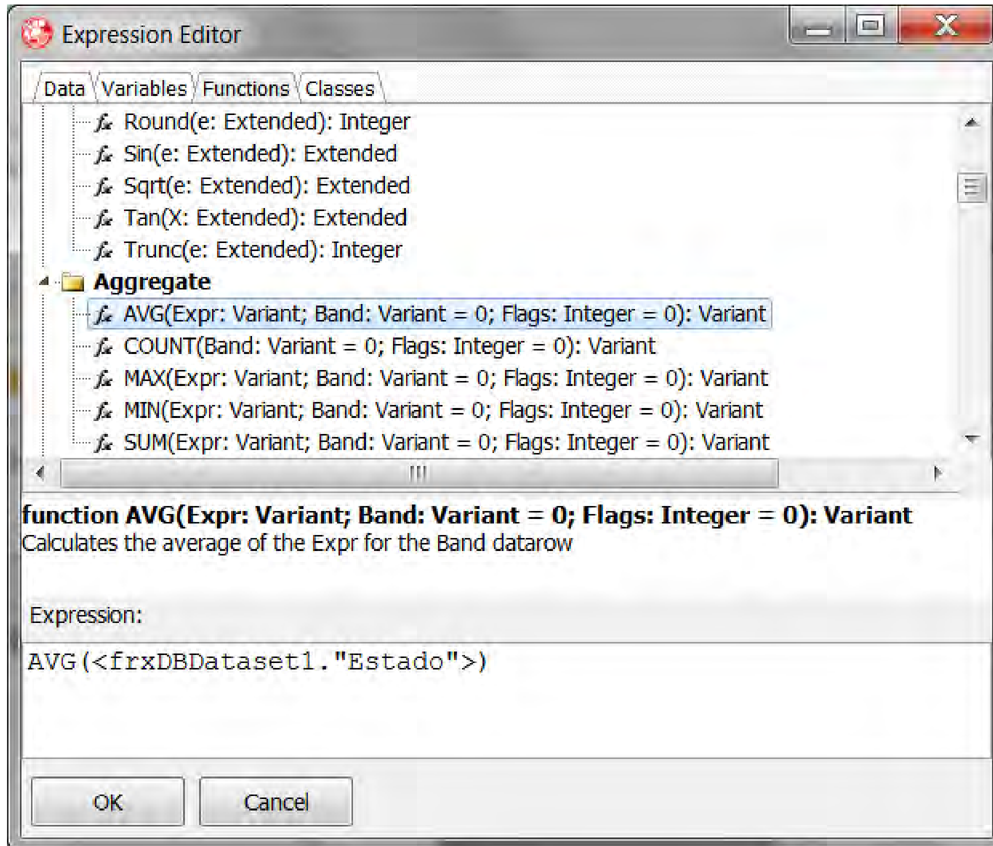
propiedad Font; la alineación de su contenido: propiedad HAlign; los bordes que se dibujarán alrededor: propiedad Frame, o si ha de dividirse el contenido en múltiples líneas en caso necesario: propiedad WordWrap.

Si lo que queremos es modificar el contenido de un objeto, generalmente tendremos que hacer doble clic sobre él para abrir el correspondiente editor. En la imagen inferior puede verse cómo se cambia el encabezado de grupo que el asistente había insertado en el punto anterior. El cuadro de diálogo que aparece en primer plano facilita la edición de ese contenido, lo que nos permite escribir un texto delante del nombre del fabricante, así como insertar expresiones y campos especiales mediante los botones que hay en la parte superior.



El diseñador cuenta con un editor específico que facilita la composición de expresiones y que, como se aprecia en la imagen de la página siguiente, pone a nuestra disposición múltiples funciones de diferentes tipos, variables de sistema y las propias columnas de datos. Podemos, por ejemplo, mostrar el

grado de conservación promedio para cada grupo de ordenadores mostrado en el informe.



> NOTA
>
> Aunque el componente `TfrxReport` conservará el diseño del informe, escribiéndolo en el archivo `.dfm` correspondiente al formulario en el que está incluido, desde el diseñador de FastReport también podemos guardar el informe en un archivo externo. Esto nos permitirá recuperarlo en cualquier momento donde lo necesitemos, ya sea el actual proyecto u otro, asegurándonos de no perderlo en caso de que surja algún problema con el citado componente.

Previsualización e impresión

Una vez que hayamos completado el diseño del informe llega el momento de usarlo, facilitando al usuario de nuestra aplicación su previsualización, impresión y exportación. Todas estas tareas pueden realizarse desde la ventana de previsualización por defecto del componente `TfrxReport`, por lo que en la práctica nuestro programa puede limitarse a abrir dicha ventana. Para ello usaremos el método `ShowReport` de `TfrxReport`.

En la ventana de previsualización, que puede verse en la imagen inferior mostrando el informe diseñado anteriormente, hay una barra de botones en la parte superior con opciones para imprimir el informe, exportarlo, ajustar la visualización, etc.

Preview

75% 1 Close

Listado de ordenadores en la colección

Nombre	Año	RAM	ROM	Estado
Ordenadores de				
Oric 1	1982			72
Ordenadores de Amstrad				
Amstrad CPC-464	1983	64	32	70
Ordenadores de Atari				
Atari 400	1979	8	16	80
Ordenadores de Commodore				
Commodore 64	1982	64	32	76
Ordenadores de Elan				
Enterprise 64	1984	64	48	85
Ordenadores de Sinclair				
ZX Spectrum	1982	48	16	70

Page 1 of 1

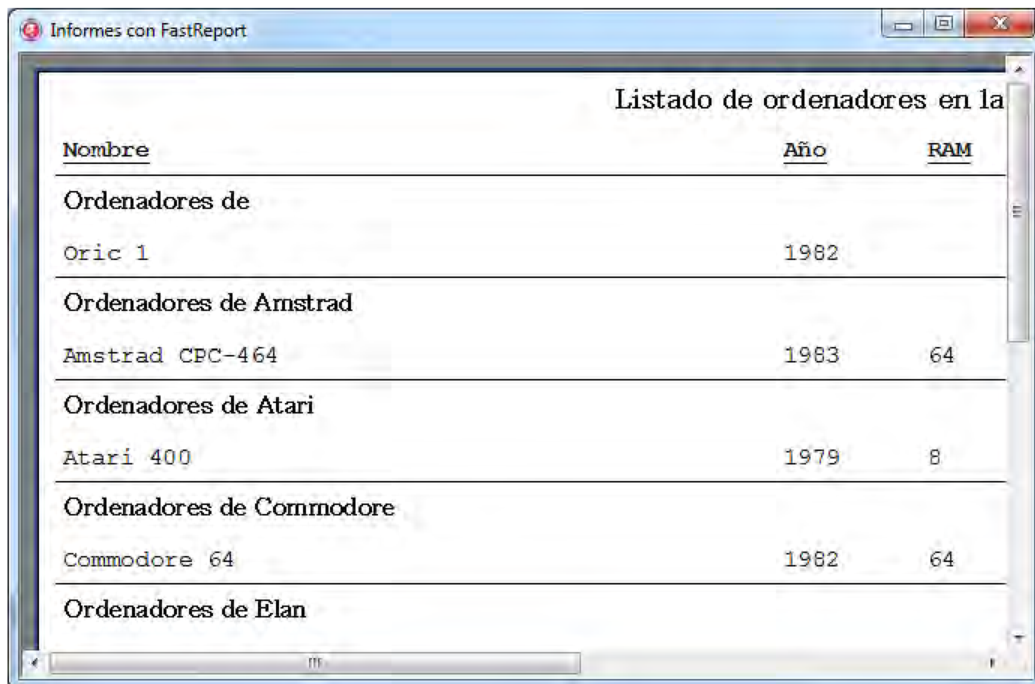
También podemos imprimir directamente el informe simplemente usando el método `Print` del `TfrxReport`, así como exportarlo mediante el método `Export` entregando como parámetro el componente de exportación adecuado.

Personalizar la vista previa

Si no queremos usar la ventana de previsualización estándar de FastReport, porque no interese que los usuarios tengan acceso a todas sus opciones o nos resulte demasiado impersonal, siempre podemos diseñar nuestra propia vista previa. Para ello recurriremos al componente `TfrxPreview`.

Este componente aparece como una página del informe y normalmente nos interesará ajustarlo a un margen del formulario, incluso ocupando todo el espacio disponible. Para que al invocar al método `ShowReport` del `TfrxReport` el informe aparezca en esa superficie, y no en la ventana independiente, tendremos que asignar a la propiedad Preview del segundo una referencia al `TfrxPreview`.

Como puede comprobarse en la imagen inferior, que muestra el mismo informe anterior en un `TfrxPreview`, este componente no ofrece en principio más que la superficie de visualización. No existe una barra de botones ni un menú con opciones para imprimir o exportar el informe. Tampoco para cambiar la página que está mostrándose o cambiar el factor de zoom.



Todas las operaciones que deseemos facilitar al usuario tendremos que ofrecerlas agregando los elementos necesarios a la interfaz, por ejemplo botones o un menú contextual, convirtiendo los eventos OnClick en llamadas a métodos como Print y Export.

Informes FastReport en aplicaciones FMX

Al inicio de este capítulo se apuntaba que una de las limitaciones de FastReport es que sus componentes tienen dependencias de la VCL, por lo que no pueden ser usados en proyectos basados en la FMX. Dado que ésta es la biblioteca preferente para el diseño de interfaces de usuario en Delphi XE2, y versiones futuras del producto, es de esperar que la empresa desarrolladora de FastReport ofrezca una versión FMX próximamente.

