PSecurityAttributes;
lInitialCount: LongInt;
lMaximumCount: LongInt;

Danypress

Programación COM en Delphi

Eric Harmon

© 2001 Danypress
[www.danypress.com]

Programación COM en Delphi.
Por Eric Harmon
Copyright versión en castellano © 2001 Danysoft Internacional
Avda. de España 17 - 28100 Alcobendas - Madrid
www.danypress.com
ISBN: 84-923926-
Depósito Legal: M.

Traducido de: Delphi COM Programming
@ 2000 by MTP
ISBN 1-57870-221-6

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 los autores como el equipo de Danysoft hemos revisado el texto para evitar cualquier tipo de error, pero no podemos prometerle que éste o estará siempre libre de errores. Por ello y para que esto no sea una molestia para usted, hemos habilitado en *http://www.danypress.com* un apartado especial para que cada libro que publiquemos permanezca "vivo". Asimismo, podrá enviarnos por email a *editorial@danysoft.com* sus comentarios sobre el libro.

DESCUENTOS ESPECIALES.
Recuerde que Danypress ofrece descuentos especiales a centros de formación y en compras por volumen, consulte con *editorial@danysoft.com* para más detalles en España o con *america@danysoft.com* en Latinoamérica.

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

Tabla de contenidos

Sobre el libro 1
El autor 1
El equipo de revisores del libro 1
Dedicatoria 2
Agradecimientos 2
¡Díganos lo que Piensa! 2

0 **Introducción 5**
A quién va dirigido este libro 5
Cómo está organizado este libro 5
Componentes desarrollados en este libro 7
Aplicaciones de ejemplo 7

1 **Las interfaces en Delphi 9**
Definiendo interfaces 9
La interfaz como un contrato 11
Diferencias entre una interfaz y una clase 11
Una interfaz es inalterable 12
Declarando una interfaz 13
Definiendo GUIDs 13
Implementando una interfaz 15
Implementando IUnknown 16
Implementando IUnknown manualmente 17
Creando, usando y destruyendo interfaces 20
Las referencias a una interfaz se cuentan 21
Obteniendo un puntero a una interfaz 23
Ejemplo: IntfDemo 28
Programando con interfaces 33
Ejemplo: SortDemo - Algoritmos con interfaces 33
Tópicos avanzados sobre interfaces 43
Implementando múltiples interfaces en una clase 43
Propiedades en las interfaces 50
Ejemplo: Demostración gráfica 50
Resumen 58

2 **Las interfaces y COM 59**
Los GUIDs y COM 59
Objetos COM y Fábricas de Clase 60
Objetos COM 61
Fábricas de clases 63
Servidores COM internos al proceso 65
Soporte de múltiples hilos 66
Registrando el servidor 67
Constructores a medida 68
Creando una instancia de un objeto COM interno al proceso. 68
Ejemplo: Distribución unidimensional óptima 71
Servidores COM externos al proceso 97
Instanciación 97
Creando una instancia de un objeto COM externo al proceso 98
Serialización de datos (marshaling) 98
Arrays variantes 99
Creando arrays variantes 102
Accediendo a un array variante 103
Mejorando el rendimiento de los *arrays* variantes 104
Ejemplo: *Array* variante 105
Usando objetos COM con otros lenguajes 109
Resumen 110

3 **Bibliotecas de tipos 111**
Definición de Biblioteca de Tipos 111
Razones para Usar Bibliotecas de Tipos 112
TTypedComObject 112
Creando una biblioteca de tipos en Delphi 115
Usando el Editor de Bibliotecas de Tipos 117
Creando un Visor de Bibliotecas de tipos 127
Interpretando enumeraciones 136
Interpretando interfaces y disp-interfaces 137
Interpretando co-clases 137
Extendiendo TIViewer 138
Resumen 138

4 **Automatización 139**
Definiendo la automatización 139
Interfaces 140
Variantes 140
Disp-interfaces 142
Interfaces Duales 144
Servidores de automatización internos al proceso 144
Ejemplo: Servidor de conversión de unidades de medida 144
CreateOleObject y GetActiveOleObject 148
Ejemplo: Cliente del servidor de conversión de unidades 149
Servidores de automatización externos al proceso 154
HResult y Safecall 154
Automatizando una aplicación existente 156
Eventos y funciones de respuesta 168
Disp-interfaces 168
Interfaces de respuesta 185
Microsoft ADO 198
La filosofía de ADO 199
Obteniendo ADO 200
Instalando ADO en Delphi 200
Conectando a una base de datos 201
Abriendo un conjunto de datos 202
Ejecutando un comando 204
Accediendo a los valores de los campos 205
Manejando errores de bases de datos 205
Ejemplo: aplicación Microsoft ADO 206
Resumen 210

5 **Controles ActiveX y ActiveForms 211**
Usando controles ActiveX existentes en Delphi 211
Microsoft Agent 211
Ejemplo de aplicación Microsoft Agent. 222
Creación de controles ActiveX 228
Razones para crear controles ActiveX 228
Convirtiendo un componente VCL a un Control ActiveX 229
Registrando un control ActiveX 247
Probando el control ActiveX en Visual Basic 247

Añadiendo una página de propiedades a un control ActiveX 247
Distribuyendo controles ActiveX 248
ActiveForms 249
Creando un ActiveForm 249
Probando un ActiveForm 264
Resumen 265

6 **DCOM 267**
Bases de DCOM 267
Programación con DCOM 267
Creando un Servidor DCOM 268
Instalación de un servidor DCOM 272
Creando un cliente DCOM 276
Implementación de conjuntos de datos remotos con DCOM 278
Definiendo la aplicación 279
Creando el servidor 279
Creando el cliente administrativo 293
Creando el cliente punto de venta 300
Resumen 305

7 **Almacenamiento Estructurado 307**
Definiendo el almacenamiento estructurado 307
Programación con almacenamiento estructurado 309
Creando un fichero de almacenamiento estructurado 309
Abriendo un fichero de almacenamiento estructurado existente . . 312
Trabajando con flujos 313
Creando y utilizando almacenes adicionales 316
Iterando a través de un fichero de almacenamiento estructurado . . 316
Comprimiendo un fichero de almacenamiento estructurado 318
Ejemplo: Uso de ficheros de almacenamiento estructurado 319
Ejemplo: Visualizar un fichero de almacenamiento estructurado . . 337
Resumen 341

8 **Almacenamiento Estructurado y OLE 343**
Conjuntos de Propiedades OLE 343
Convenios de los conjuntos de propiedades OLE 344
Las interfaces IPropertySetStorage e IPropertyStorage 344
Un componente para leer y escribir conjuntos de propiedades . . . 356

Ejemplo: Accediendo a propiedades definidas por el usuario . . 361
Ejemplo: Encontrando ficheros por sus propiedades 364
Documentos compuestos y contenedores OLE 369
Activando y desactivando servidores OLE 371
Guardando y cargando objetos en ficheros y flujos 371
Mezcla de menús 371
Usando el Portapapeles 372
Ejemplo: Un contenedor OLE sencillo 373
Resumen 376

9 **Programación del *shell* de Windows 377**
Manejadores de menús contextuales 377
La interfaz *IShellExtInit* 379
La interfaz *IContextMenu* 380
Registrando el manejador de menú contextual 385
Manejadores de *ganchos* de copia 391
La interfaz *ICopyHook* 392
Registrando un manejador de gancho de copia 396
Accesos directos del shell 399
Interfaces de los enlaces del shell 402
Creando un enlace 403
Iconos de la bandeja 412
Mensajes de iconos de bandeja 414
El componente *TDCPTrayIcon* 414
Un ejemplo de aplicación de bandeja 419
Manejadores de páginas de propiedades 421
Creando una plantilla de diálogo 423
Interfaces de un manejador de página de propiedades 424
Implementando el procedimiento de diálogo 431
Implementando la función de respuesta 432
Dónde ir desde aquí 433

Apéndices

A - Lecturas y recursos recomendados 435
Programación en Delphi 435
Programación COM 436

Indice 437

Ejemplo: Accediendo a propiedades definidas por el usuario . . 361
Ejemplo: Encontrando ficheros por sus propiedades 364
Documentos compuestos y contenedores OLE 369
Activando y desactivando servidores OLE 371
Guardando y cargando objetos en ficheros y flujos 371
Mezcla de menús 371
Usando el Portapapeles 372
Ejemplo: Un contenedor OLE sencillo 373
Resumen 376

9 **Programación del *shell* de Windows 377**
Manejadores de menús contextuales 377
La interfaz *IShellExtInit* 379
La interfaz *IContextMenu* 380
Registrando el manejador de menú contextual 385
Manejadores de *ganchos* de copia 391
La interfaz *ICopyHook* 392
Registrando un manejador de gancho de copia 396
Accesos directos del shell 399
Interfaces de los enlaces del shell 402
Creando un enlace 403
Iconos de la bandeja 412
Mensajes de iconos de bandeja 414
El componente *TDCPTrayIcon* 414
Un ejemplo de aplicación de bandeja 419
Manejadores de páginas de propiedades 421
Creando una plantilla de diálogo 423
Interfaces de un manejador de página de propiedades 424
Implementando el procedimiento de diálogo 431
Implementando la función de respuesta 432
Dónde ir desde aquí 433

Apéndices

A - Lecturas y recursos recomendados 435
Programación en Delphi 435
Programación COM 436

Indice 437

Sobre el libro

El autor

Eric Harmon es un desarrollador con 16 años de experiencia en el mundo de la programación. Sus primeros pasos fueron con Turbo Pascal pasando posteriormente a trabajar con Delphi. Eric es uno de los miembros fundadores del TPX, el equipo de colaboradores de TurboPower Software. También ha colaborado con varios artículos en la revista *"Visual Developer Magazine"*.

Eric vive en Florida (USA) con su esposa Tina, su gata Princesa y su perro Oscar. Cuando no programa, disfruta leyendo, con su motocicleta, jugando a las cartas o escuchando música.

El equipo de revisores del libro

El equipo que ha colaborado en la revisión de este libro le ha aportado su amplia experiencia *con Delphi,* ayudando a que sea un libro eminéntemente práctico. Según se estaba escribiendo, se encargaba de revisar el contenido técnico y la organización. Sus comentarios han sido claves para asegurarnos de que *Programación COM en Delphi* se ajusta a las necesidades de los lectores aportándoles una información técnica de alto nivel.

Wayne Niddery es un desarrollador independiente. Su experiencia de 16 años abarca una amplia variedad de campos y lenguajes de programación, siendo Delphi la herramienta preferida desde su aparición. Como miembro del TeamB desde septiembre de 1998, Wayne contribuye habitualmente en los grupos de noticias de Delphi, es uno de los autores de *Borland Delphi How-To* (1995, The Waite Group) y fue el editor técnico de *Delphi 3 Super Bible* (1997, The Waite Group). Wayne trabaja en la ciudad de Toronto en Canada, con su esposa Sonoe, su hijo pequeño Curtis y su perro Patch.

Danny Thorpe es ingeniero en el equipo Delphi y C++Builder de Borland. Entre sus responsabilidades actuales están: la evolución del núcleo de la arquitectura de la VCL, clases visuales, RTL, clases de rejillas, editores de propiedades en tiempo de diseño y la arquitectura del diseñador de formularios. Ha contribuido en cada versión de la linea de productos basados en el Pascal de Borland desde Turbo Pascal 6.0 (1990). Danny ha sido también editor técnico en mas de cuarenta libros de programación sobre Delphi, y es el autor de *Delphi Component Design* (1997, Addison Wesley).

Dedicatoria

Este libro está dedicado a mi profesora, Miss Mac, que me animó a ser más fuerte.

Agradecimientos

Quisiera dar las gracias a todos los que han influido para decidirme a escribir este libro, y a los que han participado para que se convirtiera en una realidad.

Primero, quiero dar las gracias a Jeff Duntemann, él ha sido el responsable de que mi carrera como escritor se llevara a cabo. A Ray Konopka, Xavier Pacheco y Kent Reisdorph que evaluaron mi propuesta inicial y estuvieron de acuerdo en que podría realizarse.

Al equipo de Macmillan que es fantástico, sobre todo a Karen Wachs, Jennifer Garrett, Leah Williams, Ami Frank Sullivan, Alissa Cayton y Elise Walter.

A Ben Riga y Kendyl Uppstrom de Borland, que hicieron posible que tuviera todos los materiales necesarios para crear este libro. A Bruneau Babet también de Borland que me respondió las preguntas técnicas que aparecieron en el camino.

Finalmente, quiero dar las gracias a mis editores técnicos: Danny Thorpe y Wayne Niddery. Danny es uno de los mejores editores técnicos que uno se pueda imaginar para este tipo de libros. Wayne es un excelente programador Delphi que está aprendiendo COM y en muchos aspectos personifica la audiencia de este libro, él me ha ayudado a explicar más claramente los conceptos del libro.

Mis más sinceras disculpas a quien haya podido omitir involuntariamente. Si le he dejado fuera, probablemente sea debido a que no he tratado con usted directamente, por ello quiero dar mi mas sincero agradecimiento a todos los que han contribuido en este proyecto. Si al final es un éxito, en parte es gracias a ellos.

¡Díganos lo que Piensa!

Como lector de este libro, usted es nuestro más importante crítico y comentarista. Valoramos su opinión y queremos saber qué estamos haciendo bien, qué pudiéramos hacer mejor; qué asuntos le gustaría ver publicados así como cualquier sugerencia que esté dispuesto a compartir con nosotros.

Sus comentarios son bienvenidos. Puede enviarnos un fax, un e-mail o escribirnos directamente y hacernos saber qué le gustó o le disgustó de este libro, además de lo que pudiéramos hacer para qué nuestros libros sean más interesantes.

Cuando escriba, asegúrese de mencionar el título del libro y su autor, así como su nombre y dirección de correo electrónico. Revisaremos cuidadosamente sus comentarios y los compartiremos con el autor y editores que han trabajado en este libro.

Fax: +34 91 654 63 82
E-mail: editorial@danysoft.com
Web: www.danypress.com

Dirección: Grupo Danysoft Internacional.
Avda. España 17
Alcobendas 28100
Madrid - España
Web: www.danysoft.com

0 Introducción

Este libro trata sobre la programación COM en Delphi. Hoy día existen en el mercado varios libros excelentes dirigidos a programadores Delphi de diferentes niveles. Asimismo, están disponibles diferentes libros que presentan los detalles del modelo COM. Lo que no existía hasta este momento es un libro que enseñara la programación COM a los desarrolladores Delphi.

A quién va dirigido este libro

Como puede adivinarse del título, este libro está destinado a los programadores Delphi. Más específicamente, este libro ha sido escrito para aquellos programadores que tengan cierta destreza en Delphi, pero no tengan familiaridad con COM, o tengan dificultad en comprender los detalles de la programación COM.

Si ha leído previamente libros o artículos sobre COM, y aún no sabe cómo comenzar a utilizarlo en sus aplicaciones Delphi, ¡no se desespere! Delphi ofrece una gran cantidad de características del lenguaje y asistentes que reducen significativamente o eliminan totalmente los dolores de cabeza asociados a la programación COM.

Este libro supone que Ud. domina la programación en Delphi, así que no ofreceré ninguna información básica sobre la programación Delphi en general. Si necesita afianzar su dominio de la programación en Delphi, por favor consulte el Apéndice A, "Lecturas y recursos recomendados", donde se ofrece una lista de cursos y libros adecuados para todos los niveles.

Cómo está organizado este libro

Este libro está compuesto por nueve capítulos. Los Capítulos del 1 al 5 discuten en detalle las bases del modelo de programación COM y cómo éste ha sido implementado en Delphi. Aunque los capítulos posteriores se apoyan en los conocimientos presentados en capítulos anteriores, una característica agradable de COM es que Ud. puede utilizar partes de él sin comprender completamente el todo. Obviamente, mientras más comprenda, mejor programador COM será en

Delphi, pero podrá empezar a utilizar lo que ha aprendido en cada capítulo antes de pasar al siguiente.