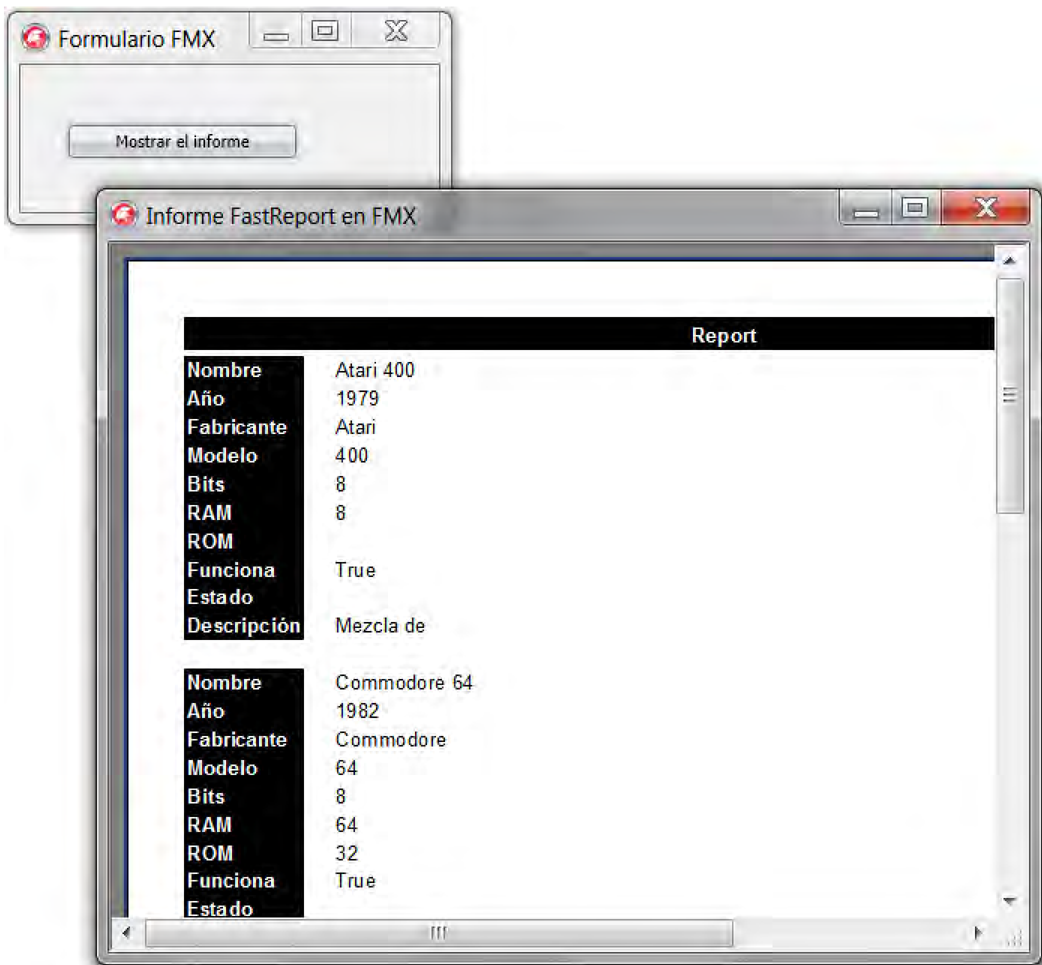
Mientras tanto, sin embargo, tendremos que convivir con esta limitación. O quizá no si encontramos la manera de combinar en un proyecto formularios FMX y VCL, algo que, teóricamente, no podemos hacer.

Si iniciamos un proyecto FMX, como hemos hecho en todos los capítulos previos, al abrir la ventana New Items para agregar nuevos elementos veremos que no aparece por ninguna parte la opción VCL Form. A pesar de ello, podemos introducir en ese proyecto un formulario VCL con un informe FastReport con los pasos siguientes:

- Creamos el nuevo proyecto FMX, con los formularios que se necesiten para la interfaz de usuario de la aplicación. El proceso sería el que hemos aprendido en capítulos previos.
- Agregamos al mismo grupo un proyecto de tipo VCL que de partida contará con un formulario.
- Diseñamos en ese formulario el informe FastReport usando los componentes que hemos conocido en este capítulo.
- Completado al informe tomamos el módulo correspondiente al formulario en el Gestor de proyectos y lo arrastramos hasta el proyecto FMX, respondiendo afirmativamente a la pregunta de si deseamos agregarlo.

- Añadimos al formulario FMX que proceda una referencia al módulo recién agregado al proyecto, así como una opción para abrir el formulario VCL y poder facilitar el acceso al informe.

Con este *truco* se ha creado un proyecto cuyo resultado puede verse en la imagen inferior. El formulario que tiene el botón Mostrar el informe es de tipo FMX, como el propio proyecto, mientras que la ventana en primer plano que muestra el informe es un formulario VCL. Ambos conviven en el mismo ejecutable sin que, en principio, parezcan existir problemas insalvables.



Conclusión

Delphi XE2 nos ofrece varias alternativas a la hora de diseñar y generar informes, por lo general a partir de la información almacenada en las bases de datos, siendo FastReport una de ellas.

En este capítulo se han descrito los componentes fundamentales de FastReport, así como el proceso a seguir para diseñar un informe partiendo desde cero, mostrar una vista previa, imprimirlo y exportarlo. También se ha explicado cómo es posible salvar el obstáculo que impide el uso de FastReport en aplicaciones FMX.

Partiendo de estos fundamentos todo lo que necesitamos es familiarizarnos con el entorno de diseño de FastReport, especialmente todas las funciones y componentes que ofrece.

A continuación

El siguiente capítulo, con el que se cierra la quinta parte y también concluye este libro, estará dedicado a un aspecto que suele siempre descuidarse pero que tienen una gran importancia: la documentación del código de nuestros proyectos.

Conoceremos el formato XMLDoc y cómo el complemento Documentation Insight facilita la edición de documentación en dicho formato, cómo esa documentación se convierte de manera inmediata en ayuda en el entorno y cómo generar documentación electrónica.

Capítulo 15: Documentation Insight

Uno de los aspectos a los que menos atención se presta durante el desarrollo de cualquier proyecto software es su documentación, especialmente cuando existe premura en los plazos a cumplir. Sin embargo es un factor muy importante para el futuro de ese software, ya que la mayor parte de su tiempo de vida se dedicará al mantenimiento, no al desarrollo inicial, y es muy habitual que otros programadores tengan que enfrentarse a ese código.

Delphi XE2 integra en su entorno una herramienta, desarrollada por la empresa DevJET Software, que nos hará más fácil documentar el código a medida que lo escribimos. Este último capítulo del libro explica cómo usar dicha herramienta y también otras opciones de Delphi relacionadas con la generación de documentación asociada a un proyecto.

Documentación XMLDoc

Delphi contempla la inclusión de comentarios de documentación en el código desde la versión 2006[145], empleando un formato XML prácticamente idéntico al de C# en Visual Studio conocido como XMLDoc. Cada línea XMLDoc se inicia con una triple barra inclinada, tras la cual se dispondrán marcas (etiquetas) y texto:

```
/// <marca>Texto de documentación</marca>
```

Las marcas se ajustan a la sintaxis del lenguaje XML, por lo que se introducen entre los símbolos < y > y siempre han de ir en parejas: una de apertura y otra de cierre. Algunas de las marcas pueden tener atributos:

```
/// <marca atributo="valor">
```

En XMLDoc existe un conjunto de etiquetas predefinidas, algunas de propósito general y otras especificas para cada tipo de entidad que puede existir en el código: en los procedimientos se contemplan los parámetros (su tipo y finalidad), en las funciones el tipo del valor de retorno, etc. En total son en torno a una veintena de marcas distintas de uso habitual.

Lo interesante es que a partir de esos comentarios con marcas que documentan el código, Delphi es capaz de extraer la documentación y usarla de manera inmediata para ofrecer ayuda mediante Help Insight, en las ventanas flotantes que aparecen al introducir el nombre de un objeto, propiedad o método o al situar el puntero del ratón sobre el nombre de un tipo o una clase. Además también es posible generar documentación electrónica, en formato XML, incluyendo en ella la información que hayamos aportado en el código. Los beneficios, como queda patente, son valiosos.

A pesar de estas ventajas son muy pocos los programadores que tienen por costumbre documentar adecuadamente el código, en ocasiones por propios principios sobre qué debe ser incluido dentro del código fuente pero la mayoría de las veces sencillamente por el esfuerzo que supone recordar y escribir manualmente todas las etiquetas XMLDoc. Sin embargo, tal y como sugiere la famosa frase "*Always write and document your code as if the*

145 En versiones previas también podían introducirse comentarios de documentación, ya que éstos no afectan al código fuente, pero es a partir de la versión 2006 cuando esos comentarios son analizados y utilizados para producir documentación útil disponible de forma inmediata en *Help Insight*.

man that is going to maintain it is a serial killer who knows where you live", deberíamos poner tanto esmero en escribir la documentación como el que procuramos el escribir el propio código, para evitar despertar malas intenciones en el futuro encargado de mantener el proyecto.

> NOTA
>
> La documentación del código es especialmente importante cuando se escriben bibliotecas de clases o componentes que van a ser utilizados por terceros. Preguntémonos cómo sería nuestro trabajo con la VCL o la FMX si nos quitasen la ayuda que va apareciendo a medida que escribimos código Delphi.