Los Capítulos del 6 al 9 utilizan los conceptos presentados en los Capítulos del 1 al 5 para profundizar en áreas específicas de la programación COM. Por ejemplo, el Capítulo 9 muestra cómo puede extender el *shell* de Windows (escribiendo, por ejemplo, aplicaciones que hacen uso de la bandeja de iconos) mediante COM.

Como resultado, este libro puede ser utilizado de dos formas distintas, pero compatibles. En primer lugar, puede utilizarlo como un tutorial, leyendo cada capítulo en orden. En segundo lugar, Ud. puede utilizar el libro como referencia, volviendo a leer un capítulo específico para reafirmar los conocimientos presentados en ese capítulo. Por esa razón, esta sección ofrece una breve descripción de los capítulos que conforman el libro, para que Ud. pueda fácilmente encontrar lo que necesita.

- El Capítulo 1, "Las interfaces en Delphi", introduce el concepto de interfaz. La interfaz es el elemento básico de la programación COM. En este Capítulo aprenderá que las interfaces son una construcción del lenguaje que tiene su propia utilidad en Delphi. Le mostraré cómo crear aplicaciones Delphi basadas en interfaces, sin siquiera discutir la programación COM.
- El Capítulo 2, "Las interfaces y COM", continúa donde termina el capítulo anterior. En el Capítulo 2 le mostraré cómo crear servidores COM en Delphi. Aprenderá cómo actualizar servidores COM existentes sin "romper" el código existente que depende de una versión anterior de ese servidor.
- En el Capítulo 3, "Bibliotecas de Tipos", introduciré el concepto de biblioteca de tipos. Como verá, las bibliotecas de tipos ofrecen el mecanismo mediante el cual los servidores COM pueden ser utilizados por prácticamente cualquier lenguaje o compilador disponible en el mercado.
- El Capítulo 4, "Automatización", se apoya en los conceptos discutidos en los capítulos anteriores para crear servidores y clientes de automatización. La automatización ofrece el mecanismo mediante el cual una aplicación puede ser controlada por otra.
- En el Capítulo 5, "Controles ActiveX y ActiveForms", discutiré los controles ActiveX y los formularios activos (ActiveForms). Mostraré cuán fácil es utilizar los controles ActiveX existentes en Delphi, y cómo crear sus propios controles ActiveX, que podrá luego ofrecer a programadores que utilicen no sólo Delphi, sino también Visual Basic, Visual C++ y otros lenguajes.
- El Capítulo 6, "DCOM", muestra cómo puede aplicar los conceptos ya aprendidos a una red. Aprenderá a crear aplicaciones distribuidas que pueden comunicarse entre sí a través de una Red de Área Local o Extensa.
- En el Capítulo 7, "Almacenamiento Estructurado", introduciré el almacenamiento estructurado, que es un mecanismo para almacenar datos

de diferentes tipos dentro de un mismo fichero. Verá cómo utilizar las interfaces COM para usar ficheros de almacenamiento estructurado en sus aplicaciones Delphi.

- El Capítulo 8, "Almacenamiento Estructurado y OLE", se apoya en el Capítulo 7, y discute los conjuntos de propiedades OLE y los contenedores OLE. Los conjuntos de propiedades OLE ofrecen un mecanismo universal para almacenar datos "marcados" en un fichero de almacenamiento estructurado. Por ejemplo, una aplicación puede almacenar el nombre del autor en un documento, y luego otra aplicación puede leer ese nombre de autor del fichero sin saber nada acerca de la estructura interna del fichero.
- En el Capítulo 9, "Programación del *shell* de Windows", explicaré cómo crear varios de los tipos más frecuentes de extensiones del *shell* de Windows, incluyendo manejadores de menús contextuales, aplicaciones que utilizan la bandeja de iconos, y otras.

Componentes desarrollados en este libro

Aunque este no es un libro sobre el desarrollo de componentes, hay varios lugares en el texto donde tiene sentido crear un componente para encapsular la idea presentada. Por esa razón, el código fuente que acompaña a este libro incluye un paquete de Delphi llamado DCP que contiene dichos componentes. El paquete DCP se ofrece en tres versiones diferentes: DCP3 para Delphi 3, DCP4 para Delphi 4 y DCP5 para Delphi 5 y 6.

El paquete incluye los siguientes componentes:

- **EventSink** – Ofrece un mecanismo para responder fácilmente a eventos de automatización en clientes de automatización (utilizado en el Capítulo 4).
- **PropertySet** – Permite leer y escribir conjuntos de propiedades OLE (utilizado en el Capítulo 8).
- **TrayIcon** – Permite controlar iconos de la bandeja (utilizado en el Capítulo 9).

Aplicaciones de ejemplo

Cada capítulo de este libro incluye varios ejemplos que ayudan a la comprensión de los conceptos presentados. Con unas pocas excepciones, todos los ejemplos compilan desde Delphi 3 a Delphi 6. En varios casos encontrará en ellos directivas de compilación destinadas a garantizar que la aplicación funcione bajo Delphi 3, y en un par de casos, el ejemplo no funcionará en Delphi 3 simplemente porque hace uso de características del lenguaje o las bibliotecas introducidas en Delphi 4. En esos casos, se hace mención explícita de ello en el libro.

Recuerde que el código fuente de los ejemplos y componentes de libro puede ser descargado desde *http://www.danypress.com*.

de diferentes tipos dentro de un mismo fichero. Verá cómo utilizar las interfaces COM para usar ficheros de almacenamiento estructurado en sus aplicaciones Delphi.

- El Capítulo 8, "Almacenamiento Estructurado y OLE", se apoya en el Capítulo 7, y discute los conjuntos de propiedades OLE y los contenedores OLE. Los conjuntos de propiedades OLE ofrecen un mecanismo universal para almacenar datos "marcados" en un fichero de almacenamiento estructurado. Por ejemplo, una aplicación puede almacenar el nombre del autor en un documento, y luego otra aplicación puede leer ese nombre de autor del fichero sin saber nada acerca de la estructura interna del fichero.
- En el Capítulo 9, "Programación del *shell* de Windows", explicaré cómo crear varios de los tipos más frecuentes de extensiones del *shell* de Windows, incluyendo manejadores de menús contextuales, aplicaciones que utilizan la bandeja de iconos, y otras.

Componentes desarrollados en este libro

Aunque este no es un libro sobre el desarrollo de componentes, hay varios lugares en el texto donde tiene sentido crear un componente para encapsular la idea presentada. Por esa razón, el código fuente que acompaña a este libro incluye un paquete de Delphi llamado DCP que contiene dichos componentes. El paquete DCP se ofrece en tres versiones diferentes: DCP3 para Delphi 3, DCP4 para Delphi 4 y DCP5 para Delphi 5 y 6.

El paquete incluye los siguientes componentes:

- **EventSink** – Ofrece un mecanismo para responder fácilmente a eventos de automatización en clientes de automatización (utilizado en el Capítulo 4).
- **PropertySet** – Permite leer y escribir conjuntos de propiedades OLE (utilizado en el Capítulo 8).
- **TrayIcon** – Permite controlar iconos de la bandeja (utilizado en el Capítulo 9).

Aplicaciones de ejemplo

Cada capítulo de este libro incluye varios ejemplos que ayudan a la comprensión de los conceptos presentados. Con unas pocas excepciones, todos los ejemplos compilan desde Delphi 3 a Delphi 6. En varios casos encontrará en ellos directivas de compilación destinadas a garantizar que la aplicación funcione bajo Delphi 3, y en un par de casos, el ejemplo no funcionará en Delphi 3 simplemente porque hace uso de características del lenguaje o las bibliotecas introducidas en Delphi 4. En esos casos, se hace mención explícita de ello en el libro.

Recuerde que el código fuente de los ejemplos y componentes de libro puede ser descargado desde *http://www.danypress.com*.