Editar la documentación con Documentation Insight

Documentation Insight está integrado en Delphi como un complemento, por lo que se aloja en el menú Tools como otras herramientas externas. Su finalidad es facilitar la edición[146] de la documentación en formato XMLDoc, evitando que tengamos que acordarnos de todas las marcas, sus atributos y estructura.

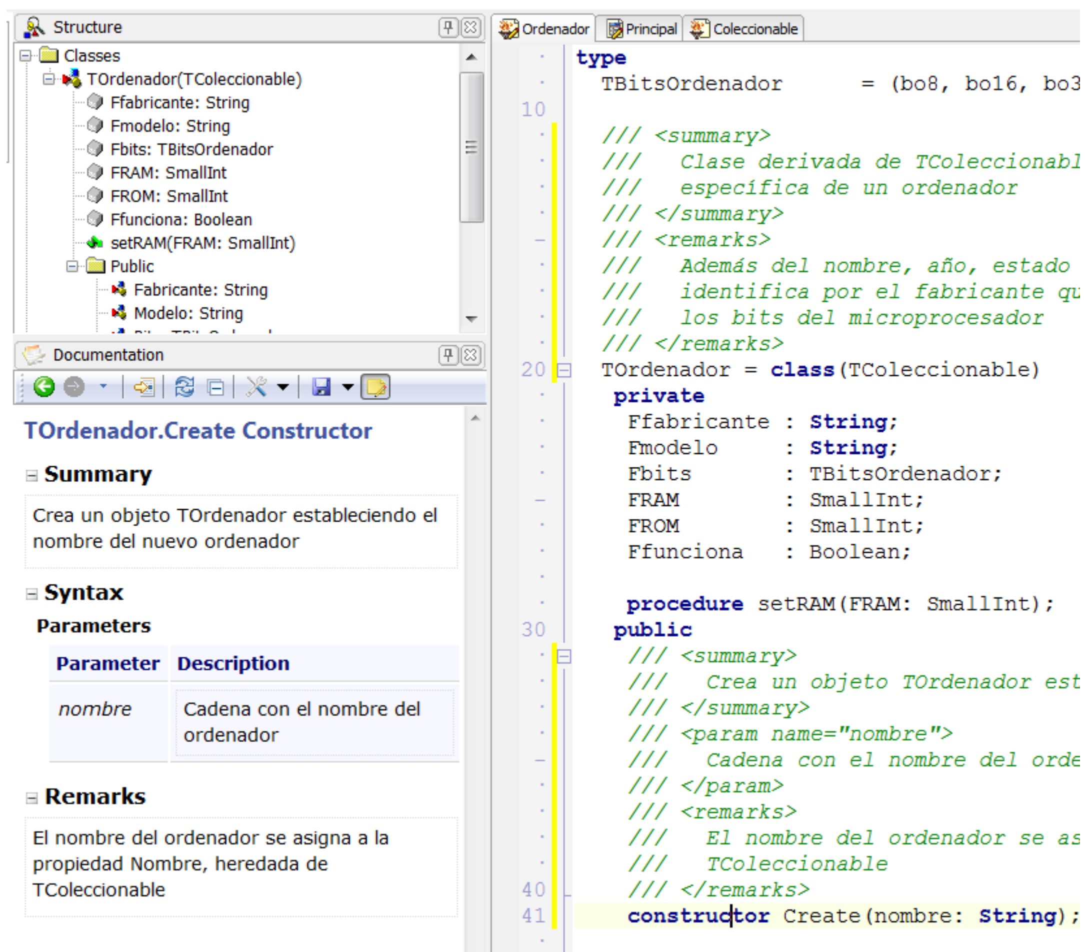
El elemento de trabajo fundamental de Documentation Insight es un panel de edición/visualización que abriremos mediante la opción Show Documentation del submenú Tools>Documentation Insight Express. También podemos usar el atajo de teclado Control+Mayúsculas+D, cuando nos acostumbremos a él nos resultará mucho más ágil.

Inicialmente el panel aparece como una ventana flotante, pero podemos adosarlo junto al Gestor de proyectos o el Inspector de objetos, según nos sea más cómodo.

---

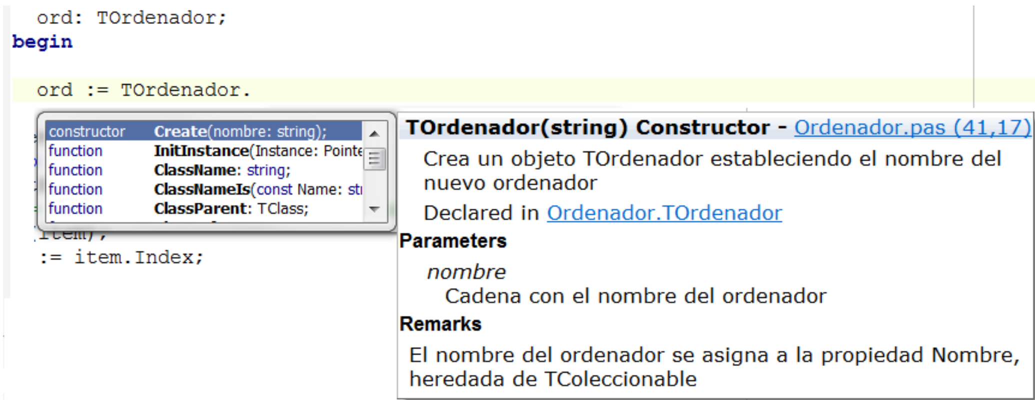
146 La edición de Documentation Insight que incluye Delphi XE2 es la Express, una versión reducida que no facilita la edición de todas las marcas de XMLDoc y que solamente contempla la documentación de miembros públicos, no protegidos o privados.

Una vez abierto el citado panel, al situar en el editor el cursor sobre la definición de una clase, un método, una propiedad, etc., en él aparecerán apartados como Summary, Remarks o Parameters. Cada uno de ellos contará con un recuadro donde podemos escribir la documentación que, de manera inmediata, se sincronizará con el editor incluyéndose de forma automática las etiquetas adecuadas. En la imagen inferior puede verse la documentación del constructor de la clase `TOrdenador`, una clase que definíamos en el capítulo dedicado al lenguaje Delphi.

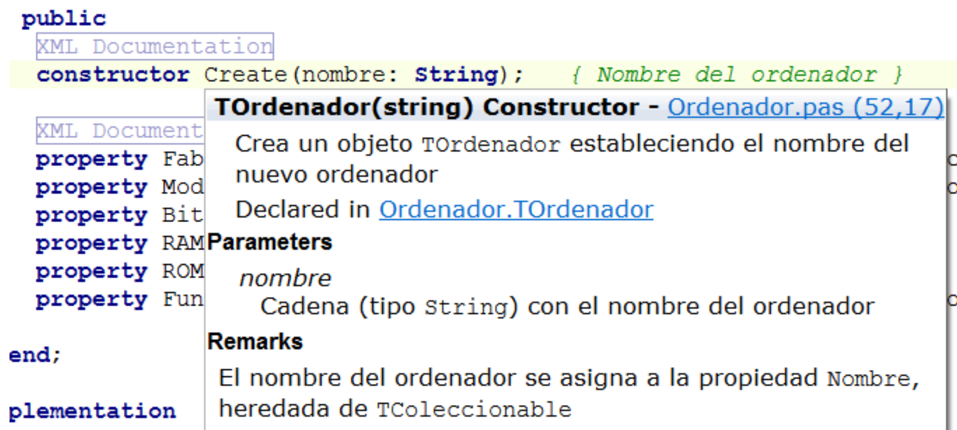


La lista de parámetros de los métodos, el tipo de los parámetros y otros datos sobre los elementos a documentar se obtienen automáticamente.

En cuanto documentemos un elemento del código, como puede ser el anterior constructor, el editor de Delphi extraerá esa información y la usará para facilitar ayuda. Podemos comprobarlo, como se muestra en la imagen inferior, creando un objeto `TOrdenador` desde otro módulo de código. Al introducir el punto tras el nombre de la clase aparece la descripción del constructor y la explicación de sus parámetros.



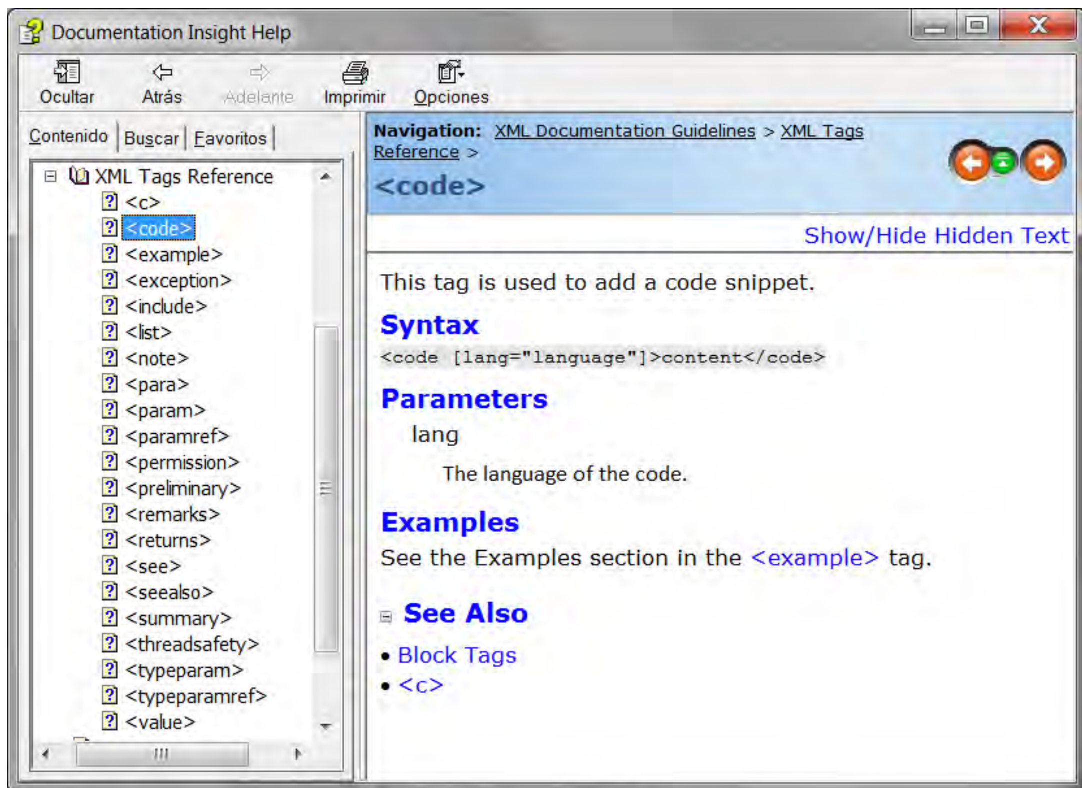
Si la documentación del código es muy extensa puede llegar a dificultar la navegación por el código mientras se trabaja con él, pero es un problema que tiene fácil solución: no hay más que usar la opción Fold all documentation in the source code de Documentation Insight, mediante el quinto de los botones que aparecen en la parte superior (contando de izquierda a derecha). En la imagen inferior pueden verse en segundo plano los bloques de comentarios cerrados y en primer plano la ayuda de Help Insight.