1

Las interfaces en Delphi

A partir de la versión 3 de Delphi, Borland introdujo en el lenguaje la palabra reservada **interface**. Las interfaces son la piedra angular de la programación en el modelo COM, aunque también pueden ser usadas en el desarrollo de aplicaciones que no precisen COM. Por tal razón, este capítulo hace énfasis en las interfaces como elemento del lenguaje. Casi todo lo que aprenderá en él se refiere 100% a COM, pero la palabra COM no se mencionará con mucha frecuencia.

A excepción de compartir el mismo nombre, el término *interfaz* del que hablaremos aquí no tiene que ver en absoluto con la sección de interfaz de las unidades (*units*) de Object Pascal. La utilización dual de la palabra clave **interface** no debe ser fuente de confusión, ya que esta utilización se produce en contextos completamente diferentes.

Definiendo interfaces

Quizás la manera más simple de entender una interfaz es pensando que equivale, más o menos, a una clase abstracta. Por si no está familiarizado con las clases abstractas, son clases que definen un comportamiento, pero no lo implementan por sí mismas. En lugar de ello se apoyan en sus clases descendientes para suministrar una implementación. Consideremos por ejemplo la siguiente declaración de clase:

```
type
  TFormattedNumber = class
  public
    function FormattedString: string; virtual; abstract;
  end;
```

Esta es una clase abstracta, y deberíamos reconocerla como la clase base para una serie de clases derivadas, todas las cuales tienen algo en común: todas ellas tienen la capacidad de formatear algún tipo de número mediante el método *FormattedString*.

Nunca crearemos una instancia de una clase abstracta (o por lo menos, nunca *deberíamos* crear una instancia de una clase abstracta). La clase se define únicamente para dictar las leyes a sus clases descendientes. Consideremos el siguiente fragmento de código:

```
var
  MyNumber: TFormattedNumber;
begin
  MyNumber := TFormattedNumber.Create;
  MyNumber.Free;
end;
```

Este código es correcto y el compilador sólo nos dará una advertencia:

> `Constructing instance of 'TFormattedNumber' containing abstract methods.`
> (Creación de una instancia de *TFormattedNumber*, que contiene métodos abstractos).

El código anterior se compila y ejecuta sin ningún problema. Sin embargo, al tratar de usar *MyNumber*, Delphi lanzará la excepción *EAbstractError*, indicando que intentamos llamar a un método abstracto.

Para que la clase *TFormattedNumber* sea útil, deberemos derivar clases de ella:

```
type
  TFormattedNumber = class
  public
    function FormattedString: string; virtual; abstract;
  end;
  TFormattedInteger = class(TFormattedNumber)
  private
    FValue: Integer;
  public
    function FormattedString: string; override;
  end;
  TFormattedDouble = class(TFormattedNumber)
  private
    FValue: Double;
  public
    function FormattedString: string; override;
  end;
```

Una vez definidas las clases, las podremos usar de la siguiente forma:

```
procedure ShowFormattedNumber(F: TFormattedNumber)
begin
  ShowMessage(F.FormattedString);
end;
```

Al procedimiento *ShowFormattedNumber* no le importa si le pasamos un *TFormattedInteger* o un *TFormattedDouble*. Simplemente llama al método *FormattedString* de la clase adecuada.

Esto no debería ser nuevo para un programador experimentado de Delphi. Veamos ahora la implementación del mismo concepto utilizando una interfaz.

Veremos cómo ésta nos permite obtener los mismos resultados, con algunos efectos colaterales interesantes y muy útiles:

```
type
  IFormattedNumber = interface
    function FormattedString: string;
  end;
```

Las similitudes entre las declaraciones de *TFormattedNumber* e *IFormattedNumber* deberían ser evidentes. Sin embargo, hay algunas sutiles e importantes diferencias que es necesario comprender. Las siguientes secciones examinan en detalle esas diferencias.

La interfaz como un contrato

Conceptualmente, una interfaz no es más que un contrato entre el que implementa la interfaz y el usuario de la misma. Al definir una interfaz, estamos diciendo realmente: "Veo la necesidad de esta funcionalidad. Reconozco que esta funcionalidad podría ser implementada de muchas formas diferentes. No me importa cómo se implemente físicamente la funcionalidad, pero deberá atenerse a estas especificaciones". El usuario de la interfaz puede escribir su código basándose en las especificaciones, sin preocuparse de posibles cambios en la implementación de éstas.

En el ejemplo anterior, decidimos crear una clase (o realmente un conjunto de clases) que soportara el formateo de diferentes tipos de números. Las clases derivadas podrán incluir clases para formatear números enteros, reales, hexadecimales, etc.

Una interfaz no es una clase: es una interfaz. Por lo tanto, debe ser declarada como tal. En la siguiente sección examinaremos con más detalle la importancia de esta afirmación.

Diferencias entre una interfaz y una clase

Aunque las interfaces tienen mucho en común con las clases abstractas, existen algunas diferencias importantes entre ellas. Examinémoslas en detalle. Consideremos de nuevo las dos declaraciones anteriores:

```
type
  TFormattedNumber = class
  protected
    function FormattedString: string; virtual; abstract;
  end;
  IFormattedNumber = interface
    function FormattedString: string;
  end;
```

Las interfaces tienen las siguientes características:

- Una interfaz se declara con un tipo **interface** y no con un tipo **class**. Por convenio, los nombres de las interfaces empiezan con la letra I, y los nombres de clases con la letra T.
- Todas las interfaces heredan, directa o indirectamente, de *IUnknown*. De la misma forma que *TObject* es la raíz de todas las clases en Delphi, *IUnknown* es la raíz de todas las interfaces en COM. Más adelante en este capítulo examinaremos en detalle la interfaz *IUnknown*.
- No se puede crear una instancia de una interfaz. El siguiente código es ilegal:

```
var
    MyNumber: IFormattedNumber;
begin
    MyNumber := IFormattedNumber.Create;  // ilegal !!
end;
```

  El compilador genera el siguiente mensaje de error al intentar compilar el código anterior:

```
Object or class type required
```

  (Se necesita un objeto o tipo clase)

  Esta restricción es una garantía de seguridad. Aunque es factible instanciar explícitamente una clase abstracta, esta limitación le permite al compilador reforzar el concepto de que no se puede instanciar una interfaz.

- En una interfaz no se pueden especificar directivas de acceso. Todos los métodos definidos en ella son públicos, y no se pueden incluir directivas de acceso (ni siquiera **public**) en la declaración de la interfaz.
- Una interfaz no puede declarar variables. Una interfaz sólo determina qué funcionalidad se va a proporcionar. No establece restricciones respecto a la forma de conseguir esa funcionalidad. Si estuviera permitido incluir variables miembros en la declaración de una interfaz, estaríamos fijando en cierta medida los medios a utilizar para conseguir la funcionalidad deseada.
- Todas las funciones y procedimientos declarados en una interfaz son, por definición, virtuales y abstractos. No es necesario declararlos como tal; de hecho, es ilegal hacerlo.

Una interfaz es inalterable

Después de definirla y publicarla, una interfaz queda "grabada en piedra". Ya no puede ser modificada. Por supuesto, durante el proceso de desarrollo de la interfaz, se realizarán revisiones y cambios en la misma y en su código asociado, pero una vez publicada para ser usada por otros, ya no será posible efectuar más cambios. Si necesitamos ampliar una interfaz, podemos crear una nueva "versión" de la misma.

Por ejemplo, si decidimos añadir un método *SetCaption* a la interfaz *IFormattedNumber*, deberíamos crear una nueva interfaz *IFormattedNumber2*:

```
type
  IFormattedNumber2 = interface
    function FormattedString: string;
    procedure SetCaption(ACaption: string);
  end;
```

Cualquier código existente que aún utilice la interfaz *IFormattedNumber* continuará funcionando sin modificaciones. El nuevo código que se desarrolle podrá aprovechar las ventajas de la nueva interfaz *IFormattedNumber2*.

Nota

El hecho de declarar la nueva interfaz *IFormattedNumber2* no significa tener que reimplementarla completamente. Si todo lo que hace *IFormattedNumber2* es añadir el nuevo método *SetCaption*, se puede derivar *IFormattedNumber2* de *IFormattedNumber*:

```
type
  IFormattedNumber2 = interface(IFormattedNumber)
    procedure SetCaption(ACaption: string);
  End;
```

Ahora ya sabemos lo que es una interfaz y cómo se relaciona con (y difiere de) una clase abstracta. En la próxima sección veremos cómo declarar una interfaz.

Declarando una interfaz

Usted ya ha visto un ejemplo de una declaración sencilla de interfaz:

```
type
  IFormattedNumber = interface
    [ '{2DE825C1-EADF-11D2-B39F-0040F67455FE}']
    function FormattedString: string;
  end;
```

Se advierte inmediatamente la presencia de una nueva cadena de caracteres justamente debajo del nombre de la interfaz. A esta cadena se la conoce como GUID (*Globally Unique Identifier* - Identificador Global Único). Todas las interfaces COM, así como ciertas interfaces que en principio vayan a utilizarse sólo internamente en nuestras aplicaciones, requieren un GUID para funcionar adecuadamente.

Definiendo GUIDs

Un GUID es un número único de 16 bytes. El algoritmo para crear un GUID es muy complejo, y está definido por la *Open Software Foundation*. Este algoritmo toma en cuenta, entre otros datos: la fecha y hora actual del sistema, el número de proceso actual del programa usado para generar el GUID y el identificador único

de su tarjeta de red si la tiene instalada. Afortunadamente, no es necesario comprender el algoritmo para la generación de un GUID.

Si Ud. tiene una tarjeta de red, se garantiza que todos los GUIDs generados en su ordenador serán únicos. Todas las tarjetas de red contienen un número de serie único (conocido como *dirección MAC*), que se incluye como parte del GUID. Incluso si no tiene una tarjeta de red, la unicidad del GUID estará estadísticamente garantizada, debido al gran número de dígitos y al algoritmo usado para generarlos.

Es de vital importancia que al crear una nueva interfaz se genere un nuevo y único GUID. Nunca se debe copiar un GUID de una interfaz a otra.

Delphi facilita la generación de GUIDs. Simplemente debemos posicionar el cursor en el lugar del código fuente donde deseamos insertar el GUID y pulsar **Ctrl+May+G**. Internamente, Delphi hace una llamada a la función *CoCreateGuid* del API de Windows para generarlo.

El Listado 1.1 nos presenta un ejemplo que llama directamente a *CoCreateGuid* para generar una secuencia de GUIDs.

Listado 1.1: Formulario principal de la aplicación *GUIDDemo*

```
unit MainForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ComObj, ActiveX;
type
  TForm1 = class(TForm)
    btnGenerate: TButton;
    Memo1: TMemo;
    procedure btnGenerateClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.btnGenerateClick(Sender: TObject);
var
  Guid: TGUID;
begin
  CoCreateGuid(Guid);
  Memo1.Lines.Add(GuidToString(Guid));
end;
end.
```

La Figura 1.1 muestra la aplicación *GUIDDemo* en acción.

Figura 1.1: El ejemplo *GUIDDemo* muestra cómo llamar a *CoCreateGuid*.

Delphi define la estructura TGUID como sigue:

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

Los cuatro campos en TGUID se corresponden con las cuatro partes de la estructura de un GUID. Los GUIDs se pueden representar en Delphi de dos formas. Tomando como ejemplo el GUID de la interfaz *IFormattedNumber*, las siguientes declaraciones son idénticas:

```
MyGuid := ['{2DE825C1-EADF-11D2-B39F-0040F67455FE}'];
MyGuid: TGUID = (D1: $2DE825C1; D2: $EADF;
                 D3: $11D2; D4: ($B3,$9F,$00,$40,$F6,$74,$55,$FE));
```

Es evidente que la primera construcción es más fácil de leer.

Declarar la interfaz es el primer paso para la creación de una interfaz funcional. Una vez realizada la declaración de la interfaz, debemos implementarla. En la próxima sección veremos cómo implementar una interfaz.

Implementando una interfaz

Como mencionamos en la sección anterior, no se puede instanciar una interfaz. Entonces, ¿cómo se puede usar la interfaz *IFormattedNumber*?.

La respuesta consiste en que se puede crear una clase que implemente la interfaz:

```
type
  TFormattedInteger = class(TObject, IFormattedNumber)
  private:
    FValue: integer;
  public:
```

```
    constructor Create(AValue: Integer);
    function FormattedString: string;
    procedure SetValue(AValue: Integer);
  end;
```

Ahora podemos completar nuestra clase de la forma habitual:

```
constructor TFormattedInteger.Create(AValue: Integer);
begin
  inherited Create;
  FValue := AValue,
end;
function TFormattedInteger.FormattedString: string;
begin
  Result := 'The value is ' + IntToStr(FValue);
end;
procedure TFormattedInteger.SetValue(AValue: Integer);
begin
  FValue := Value;
end;
```

Si intentamos compilar el código en este momento, el compilador nos mostrará los siguientes errores:

```
Undeclared identifier 'QueryInterface'
```
(Identificador no declarado 'QueryInterface')

```
Undeclared identifier '_AddRef'
```
(Identificador no declarado '_AddRef')

```
Undeclared identifier '_Release'
```
(Identificador no declarado '_Release')

¿De dónde provienen estos identificadores, si no los hemos definido en nuestro código? Son funciones declaradas en la interfaz *IUnknown*. Recuerde que, en última instancia, todas las interfaces descienden de *IUnknown*. Sin embargo, aún no hemos explicado exactamente qué es *IUnknown*.

Implementando IUnknown

Si buscamos el fichero *System.pas* en el código fuente de la VCL, encontraremos la siguiente declaración:

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

IUnknown es la raíz de todas las interfaces, y ha sido definida por Microsoft.

QueryInterface, *_AddRef*, y *_Release* son funciones que deben ser implementadas por todos los descendientes de *IUnknown*, lo cual significa que deben ser implementadas por todas las clases que implementan una interfaz.

Invirtamos un poco de tiempo en examinar en detalle la interfaz *IUnknown*.

QueryInterface

Esta es una función que pide un puntero a una interfaz. Si la interfaz es implementada por el objeto en cuestión, *QueryInterface* devuelve la interfaz a través del parámetro *Obj*, y devuelve el valor 0. Si la interfaz no es implementada por el objeto, devuelve la constante definida por Microsoft *E_NOINTERFACE*.

_AddRef

Las referencias a una interfaz siempre *se cuentan*; por lo tanto, cuando se obtiene un puntero a una interfaz soportada por un objeto, se incrementa el contador de referencias al objeto. Cuando se termina de trabajar con la interfaz, se decrementa el contador de referencias al objeto. Cuando el contador de referencias alcanza el valor cero, el objeto es automáticamente destruido. *_AddRef* es la función responsable de incrementar el contador de referencias. Cómo almacenar físicamente el contador de referencias es asunto nuestro; generalmente se utiliza una variable entera para contenerlo.

_Release

_Release es la función responsable de decrementar el contador de referencias. Cuando el contador alcanza el valor de cero, el objeto es automáticamente destruido.

El conteo de referencias se analizará con más detalle en una sección posterior. Por ahora, simplemente tenga presente que las referencias a una interfaz se cuentan, y que *_AddRef* incrementa el contador de referencias y *_Release* lo decrementa.

Implementando IUnknown manualmente

Hasta el momento no podemos compilar *TFormattedNumber* porque no implementa la interfaz *IUnknown*. Es muy sencillo añadir las funciones necesarias.

Echemos un vistazo a la nueva declaración de *FormattedInteger*, que se presenta en el Listado 1.2.