Edición manual de la documentación

La edición *Express* de Documentation Insight está limitada a la edición de las etiquetas summary, remarks, param, returns y value. Para editar las marcas restantes tendríamos que adquirir la versión *Professional* del producto o, en su lugar, optar por complementar la documentación editándola manualmente.

Con la opción Tools>Documentation Insight Express>Help podemos acceder a un documento de ayuda (véase la imagen inferior) en el que se facilita una referencia completa de las marcas de XMLDoc que reconoce Delphi. Algunas de ellas se aplican en el interior de un texto para darle formato, por ejemplo la etiqueta c para hacer que una palabra dentro de una frase aparezca como su fuese código, mientras que otras marcan secciones completas como puede ser example o code.



Apoyándonos en esta información podríamos complementar la documentación asociada al constructor de la clase `TOrdenador`, dejándola como se muestra a continuación:

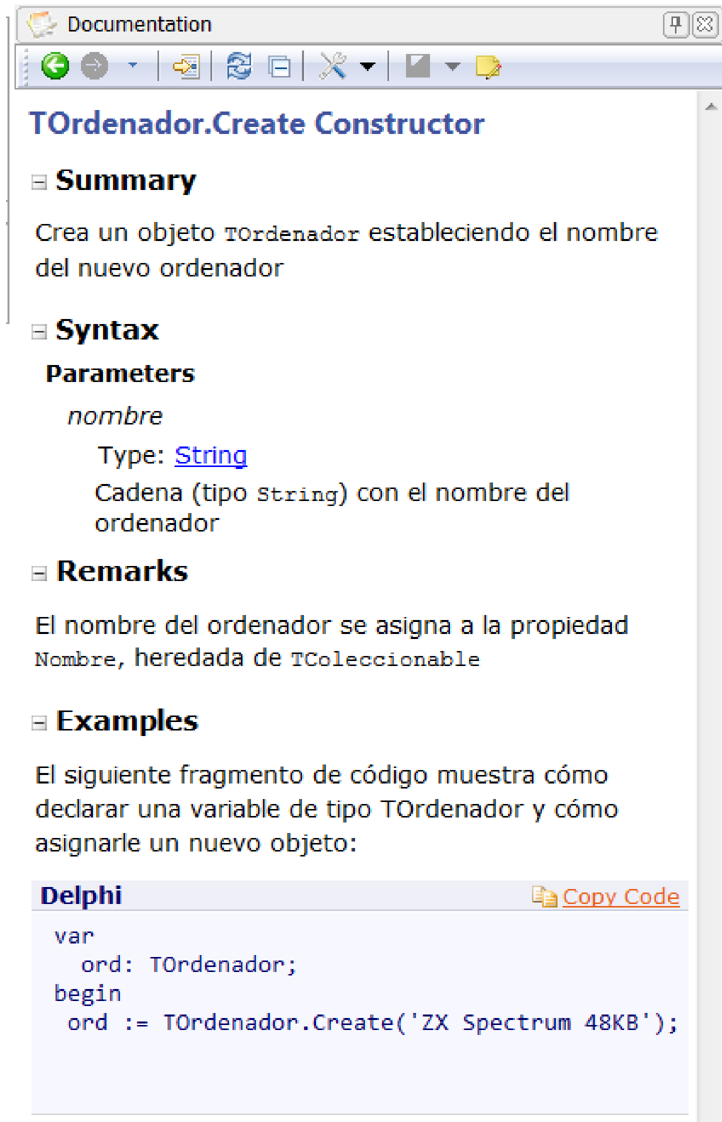
```
///  <summary>
///    Crea un objeto <c>TOrdenador</c> estableciendo el
///  nombre del nuevo ordenador
///  </summary>
///  <param name="nombre">
///    Cadena (tipo <c>String</c>) con el nombre del
///    ordenador
///  </param>
///  <remarks>
///    El nombre del ordenador se asigna a la propiedad
///    <c>Nombre</c>, heredada de <c>TColeccionable</c>
///  </remarks>
///  <example>
///    El siguiente fragmento de código muestra cómo
///    declarar una variable de tipo TOrdenador y cómo
///    asignarle un nuevo objeto:
///    <code lang="Delphi">
///   var
///     ord: TOrdenador;
///   begin
///    ord := TOrdenador.Create('ZX Spectrum 48KB');
///    </code>
///  </example>
```

Hemos usado la marca `c` en varios puntos para dar formato de código a nombres de clases y propiedades. En la parte final hemos agregado un ejemplo mediante la etiqueta `example`. Ésta contiene la explicación del ejemplo, mientras que el código en sí está contenido en una marca `code` que, como puede verse, cuenta con un atributo llamado `lang`. Su finalidad es indicar el lenguaje de programación en que está escrito el código.

De manera similar podríamos agregar enlaces a información adicional externa, con las etiquetas `see` y `seealso`, incluir detalles sobre posibles excepciones que podrían producirse, con la etiqueta `exception`, etc.

La sincronización entre la documentación incluida en el código, siempre que se ajuste a la sintaxis XMLDoc, y el panel de Documentation Insight suele ser inmediata. No obstante algunos apartados, como es el caso del ejemplo que se ha añadido al final, no aparecerán en el modo de operación por defecto en dicho panel. Tendremos que cambiar del modo de edición al de

visualización, mediante el último de los botones, para obtener un resultado como el que puede verse en la imagen siguiente.



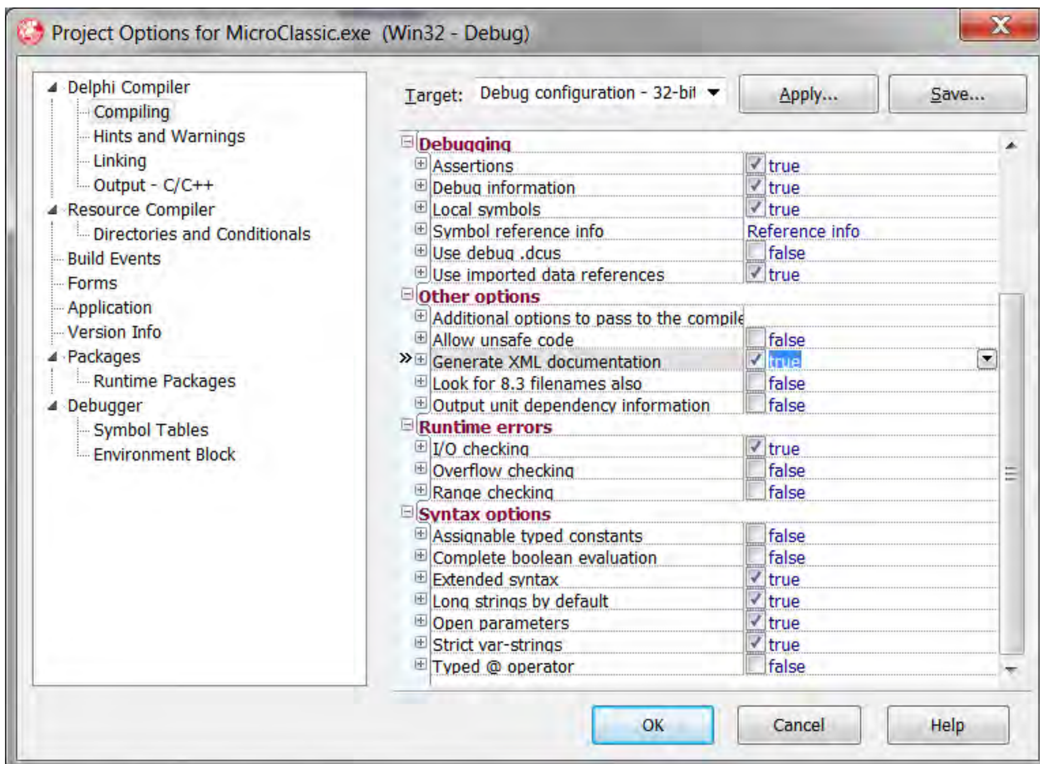
No solamente vemos el ejemplo, con el código formateado, sino que incluso podemos copiar ese código al portapapeles para pegarlo en el editor de Delphi y reutilizarlo.