Listado 1.2: Implementación de *IUnknown*

```
type
  TFormattedInteger =  class(TObject, IFormattedNumber)
  private:
    FRefCount: Integer;
    FValue: Integer;
```

```
  public:
    constructor Create(AValue: Integer);
    // interfaz IUnknown
    function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
    //interfaz IFormattedNumber
    function FormattedString: string; virtual;
    procedure SetValue(AValue: Integer);
  end;
constructor TFormattedInteger.Create(AValue: Integer);
begin
  inherited Create;
  FValue := AValue;
end;
// Interfaz IUnknown
function TFormattedInteger.QueryInterface(const IID: TGUID;
out Obj) : HResult; stdcall;
const
  E_NOINTERFACE = $80004002;
begin
  if GetInterface(IID, Obj) then
    Result := 0
  else
    Result := E_NOINTERFACE
end;
function TFormattedInteger._AddRef: Integer; stdcall;
begin
  Inc(FRefCount);
  Result := FRefCount;
end;
function TFormattedInteger._Release: Integer; stdcall;
begin
  Dec(FRefCount);
  Result := FRefCount;
  if FRefCount = 0 then
    Destroy;
end;
// Interfaz IFormattedNumber
function TFormattedInteger.FormattedString: string;
begin
  Result := 'The value is' + IntToStr(FValue);
end;
function TFormattedInteger.SetValue(AValue: Integer);
begin
  FValue := Value;
end;
```

Por el momento, vamos a ignorar la implementación de *QueryInterface*. Por ahora, aceptemos que es una implementación estándar de la función. Posteriormente en este capítulo examinaremos *QueryInterface* más detalladamente.

Como podemos observar, las funciones *_AddRef* y *_Release* simplemente incrementan y decrementan la variable *FRefCount*. Cuando el valor de la variable es cero el objeto es destruido.

Usando TInterfacedObject para implementar IUnknown automáticamente

Afortunadamente, no hay necesidad de escribir todo este código cada vez que creamos una clase que implementa una interfaz. Delphi ofrece una clase llamada *TInterfacedObject* que ya implementa *IUnknown* por nosotros.

El Listado 1.3 muestra el fragmento del código fuente de *TInterfacedObject* que es de interés en esta discusión. *TInterfacedObject* está definida en *System.pas*.

Listado 1.3: Código fuente de *TInterfacedObject*

```
type
  TInterfacedObject = class(TObject, IUnknown)
  protected
    FRefCount: Integer;
    function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  public
    procedure BeforeDestruction; override;
    property RefCount: Integer read FRefCount;
end;
procedure TInterfacedObject.BeforeDestruction;
begin
  if RefCount  0 then
    Error(reInvalidPtr);
end;
function TInterfacedObject.QueryInterface(Const IID: TGUID; out Obj):
HResult;
const
  E_NOINTERFACE = $80004002;
begin
  if GetInterface(IID, Obj) then Result := 0 else Result := E_NOINTERFACE;
end;
function TInterfacedObject._AddRef : Integer;
begin
  Result := InterlockedIncrement(FRefCount);
end;
function TInterfacedObject._Release: Integer;
begin
  Result := InterlockedDecrement(FRefCount);
  if Result = 0 then
    Destroy;
end;
```

Aparte del uso de las funciones *InterlockedIncrement* e *InterlockedDecrement* para garantizar el funcionamiento seguro en un entorno multihilo, y de la propiedad *RefCount*, la implementación de *IUnknown* es prácticamente idéntica a la que hemos desarrollado nosotros. Sin embargo, la existencia de esta clase implica que, en vez de derivar clases de *TObject* e implementar repetitivamente *IUnknown*, podemos simplemente derivar nuestras clases de *TInterfacedObject*.

No obstante, hay ocasiones en las cuales Ud. desearía brindar su propia implementación de la interfaz *IUnknown*. Posteriormente en este capítulo, en la sección "Evitando el conteo automático de referencias" estudiaremos las razones por las cuales desearíamos realizar lo anterior.

Creando, usando y destruyendo interfaces

Una vez escrito el código para la clase, ésta se puede usar como cualquier otra. Por ejemplo:

```
var
  MyNumber: TFormattedInteger;
begin
  MyNumber := TFormattedInteger.Create(12);
  ShowMessage(MyNumber.FormattedString);
  MyNumber.Free;
end;
```

Probablemente se esté preguntando qué ventajas tiene la declaración de una interfaz, si aparentemente no aporta beneficio alguno. El fragmento de código anterior puede ser escrito de un modo diferente. Echemos vistazo al siguiente código, funcionalmente equivalente al del ejemplo anterior:

```
var
  MyNumber: IFormattedNumber;
begin
  MyNumber := TFormattedInteger.Create(12);
  ShowMessage(MyNumber.FormattedString);
end;
```

Hay varios puntos de interés en este fragmento de código. En primer lugar, la variable *MyNumber* es definida del tipo *IFormattedNumber* en lugar de *TFormattedInteger*. En este ejemplo *MyNumber* es un puntero a una interfaz, no un puntero a una instancia de un objeto. Hay una distinción importante que hacer aquí. *MyNumber* no tiene acceso a ninguna información, aparte de la definida en la interfaz *IFormattedNumber*. Por ejemplo, el siguiente código no compilará:

```
MyNumber.SetValue(10);
```

El compilador se quejará diciendo:

```
Undeclared identifier: 'SetValue'
```

(Identificador no declarado: 'SetValue')

Cualquier variable de tipo IFormattedNumber sólo "ve" la declaración de IFormattedNumber, incluso si fue creada a partir de un objeto TFormattedInteger, como en el código anterior.

Otro importante aspecto a tener en cuenta es que este código no libera explícitamente MyNumber. Esto puede parecer una fuente de "goteos" de memoria, pero realmente la variable MyNumber se libera automáticamente al final del procedimiento. Esto requiere algunas aclaraciones.

Las referencias a una interfaz se cuentan

Como mencionamos previamente, las referencias a una interfaz están sometidas a un mecanismo de conteo. Lo que no hemos explicado, sin embargo, es cómo funciona exactamente el mecanismo del contador. Delphi trabaja duro entre bastidores para hacer el conteo de referencias tan invisible como es posible. Debo indicar que no es el caso de la mayoría de los lenguajes. Si, por ejemplo, estamos escribiendo código en C++ (¡pero no en C++ Builder!), necesitaremos llamar manualmente a *AddRef* y *Release* para incrementar y decrementar, respectivamente, el contador de referencias.

¿Hubo un error tipográfico en el párrafo anterior?. Cuando examinamos *TFormattedInteger*, implementábamos las funciones *_AddRef y _Release*. Ahora estamos hablando de *AddRef* y *Release*, sin el subrayado inicial. Los nombres originales asignados por Microsoft a estas funciones fueron *AddRef* y *Release*. Borland cambió los nombres, anteponiéndoles el carácter '_' a fin de reforzar la idea de que no las llamaremos explícitamente.

Para poder apreciar todo lo que Delphi hace por nosotros, tomemos un momento para considerar cómo escribiríamos código que use una interfaz en lenguaje de programación tradicional. Para ello usaremos la sintaxis de Delphi, ya que es posible que no esté familiarizado con otros lenguajes como C++ o C.

```
procedure DoSomethingWithInterfaces;
var
  MyIntf: IFormattedNumber;
begin
  MyIntf := TFormattedInteger.Create(12);
  MyIntf.AddRef;
  ShowMessage(MyIntf.FormattedString);
  MyIntf.Release;

  MyIntf := TFormattedHexInteger.Create(1024);
  MyIntf.AddRef;
  ShowMessage(MyIntf.FormattedString);
  MyIntf.Release;
end;
```

Tan pronto como obtengamos una referencia a una interfaz (en el ejemplo, esto se hace en la primera línea de código), es nuestra responsabilidad realizar una llamada explícita a la función *AddRef* para añadir una referencia a la interfaz. Mientras alguien mantenga una referencia a la interfaz de un objeto, el objeto no será destruido. Por consiguiente, necesitamos incrementar el contador de referencias del objeto lo antes posible.

Después de llamar a *AddRef*, podemos usar la interfaz para llamar a cualquiera de sus métodos. Una vez que hemos terminado con la interfaz, llamamos a *Release* para decrementar el contador de referencias. Dentro de la función *Release* se evalúa si el contador de referencia es cero, en cuyo caso el objeto es destruido automáticamente.