Generar documentación electrónica

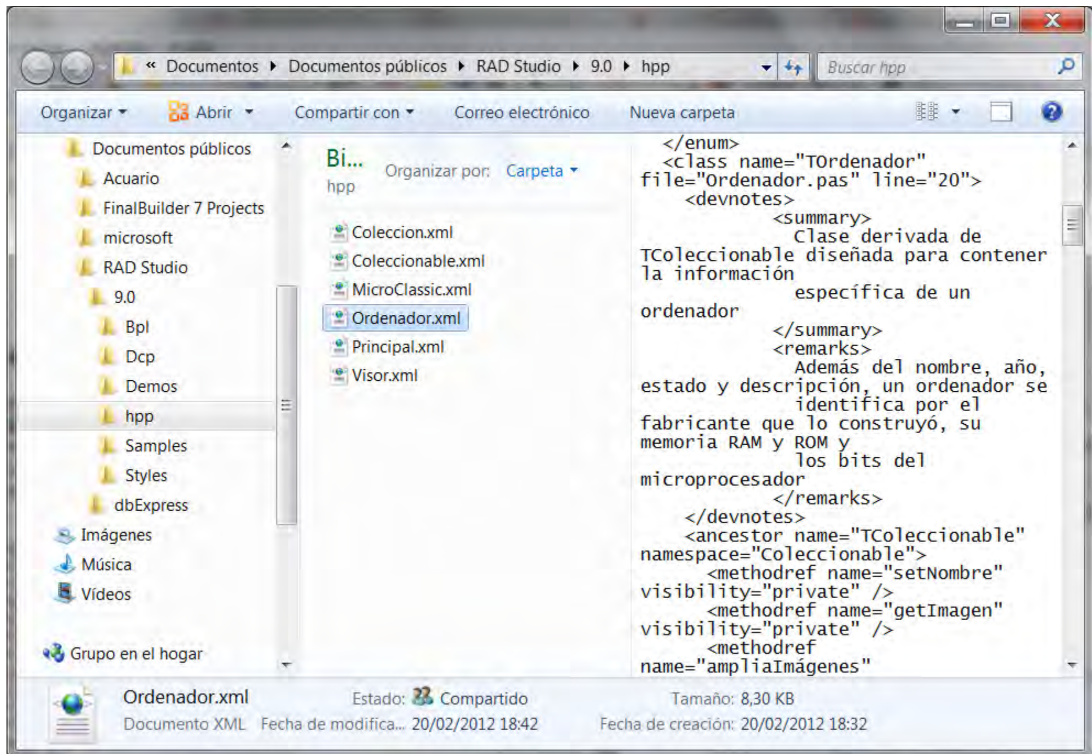
En principio la documentación que incluyamos en el código de un proyecto solamente servirá para que Help Insight, o el propio panel de Documentation Insight, nos ofrezcan ayuda a medida que escribimos nuevo código.

Por regla general también nos interesará generar una documentación en formato electrónico, una ayuda que pueda consultarse de manera independiente al editor de código. Para ello tenemos básicamente dos alternativas.

La primera es activar la opción Generate XML documentation en la rama Delphi Compiler>Compiling de la ventana de opciones del proyecto, tal y como se muestra en la imagen inferior.



Tras marcar dicha opción al compilar el proyecto se generarán una serie de archivos XML, uno por módulo, que contiene no solamente los comentarios que hemos introducido nosotros, sino mucha más información extraída del propio código tal y como se aprecia en el panel derecho de la imagen inferior.

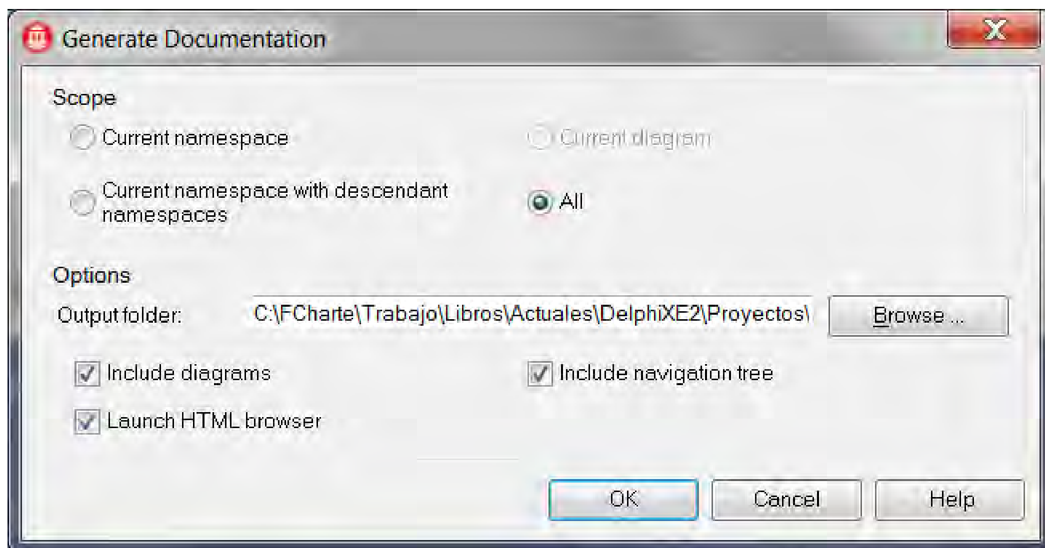


A partir de estos archivos XML se produce la ayuda en el formato que se desee, normalmente usando una hoja de estilos y el estándar XLST (*eXtensible Stylesheet Language Transformations*). También existen herramientas que se encargan de esta tarea y que normalmente ofrecen como salida ayuda en formato PDF, HTML, CHM, etc. Contacta con Danysoft para más información.

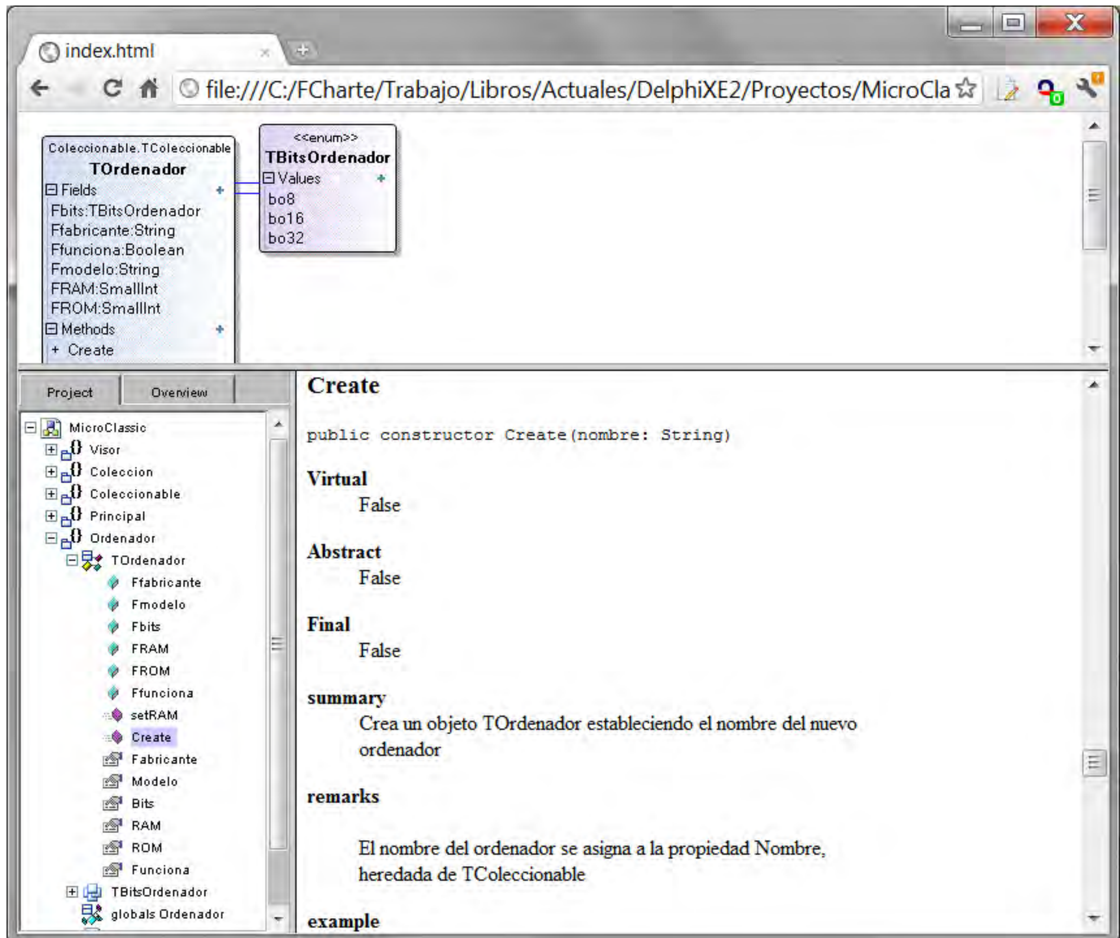
> NOTA
>
> En http://www.andromeda.nl/projects/xmldoc/xmldoc.html se puede obtener una utilidad que genera HTML y LaTeX desde XMLDoc.

La segunda alternativa produce la documentación electrónica directamente a partir del código fuente, sin necesidad de generar archivos XML intermedios. Para obtenerla hemos de dar los siguientes pasos:

- Abrimos el panel Model View. Suele aparecer como una página junto al Gestor de proyectos, pero siempre podemos mostrarlo con la opción View>Model View.
- Aparecerá un pequeño cuadro de diálogo indicando que el proyecto actual no está configurado para usar herramientas de modelado, dándonos opción a activar esta funcionalidad. Respondemos afirmativamente.
- En la ventana Model View aparece el proyecto como un nodo raíz, con los módulos como ramas. Abrimos el menú contextual del proyecto y seleccionamos la opción Generate Documentation.
- Se abre el cuadro de diálogo del mismo nombre (véase la figura inferior), en el que configuraremos la documentación a generar: el ámbito de la documentación por defecto es All, por lo que se incluirán todos los módulos. En el apartado Options elegiremos si han de incluirse diagramas de clases y si queremos o no un diagrama de navegación por la ayuda. También facilitaremos la ruta donde se escribirá.



Terminado el proceso, siempre que no hayamos desmarcado la opción Launch HTML Browser, veremos aparecer la documentación electrónica en el navegador por defecto, como en el caso de la imagen siguiente. En él se aprecia el diagrama con la clase `TOrdenador` y la enumeración `TBitsOrdenador` en la parte superior, el árbol de navegación en el panel inferior izquierdo y la documentación propiamente dicha a su derecha.



En lugar de realizar este proceso desde el entorno de Delphi, con los pasos indicados, también podemos ejecutarlo con la utilidad `GenDocCLI.exe` que encontraremos en la subcarpeta `bin` de la carpeta en que esté instalado Delphi XE2. Solamente hay que indicar el nombre del proyecto.

Conclusión

Documentar el código de un proyecto software implica una mayor cantidad de trabajo por parte del programador, pero éste hay que verlo como una inversión que se hace ahora para el futuro, cuando nosotros mismos o un tercero tenga que mantener ese código. Dicha inversión es del todo imprescindible cuando el proyecto en el que se trabaja está pensado para servir como base para desarrollos de terceros, como cuando se crean bibliotecas de clases y de componentes.

Con Documentation Insight la tarea de documentar el código, aunque sea en sus aspectos más básicos, se ve simplificada como hemos podido comprobar en este capítulo. Basta con escribir la descripción de métodos, parámetros y clases en unos recuadros de texto, desencadenando la inserción automática de las etiquetas XMLDoc apropiadas. De esta manera Delphi XE2 elimina una de las escusas habituales para no documentar el código.

Indice

32-bit Windows....368
3D Shapes....439
64 bits....363
64-bits Windows....369
Abstract....114
Actions....343
Active....216
Add Breakpoint....71
Add ConnectionString Param....209
Add Feature Files....378
Add New....67
Add New Connection....149
Add New>Unit....112
Add Platform....368
ADO....141
AfterDispatch....351
Aggregates....158
AlBottom....51, 395
AlC....46
AlCenter....395
AlClient....50, 395
AlContent....395
AlContents....46
AlFit....395
AlFitLeft....395
AlFitRight....395
AlHorizontal....395
AlHorzCenter....395
Align....46, 50, 394
AlLeft....50, 395
AllowMultipleSessionsPerUser....306
AlMostBottom....394
AlMostLeft....394
AlMostRight....394
AlMostTop....394
AlNone....50
AlRight....51, 395
AlScale....395
AlTop....51, 395
AlVertCenter....395
AlVertical....395
Ambient....447
Anchors....77
And....80, 98
Android....310
AnimationType....419, 421
AnsiChar....87
AnsiString....87

Application...417
ApplyStyleLookup...............................406
ApplyTo...291
ApplyUpdates..............................165, 224
AppName...305
Architect...31
ARM..468
Arquitecturas (bases de datos).............144
Array..91p.
Array of array.......................................94
AsBoolean..336
ASCII..85
AsInteger...336
Assign Remote Profile.........................374
Assigned..124
AtIn...422
AtInOut..422
AtOut...422
Atributos..109
Atributos de clase................................122
AttachedMenu....................................320
AuthBeforeNewSession................306, 327
AuthenticationManager.......................290
Auther...327
AutherPolicy.......................................326
AuthorizedRoles..................................291
Autocompletado de clases....................132
AutoReverse.......................................422
AutoStart..258
BDE..141
BeforeConnect....................................210
BeforeDispatch....................................351
Begin..61, 80p.
BevelKind...400
BevelWidth...400
BindingSource.....................................177
BindScope...192
Bitmap...58, 68
Blob..236
BlockCount...432
BLS Pascal..16
Boolean...88
BorderStyle.................................67, 400
Borland..16
Break...103
BsSizeToolWin.....................................67
BufferKBSize......................................261
Build Configurations............................40
Byte...86
ByteBool..88
C (marca XMLDoc)..............................510
C++ Builder...32
CacheContext......................................347
CacheDir..306
Call Stack..72
Camera...450
Caption..67
Cardinal...86
Case..101
ChangeCount......................................165
ChannelInfo..266
ChannelInfo.Id....................................266
Char..87
Check for Updates.................................33
Checked..336
Children..397
Chrome...310
Clase derivada....................................120
Clase final...125
Clase genérica.....................................128
Clases abstractas.................................114
Class...109
Class var...122
ClassGroup...154
ClassName..408
Clear...191
ClearExpressions.................................193
Cliente/Servidor (arquitectura)...........144
ClientInfo..266
ClipChildren..399
Close...211
CLX...17
Code...510p.
CodeGear...18
Color...400
Columns..357
COM..17
COM6..17
Comentarios..84
CommandText..............................217, 219

CommandType....217
Compartir Archivos....474
Compartir Pantalla....474
COMPAS Pascal....16
Conexiones....147
Conjuntos....96
Connected....208, 211
Connection....230
ConnectionName....206
ConnectionString....209
ConnectProperties....266
Constantes....116
Constraints....158
Constructor....109, 111
Constructores de clase....122
Contenedores....43
Continue....103
Controlador de bases de datos....146
ControlComponent....175, 177, 335
ControlExpression....175, 177
Controls....83
CORBA....17
Create....109
Create a Remote Profile....374
Create DataSet....162
Create Style....409
CreateDataSet....166
CtQuery....217
CtServerMethod....217
CtStoredProc....217
CtTable....217
Currency....87
CurValue....228
Danysoft ....1p.
Data....308, 404, 442
Data Explorer....146
Data.DBXCompressionFilter....290
Data.DBXFirebird....245
Data.DBXRSAFilter....290
Database....206
DataField....170, 312, 339
DataSet....187, 223, 490
DataSnap....142, 247
DataSnap.DBClient....157
DataSource....170, 189, 312, 339
DbExpress....141
DbGo....141
DBX....141
DbxCommandType....218
DBXCommon....212
Dbxconnections.ini....206
Dbxdrivers.ini....206
Dbxfb.dll....245
DbxMetaData....218
DBXMetaDataNames....215
DbxSQL....218
DbxStoredProcedure....218
DbxTable....218
DCC32....27
DCC64....27
DCCOSX....27, 456
DDL....216
Debug....70
Dec....118
Default....343
Definir clases....108
Delay....421
Delta....223
DeniedRoles....291
Deployment....76
Depth....440
Depurador....70
DerivedGetCommandTypes....218
Description....305
Destroy....109
Destructor....109
Destructores de clase....122
Diferencia (conjuntos)....99
Diffuse....447
DirBidirectional....175
DirControlToSource....175
Direct2D....388
Direct3D....436
Direction....175, 431
DirectoryAction....350
DirectoryMask....350
DirectX....388
DirSourceToControl....175
Diseñador de elementos....48
DisplayName....305

Dispose....119
Distribuida (arquitectura)....144
Div....98
DML....216
DoCommand....346
Documentation Insight....20
Double....86
Downto....102
Dpr2xcode.exe....469
Driver....206
DropDownCount....55
DSAdmin....259, 355
DSAuthenticationPassword....292
DSAuthenticationUser....292
DSContext....346, 355
DSProxy....485
DSProxyJavaAndroid....482
DSRESTCommand....484
DSRESTConnection....484p.
DSServer....249
Duration....419, 421
DYLD_LIBRARY_PATH....460
EAccessViolation....106
Edit....196
Edit Custom Style....403
Edit Default Style....408
Edit>Format Source....133
EditControl....189
Editor de campos....159
Else....101
Embarcadero....18
Emissive....447
Enabled....343, 422
Encapsulación....108
End....61, 80p.
Enterprise....31
Enumeraciones....90
EReconcileError....227
Estilos VCL....19
Evaluate/Modify....74
Eventos....46, 60
Events....46, 61
Example....510
Excepciones....106
Except....106
Exception....106, 511
Exclude....291
ExcludeClasses....349
ExcludeMethods....349
Execute....66, 216, 304
ExecuteMethod....353
Exponencial....89
Export....491, 500
Expresiones....98
Expressions....180
Extended....86
False....88, 422
FastReport....20, 489
Fbclient.dll....233
FetchParams....158
FieldByName....241
FieldDefs....158
FieldName....189
Fields....158, 240
Fields Editor....159
File>New....36
FileName....164
FilesDir....306
Fill....425
Fill.Color....409
Fill.Kind....409
FillList....199
FillMode....446
Filter....65
FilterId....261, 289
Filters....261, 289
Finalization....82
Finally....107
Firebird....142, 232
Firefox....310
FireMonkey....20, 28, 38, 387
FireMonkey HD Form....67
FlameRobin....148
FmSolid....446
FmWireframe....446
FMX....26, 38, 387
FMX_Types.TControl....154
FMX.Controls....84
FMX.Effects....431
FMX.Import....445