Ahora veamos qué necesitamos hacer en Delphi para obtener los mismos resultados:

```
procedure DoSomethingWithInterfaces;
var
  MyIntf: IFormattedNumber;
begin
  MyIntf := TFormattedInteger.Create(12);
  ShowMessage(MyIntf.FormattedString);

  MyIntf := TFormattedHexInteger.Create(1024);
  ShowMessage(MyIntf.FormattedString);
end;
```

¡Eso es todo! Podemos escribir código como si *_AddRef* y *_Release* ni siquiera existiesen. Si no llamamos a *_AddRef* y *_Release*, entonces ¿cómo (y cuándo) son llamadas? La respuesta estriba en que Delphi hará las llamadas automáticamente por nosotros en el momento oportuno. Echemos un vistazo entre bastidores para ver exactamente qué está haciendo Delphi por nosotros. El código que Delphi introduce automáticamente dentro de nuestra aplicación está marcado como comentario. Observe que Delphi no añade físicamente este código a nuestros ficheros fuente - es simplemente una representación hipotética de lo que será añadido al ejecutable cuando compilemos la aplicación.

```
procedure DoSomethingWithInterfaces;
var
  MyIntf: IformattedNumber;
begin
  MyIntf := TFormattedInteger.Create(12);
  MyIntf._AddRef;  // A□adido por Delphi
  ShowMessage(MyIntf.FormattedString);
  MyIntf._Release;  // A□adido por Delphi
  MyIntf := TFormattedHexInteger.Create(1024);
  MyIntf._AddRef;  // A□adido por Delphi
  ShowMessage(MyIntf.FormattedString);
  MyIntf._Release;  // A□adido por Delphi
end;
```

Por cierto, si alguna vez desea forzar la liberación de una interfaz, basta con asignarle el valor **nil** a la variable interfaz:

```
MyIntf := nil;
```

Obteniendo un puntero a una interfaz

Delphi nos ofrece varias formas de obtener un puntero a una interfaz. Podemos obtener un puntero a una interfaz a partir de un objeto que la implemente, o a través de un puntero a otra interfaz. Más adelante en este mismo capítulo abordaremos las interfaces múltiples en la sección "Temas avanzados sobre interfaces".

Asignación directa

La forma más sencilla de obtener una interfaz a partir de un objeto es por asignación directa. Las clases son compatibles por tipo con las interfaces que implementan, de forma que podemos escribir el siguiente código:

```
var
  MyInteger: TFormattedInteger;
  MyNumber: IFormattedNumber;
begin
 MyInteger := TFormattedInteger.Create(12);
 MyNumber := MyInteger;
end;
```

Este código se comprueba durante la compilación, y si el objeto en cuestión no soporta la interfaz, obtendremos un error de compilación. En ocasiones, sin embargo, no sabemos si un objeto soporta una interfaz, y nos gustaría averiguarlo en tiempo de ejecución. La función *GetInterface* permite hacer exactamente esto.

GetInterface

La función *GetInterface* se define de la siguiente manera:

```
function TObject.GetInterface(const IID: TGUID; out Obj): Boolean;
```

El parámetro *IID* representa el GUID de la interfaz de la cual deseamos obtener un puntero.

```
var
  MyObject: TObject;
  MyNumber: IFormattedNumber;
begin
  MyObject := TFormattedInteger.Create(12);
  // 
  if MyObject.GetInterface(IFormattedNumber, MyNumber) then
    ShowMessage(MyNumber.FormattedString);
end;
```

Recuerde que la implementación de *QueryInterface* que vimos con anterioridad llamaba a la función *GetInterface* para obtener la interfaz en cuestión.

El operador as

Delphi nos hace las cosas aún más fáciles en la obtención de una interfaz, si sabemos que está soportada por un objeto dado. Delphi sobrecarga el operador **as** de forma que podemos escribir código como éste:

```
var
 MyObject: TObject;
 MyNumber: IFormattedNumber;
begin
  MyObject := TFormattedInteger.Create(12);
  // 
  MyNumber := MyObject as IFormattedNumber;
  ShowMessage(MyNumber.FormattedString);
end;
```

La diferencia entre este código y el que usa la función *GetInterface* es que, en este ejemplo, en caso de que *MyNumber* no soporte la interfaz *IFormattedNumber*, se lanzará una excepción.

En general, podemos usar el operador **as** si sabemos que el objeto en cuestión soporta una interfaz determinada, o que sería erróneo que el objeto no la soportara. Por el contrario, usaremos la función *GetInterface* para conocer si un objeto soporta una interfaz, sin que se lance una excepción en el caso de que no lo haga.

El uso del operador **as** difiere del de la función *GetInterface* en otro importante aspecto. *GetInterface* es un método de *TObject* y sólo puede ser usado sobre objetos. El operador **as** puede ser usado sobre objetos que implementan *IUnknown* (como está establecido en Delphi) y sobre las propias interfaces. Por ejemplo:

```
procedure DoSomethingWithObject(o: TObject);
var
 MyNumber: IFormattedNumber;
begin
 If o.GetInterface(IFormattedNumber, MyNumber) then
  ShowMessage(MyNumber.FormattedString);
end;
procedure DoSomethingWithInterface(I: IUnknown);
var
 MyNumber: IFormattedNumber;
begin
  ShowMessage((I as IFormattedNumber).FormattedString);
end;
```

Nota

La función *GetInterface* y el operador as sólo trabajarán si la interfaz se declara con un GUID. Por esto es por lo que mencionábamos con anterioridad que siempre se debería crear un GUID para una interfaz, aún cuando esto no sea estrictamente requerido por una interfaz no-COM. Si no asociamos un GUID a una interfaz, el compilador devolverá un críptico error de compilación:

```
Operator not applicable to this operand type
```
(Operador no aplicable a este tipo de operando)

La función GetInterface generará un mensaje de error de compilación más descriptivo:

```
Interface 'IFormattedNumber' has no interface identification
```
(La interfaz 'IFormattedNumber' no tiene identificación de interfaz)

No combine el Modelo de Objeto y el Modelo de Interfaz

En los ejemplos presentados hasta el momento hemos mostrado las referencias a objetos y las referencias a interfaces como si trabajaran juntas y fueran intercambiables. No es este el caso. El modelo de interfaz es una alternativa al modelo de referencia. Cuando implementamos una o más interfaces en una clase, deberíamos manipular los objetos de la clase usando únicamente interfaces. En esta sección veremos los problemas que puede ocasionar la combinación de ambos modelos, y un "truco sucio" que Ud. podrá utilizar si se encuentra en una situación en la que sea absolutamente necesario la combinación de los mismos.

Nota

Si usted se encuentra en una situación que requiera mezclar el modelo de objeto y el modelo de interfaz, lo más probable es que haya un error de diseño en su aplicación.

El operador **as** tiene un efecto colateral que debemos tener presente. El uso del operador **as** hace que Delphi llame a la función *_AddRef*. Por supuesto, al terminar con la interfaz, Delphi llamará a *_Release* y el objeto será destruido (asumiendo que el contador de referencias sea cero). A continuación se muestra un ejemplo que combina el modelo de objeto con el modelo de interfaz, y sufre las consecuencias:

```
procedure DoSomethingWithInterface(Intf: IFormattedNumber);
begin
 ShowMessage(Intf.FormattedString);
end;
procedure CreateAndUseObject;
var
  MyInteger: TFormattedInteger;
begin
  MyInteger := TFormattedInteger.Create(12);
  DoSomethingWithInterface(MyInteger as IFormattedNumber);
  MyInteger.SetValue(10);
end;
```

En este ejemplo, *MyInteger* es un objeto de la clase *TFormattedInteger*. Es creado usando el modelo de objeto (en otras palabras, el resultado de *TFormattedInteger.Create* se asigna a una variable objeto). En la llamada a *DoSomethingWithInterface* se obtiene una referencia a *IFormattedNumber* (aquí es

donde el código cambia al modelo de interfaz). Delphi llama automáticamente a *_AddRef* justo antes de la llamada a la función. En este momento se incrementa el contador de referencias, que toma el valor 1. Cuando finaliza la ejecución de la función, Delphi llama a *_Release* y se decrementa el contador de referencias, que toma el valor 0, destruyéndose el objeto en el proceso.