FMX.Types......................84, 113, 389, 393
FMX.Types.TControl............................154
FMX.Types3D....................................437
Font...............................54, 177, 400, 498
Footer..357
For...102, 105
For (colección)......................................103
Form1..83
Format...176, 191
Format Project Sources........................133
FormatDateTime.................................176
FormatExpressions..............................193
Formatter...133
Formulario..39
FPC..468
Frame..498
FreeMem...119
FreePascal...468
Funciones..103
GDI+..388
GenDeleteSQL.....................................226
GenDocCLI.exe....................................516
Generador...238
Generate DataSnap client classes. 276, 286
GenInsertSQL......................................226
GenSelectSQL......................................226
GenUpdateSQL....................................226
Gestor de proyectos.........................29, 40
GetCommandTypes.............................218
GetConnection.....................................212
GetConnectionFactory..........................212
GetConnectionItems............................212
GetConnectionProperties.....................212
GetDriverFunc.....................................206
GetDriverNames..................................212
GetDriverProperties............................212
GetFieldNames....................................215
GetIndexNames...................................215
GetInstance..123
GetLists...266
GetMem...119
GetStyleObject....................................408
GetTableNames............................215, 219
GlobalDisableFocusEffect.....................390
GlobalMemoryStatus...........................382
GlobalUseDirect2D..............................389
GlobalUseDirect2DSoftware.................389
GlobalUseHWEffects...........................389
GMainFormClass.................................308
Goto...107
GPU...20, 388
GSC...307
GServerController...............................307
GUI..37
GUIActive..306
HAlign..498
Header...357
Height..................................50, 394, 440
Hejlsberg...16
Help Insight...506
Herencia..108
Hexadecimal..89
Hide...319
HideDSAdmin......................................259
High...94, 105
History...135
HitTest...412
HorzIncrement.......................................55
HostName...150
HTMLDoc..................................344, 358
HTMLFile...................................344, 358
IBX 141
IDE..26, 35
Identificadores......................................88
If...100
If-then-else...101
IfThen..429
Implementación28.................................82
Implementation.....................................83
In..97, 99
Inc...118
IndexDefs...158
Indy..20, 299
Info..266
Inherited...............................111, 121, 124
InitialDir..65
Initialization..82
Inprise...18
Inspector de objetos...............................46
Instanciación..111

Int64......86
Integer......86
InterBase......141
Interface......83
Interfaz......82
InternalFilesDir......306
InternetExplorer......310
Interpolation......421
Intersección......99
Interval......165
IntraWeb......20, 301
Inverse......419
IOS......38
ISAPI......304
IScene......417
IsChecked......56, 402
IsExpanded......58
IsMouseOver......428
IsSelected......53
IsSupported......310
ItBounce......421
ItCubic......421
ItemIndex......53, 63
Items Designer......48
ItExponential......422
ItLinear......421
ItQuadratic......421
ItQuartic......421
ItQuintic......421
ItSinusoidal......421
IViewPort3D......437
IW.Browser.Browser......309
IWGlobals......307
IWInitService......304
IWMain......304
IWServerController......307, 314
IWStart......304
IWURLMap......324
Java3D......436
JavaProxyDownloader......484
JSON......340
KeepAliveXXX......261
KeepConnection......211
Keys......421
KSDev......387
Kylix......17
Label......107
Lang......511
Length......94, 105, 118
LibraryName......206
LifeCycle......260, 279p.
Lighting......446
LightType......448
Lines......56
Link to DB DataSource......202
Link to DB Field......187
ListMethods......355
Literales......88
LiveBindings......20, 164, 169
LoadFromFile......157
LoadParamsFromIniFile......211
LoadParamsOnConnect......210
Local (arquitectura)......144
Local Variables......72
Local Variables......74
Locales (variables)......104
Locate......199
Lock......333
LockDataModule......333
LoginPrompt......207, 210
LongBool......88
LongInt......86
LongWord......86
Loop......422
Low......94, 105
LowerCase......176
LtDirectional......448
LtPoint......448
LtSpot......448
MachExceptions......456
MacOS X......26, 38
Manipulación de bits......98
Margins......50p., 395
Matrices dinámicas......93
Matriz......91
Matriz bidimensional......92
Max......55p.
MaxRows......357
MaxThreads......262
MDI......38

MeshCollection....................................444
MetaDataProvider................................349
MethodType.......................................343
Métodos.......................................103, 109
Métodos de clase.................................122
MIDAS...247
Midas.dll......................................166, 246
MidasLib.....................................166, 246
Miembros de clase................................122
Miembros de instancia..........................121
Min..55p.
MinSize...50
MinSupportedVersion..........................310
Mobile Connectors...............................481
Mod...98
Modify Connection..............................149
Módulo...80
Módulo de datos..................................153
Módulos de datos.................................146
Moldeado (conversión) de tipo............100
MultiSelect...53
MyBase...141
NativeInt..86, 381
NativeUInt...86
New...119
New DB Link..188
New Field...159
New Items...37
New LiveBinding.................................173
New Project..36
New Table...151
NewValue..228
Next...338
Nil..117
Normals...442
Not...80, 98
NOT NULL..236
Notación...89
Notify..182
Nuevo proyecto 36
Object Inspector..............................42, 46
Object Pascal..17
ODBC...19, 142
OfFileMustExist....................................65
OldValue...228
OLE DB..141
OLTP...140
OnAction..345
OnAfterRender....................................307
OnAssignedValue.................................189
OnAssigningValue................................189
OnAuthenticate...................................326
OnBeforeRender..................................307
OnBrowserCheck.................................306
OnChange.....................................63, 318
OnCheck..326
OnClick..60, 319
OnCloseSession...................................306
OnConnect...266
OnCreate....................................164, 308
OnCreateInstance................................260
OnDblClick...68
OnDestroy..................................164, 308
OnDestroyInstance..............................260
OnDisconnect......................................266
OnEnter...243
OnExit..64
OnGetClass.................................259, 265
OnGetValue...490
OnHTMLTag.......................................344
OnKeyUp...274
OnLogin...210
OnNewRecord.....................................241
OnNewSession............................306, 320
OnNext...490
OnProcess..421
OnReconcileError.........................227, 245
OnSessionTag......................................307
OnTimer.......................................165, 451
OnUserAuthenticate............................292
OnUserAuthorize.................................293
OnUserTag...307
OpDisableDeletes.................................224
Open..164, 211
OpenGL...388
Options..65, 224
Or...80, 98
Orientado a objetos.............................108
Other...310
OWL..17

Owner...397
PackageName...217
Padding...50p., 395
Paint...401
Paleta de herramientas...36, 42
Param...510
Parámetros...104
Params...158, 206, 216, 221
Parent...397, 433
Parse...191
ParseExpressions...193
Paserver...371, 457
Paserver.config...373
Password...55, 206
PathInfo...343
Pause...422
Percent...336
Perfiles remotos...370
Pila de llamadas...72
PjCamera...452
PjScreen...452
Plantilla de código...131
Point...394
Pointer...384
Points...442
Polimórfica...119
Polimorfismo...108
PolyPascal...16
Pool Data Connections...331
PoolCount...332
PoolSize...262
Popup...72
PopupMenu...57, 59
Port...150, 261, 306
PosClearExpressions...193
PosControlExpressions...193
Position...50, 393, 425, 440
PosSourceExpression...193
Preview...501
PreviewOptions...490
PRIMARY KEY...236
Print...490, 500
PrintOptions...490
Prior...338
Prism...32, 481
Private...110
Procedimientos...103
ProcessEffect...431
Producer...343
ProducerContent...345
Professional...31
Program...81
Progress...432
Project Manager...40
Projection...452
Properties...46, 261
PropertyName...410, 421
Propiedades...46
Propiedades de clase...122
Protected...110
ProviderFlags...226
ProviderName...223, 271
Proxys...276
Public...110
Published...110
Punteros...88
Quartz...388
QuickReports...489
RAD...17
RAD Studio XE2...32
RadPHP...32, 301, 481
Raise...106, 117
Rave Report...489
RDBMS...137, 145
Read...115
RecNo...199
ReconcileError...227
Record...94
Refactor...134
Refactor>Extract Method...134
ReferringURL...308
Refresh...245
RefreshRecord...245
Registro...94
Remarks...510
RemoteServer...271
Repeat...102
ReplaceText...344
Report Wizard...495
ReportBuilder...489

ReportOptions....490
ReportSmith....489
RequestAuth....326
ResolveToDataSet....225
REST....330
RESTContext....347, 355
Result....104
Returns....510
Roles....291
RotationAngle....419, 424, 440
RotationCenter....424
Round....100, 176, 195
RSA....289
RTL....26
RTTI....83
Run....70, 73
Run to Cursor....73
Run Until Return....73
Run-Time Library....28
Rutinas anidadas....105
Safari....310
SafariMobile....310
Save All....41
SaveToFile....157, 164
Scale....419, 424
Scratch-dir....372
Scratchdir....380
SDI....38
Sealed....125
Search....134
Secuencia (generador)....238
See....511
Seealso....511
SelectedItem....63
Self....121
Sender....68
Sentencia compuesta....101
Sentencias....100
Server....257
ServerClassName....271, 332
ServerFunctionExecutor....353
ServerMethodName....273
SessionTimeout....306
Set of....96
SetAsMainForm....308
SetBounds....394
SetFocused....243
SetHost....485
SetLength....93, 118
SetPort....485
SetServerControllerClass....305
Setup_paserver.exe....371
ShadeMode....445
Shaders....388
ShadowColor....431
Shl....98
ShortInt....86
Show....319
Show Documentation....507
ShowMessage....318
ShowReport....490, 500
Shr....98
Single....86
Singleton....123
SizeOf....382, 384
SmallInt....86
SmFlat....445
SmGouraud....445
Sobrecarga....104
Software cliente....145
Sorted....53
SourceComponent....175, 177, 335
SourceExpression....175, 177
SourceMemberName....193, 335
Specular....447
SplitString....127
SQL....216
SQL Window....151
SQLConnection....216
SSLOptions....306
Start....258, 419, 422
Starter....31
StartValue....410, 421
Step Over....73
Stop....258, 419, 422
StopValue....410, 421
StoredProcName....217
String....87
Stroke....409, 426
StrokeDash....425

StrokeThickness....425
StrToDateTime....176
Structure....49, 134
StyleBook....401, 415
StyleFileName....417
StyleLookup....406
StyleName....408
StyleSheet....306
Subconjunto....99
Subrango....91
Summary....510
Superconjunto....99
SVG....404
Sysdba....149
System.Character....87
System.Classes.Persistent....154
System.Types....84, 126
T....408
TabIndex....48
TableName....216
TaCenter....47
Tag....63
TAggregate....158
TagString....344
TAlignLayout....90, 394
TAnimation....419
Target Platforms....40, 368
TargetDirectory....349
TargetUnitName....349
TBevelEffect....431
TBindDBCheckLink....190
TBindDBEditLink....174, 189
TBindDBGridLink....202
TBindDBMemoLink....190
TBindExpression....174p., 180
TBindingsList....180, 182
TBindLink....193, 335
TBindList....196
TBindNavigator....191
TBindPosition....193
TBindScope....183
TBindScopeDB....187p.
TBitBtn....397
TBitmap....113, 431
TBitmapAnimation....420
TBitmapListAnimation....420
TBlurEffect....432
TBlurTransitionEffect....432
TBounds....395
TBrowser....306, 309
TButton....52, 60
TCamera....450
TCharacter....87
TCheckBox....56
TCheckConstraints....158
TChrome....311
TCircle....409, 425
TClientDataSet....141, 156, 204, 271
TColorAnimation....402, 410, 420
TColorKeyAnimation....420
TComboBox....55
TComponent....249, 397
TCone....439
TControl....48, 393
TCube....439
TCustomContentProducer....344
TCustomForm....417
TCustomForm3D....437
TCustomMesh....444
TCustomSQLDataSet....216
TCylinder....439
TDataModule....153, 256
TDataSet....157
TDataSetPageProducer....356
TDataSetProvider....204, 223, 249
TDataSetTableProducer....344, 356
TDataSource....171, 187
TDBXConnection....212
TDBXConnectionFactory....212
TDBXProperties....212
TDCOMConnection....294
TDSAdminClient....277
TDSAuthenticationManager....255, 290
TDSConnectEventObject....266
TDSHTTPServer....346
TDSHTTPServerTransport....248, 346
TDSHTTPService....248
TDSHTTPWebDispatcher....340, 346
TDSProviderConnection....249, 271, 285
TDSProxyDispatcher....482

TDSProxyGenerator....................348, 482
TDSRESTService.................................347
TDSRoleItem.......................................291
TDSServer...................................249, 258
TDSServerClass..........................249, 259
TDSServerMetaDataProvider..............348
TDSServerModule.......................249, 256
TDSTCPServerTransport......248, 261, 288
TEdit..54
TeeChart..20
TEffect...431
TEllipse3D...439
TemplateDir..306
Templates..131
TexCoordinates...................................442
TExpander...58
Text...47, 51
TextAlign.......................................47, 177
Texture..445
TField...........................158, 160, 226, 240
TFieldDef..158
TFilter...431
TFirefox...311
TFloatAnimation................................420
TFloatKeyAnimation........................420p.
TFmxObject.........................393, 397, 401
TForm1..83
TForm3D..437
TfrxDBDataset....................................490
TfrxPDFExport....................................491
TfrxPreview.................................490, 501
TfrxReport..490
TfrxUserDataSet.................................490
TGlowEffect..402
TGradientAnimation...........................420
TGrid3D..439
TGroupBox....................................51, 395
Then...100p.
THttpRequest.....................................326
TImage...................................58, 397, 401
TImageFXEffect..................................431
TImageViewer.......................................67
TIndexDef...158
TIntegerField......................................160
TInternalSQLDataSet..........................230
TIWAppForm......................................308
TIWApplet...311
TIWApplication...................306, 308, 318
TIWAutherEvent.................................326
TIWAutherINI.....................................325
TIWAutherList....................................326
TIWBaseForm.....................................307
TIWButton............................311, 318, 335
TIWCheckBox..............................318, 335
TIWDataModulePool..........................332
TIWDBCheckBox..........................312, 339
TIWDBEdit..................................312, 339
TIWDBListBox.....................................312
TIWDBNavigator..........................312, 339
TIWDBRadioGroup.............................312
TIWEdit................................311, 318, 335
TIWFileReference...............................306
TIWFlash...311
TIWForm...315
TIWList..311
TIWListBox...318
TIWMenu....................................311, 319
TIWProgressBar..................................335
TIWQuickTime....................................311
TIWServerController...........................305
TIWServerControllerBase....................305
TIWSilverlight......................................311
TIWStart...304
TIWURLMap.......................................324
TIWUserSessionBase...........................309
TLabel...54
TLayer3D..452
TLayout.................................50, 402, 409
TLayout3D..438
TLight...448
TLine...397, 425
TListBox...52, 63
TListBoxItem.................................53, 180
TLocalConnection...............................294
TMagnifyEffect...................................432
TMagnifyTransitionEffect....................432
TMainMenu..319
TMaterial..445
TMemo..56
TMemoryStatus...............................382p.

TMenuItem..............................58, 65, 319
TMesh..442
TMeshData......................................442
TModel3D..444
TModelImportServices.......................445
TNumberBox.......................................55
To...102
To-Do List..85
TObserver...187
TObserverMapping...........................174
TObservers..174
TODO..84
Tool Options.....................................469
Tool Palette....................................36, 42
Tools>Options...................................133
TOpenDialog..65
ToStr..176
TPageProducer............................340, 344
TPaintBox..397
TPanel...77, 395
TParam................................158, 216, 221
TPath...402, 404
TPathAnimation..................................420
TPixelateEffect...................................432
TPixelateTransitionEffect....................432
TPlane...439
TPopupMenu..57
TPosition....................................393, 424
TPosition3D..440
Trace Into...73
TReconcileAction................................228
TRectangle...401
TRectAnimation..................................420
TReflectionEffect.................................431
TriangleIndices...................................442
Trigger.................................410, 423, 431
TriggerInverse.............................410, 423
TRoundCube.......................................439
TRoundRect..409
True...88
Trunc...100
Trunc()..65
Try/except..106
TShadowEffect....................................431
TShape3D...444
TSimpleDataSet..................................230
TSphere..439
TSplitter.......................................50, 395
TSQLConnection..........204, 206, 219, 270
TSQLDataSet...............................204, 217
TSQLQuery.................................204, 216
TSQLResolver.....................................226
TSqlServerMethod.......................249, 273
TSQLStoredProc..........................204, 217
TSQLTable...................................204, 216
TStatusBar..183
TStringDynArray.................................126
TStringField..160
TStringGrid..................................202, 212
TStrings..212
TStrokeCube.......................................439
TStyleBook..401
TStyledControl.............................393, 401
TSwitch..425, 429
TTabControl..47
TTabControl19......................................45
TTabItem...46
TText...401, 409
TText3D...441
TTimer...165, 451
TTrackBar......................................56, 425
TTransportFilter...................................251
TTransportFilterItem...........................289
Turbo Pascal...16
TVertScrollBox.......................................51
TWebActionItem.................................343
TWebDirectoryItem.............................349
TWebFileDispatcher............................349
TWebModule.......................................340
TWebRequest......................................345
TWebResponse....................................345
TWideStringArray................................266
TWinControl..401
Type...90
Type-casting..100
Types..83
UInt64..86
UkDelete...227
UkInsert..227
UkModify..227

Ultimate....31
UNativeInt....381
UNICODE....85
UnicodeString....87
Unión....99
Unit....80, 82
Unit1.pas....40
Until....102
UpdateKind....227
UpdateMode....225
UpperCase....176
UpWhereAll....225
UpWhereChanged....225
UpWhereKeyOnly....225
Use Unit....68, 81
User_Name....206
UserRoles....293
UserSession....309, 314
Uses....81, 83, 164
UseSmallScrollBars....51
UsingDesignCamera....450
Value....55, 510
Var....82, 85
Var (parámetros)....104
Variables....85
VBK....17
VCL....26, 38
Vcl.Controls....84
Vcl.Controls.TControl....154
Vector....394
VendorLib....206
Versiones previas....135
View Source....81
View>Templates....131
Visual Component Library....28
VNC....475
Watches....72
WebApplication....308
WebBroker....300
WebDirectories....349
WebFileExtensions....351
WebSnap....300
While....102
WideChar....87
Width....47, 50, 394, 440
Win32....26
Win32ProxyDownloader....484
Win64....26
Windows....382
With....96, 107
Word....86
WordBool....88
WordWrap....498
WOW64....366
WPF....436
Write....115
Writer....349
X....393p.
Xcode....469
XLST....514
XML....157
XMLDoc....506
Xor....98
Y....393
ZLibCompression....289
^....88
:=....97
.dpr....81
.NET....18
.pas....81
.style....407
'style'....408
(*....84
{....84
@....88
//....84
+ (conjuntos)....97
<>....98