El resultado final es que después de la llamada a *DoSomethingWithInterface* el objeto *MyInteger* ya no es válido. Cualquier referencia posterior a *MyInteger* causará con seguridad una violación de acceso.

Observe que si *DoSomethingWithInterface* se declara como:

```
procedure DoSomethingWithInterface(var Intf: IFormattedNumber);
```

ó

```
procedure DoSomethingWithInterface(const Intf: IFormattedNumber);
```

entonces la interfaz no será liberada, y el objeto no se destruirá automáticamente. La razón es muy simple: un parámetro **const** o **var** hace que Delphi pase la interfaz por referencia en lugar de por valor. El paso de la interfaz por valor fuerza la llamada a *_AddRef* y *_Release*.

La forma correcta de escribir el código anterior (siguiendo el modelo de interfaz) es la siguiente:

```
procedure DoSomethingWithInterface(Intf: IFormattedNumber);
begin
 ShowMessage(Intf.FormattedString);
end;
procedure CreateAndUseInterface;
var
  MyInteger: IFormattedInteger;
begin
  MyInteger := TFormattedInteger.Create(12);
  DoSomethingWithInterface(MyInteger);
  MyInteger.SetValue(10);
end;
```

Este código sólo usa el modelo de interfaz (no hay variables de tipo *TFormattedInteger*). Como resultado, el contador de referencias de *MyInteger* es 1 cuando el objeto es creado. Durante la llamada *DoSomethingWithInterface*, el contador de referencias se incrementa momentáneamente a 2, y después se decrementa a 1, dejando intacta la variable *MyInteger*.

Evitando el conteo automático de referencias

Ya hemos visto lo que puede ocurrir al mezclar los modelos de referencia en una aplicación. Ahora veremos dos formas de evitar el conteo automático de referencias cuando sea necesario combinar ambos modelos. Echemos un vistazo al siguiente código:

```
procedure DoSomethingWithInterface(Intf: IFormattedNumber);
begin
 ShowMessage(Intf.FormattedString);
end;
procedure CreateAndUseObject;
var
 MyInteger: TFormattedInteger;
begin
  MyInteger := TFormattedInteger.Create(12);
  MyInteger._AddRef;
  DoSomethingWithInterface(MyInteger as IFormattedNumber);
  MyInteger.SetValue(10);
  MyInteger._Release;
end;
```

Esta solución funciona, pero tenemos que acordarnos de llamar a *_AddRef* y *_Release* para cada objeto cuando dejemos de usarlo. Otra solución es atacar el problema desde la raíz implementando una clase que no se destruya a sí misma cuando el contador de referencias llegue a cero.

Delphi no proporciona ese tipo de clase, pero es fácil crearla. A continuación implementaremos una clase que no es destruida automáticamente por el conteo de referencias.

```
type
  TNonRefCountedObject = class(TInterfacedObject, IUnknown);
  protected
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  end;
function TNonRefCountedObject._AddRef: Integer;
begin
  Result := -1;
end;
function TNonRefCountedObject._Release: Integer;
begin
  Result := -1;
end;
```

Veamos qué está ocurriendo aquí. Simplemente derivamos *TNonRefCountedObject* de *TInterfacedObject*, y reimplementamos los métodos necesarios para evitar el conteo de referencias.

Nota

Aunque *_AddRef* y *_Release* no están declarados como virtual en *TInterfacedObject*, podemos redefinir la implementación de los mismos, simplemente redeclarando la interfaz y los métodos en la clase descendiente. Si hubieran sido declarados virtual, aún podríamos redefinirlos usando la directiva override al final de la definición en *TNonRefCountedClass*, pero el compilador generaría más código para cada método.

Como puede verse, los métodos *_AddRef* y *_Release* ahora devuelven –1 en lugar de implementar un conteo de referencias. Este código nunca intentará destruir el objeto. Por tanto, asumiendo que *TFormattedInteger* desciende de *TNonRefCountedObject*, podemos usarlo así:

```
procedure DoSomethingWithInterface(Intf: IFormattedNumber);
begin
  ShowMessage(Intf.FormattedString);
end;
procedure CreateAndUseObject;
var
  MyInteger: TFormattedInteger;
begin
  MyInteger := TFormattedInteger.Create(12);
  DoSomethingWithInterface(MyInteger as IFormattedNumber);
  MyInteger.SetValue(10);
  MyInteger.Free;
end;
```

Tengamos presente que ambas "soluciones" no son más que trucos sucios. La solución *real* es evitar totalmente la mezcla del modelo de objeto y el modelo de interfaz.

Esto significa que no debemos añadir arbitrariamente soporte de interfaz a una clase que ya haya sido escrita. Si vamos a usar interfaces, la clase debe ser diseñada para ello desde el principio, y las interfaces deberían ser el único método de acceso usado por la clase.

Ejemplo: IntfDemo

El siguiente ejemplo ilustra la mayoría de los conceptos discutidos hasta ahora. La Figura 1.2 muestra el formulario principal de *IntfDemo*.

Figura 1.2: Aplicación *IntfDemo*.

Este ejemplo usa la interfaz *IFormattedNumber*, ya familiar para nosotros, y crea dos clases que la implementan: *TFormattedInteger* y *TFormattedHexInteger*. Como *TFormattedHexInteger* desciende de *TFormattedInteger*, no tiene que hacer referencia explícita a *IFormattedNumber*; *TFormattedInteger* ya ha se ha ocupado de ello.

Los destructores de *TFormattedInteger* y *TFormattedHexInteger* han sido redefinidos, de forma que podamos ver exactamente cuándo son llamados.

El formulario principal tiene cinco botones y un memo. El primer botón crea un objeto, llama a un método de ese objeto, y después lo destruye - nada extraordinario.

El segundo botón también crea un objeto, pero lo asigna a una variable de tipo interfaz. La interfaz cae fuera de ámbito al final del procedimiento y el objeto es automáticamente destruido - no hace falta liberar explícitamente el objeto.

El tercer botón hace lo mismo que el segundo, excepto que demuestra cómo asignar **nil** a una variable de tipo interfaz, con el objetivo de forzar su liberación. Como consecuencia, el objeto es destruido inmediatamente.

El cuarto botón crea un objeto, y seguidamente llama a una función que, implícitamente, obtiene una interfaz a partir del objeto. Cuando finaliza la ejecución de la función, el objeto se destruye automáticamente. Es importante comprender exactamente qué está ocurriendo aquí, porque esto puede dar lugar a problemas de memoria si no somos cuidadosos.

El último botón hace lo mismo que el cuarto, excepto que el parámetro de la función a la que llama se define como **const**. Esto evita las llamadas automáticas a *_AddRef* y *_Release*, y el objeto no será destruido automáticamente.

En el fichero *IntfUnit.pas* se encuentra la definición de la interfaz *IFormattedNumber*, y las dos clases que la implementan (ver Listado 1.4).

Listado 1.4: Código fuente de IntfDemo - *IntfUnit.pas*

```
unit IntfUnit;
interface
uses
  Classes, SysUtils, Dialogs;
type
  IFormattedNumber = interface
    ['{2DE825C1-EADF-11D2-B39F-0040F67455FE}']
    function FormattedString: string;
    function GetName: string;
  end;
  TFormattedInteger = class(TInterfacedObject, IFormattedNumber)
  private
```

```
    FValue: Integer;
  public
    constructor Create(AValue: Integer);
    destructor Destroy; override;
    function FormattedString: string;
    function GetName: string;
  end;
  TFormattedHexInteger = class(TFormattedInteger, IFormattedNumber)
  public
    destructor Destroy; override;
    function FormattedString: string;
    function GetName: string;
  end;
implementation
uses
 MainForm;
{ TFormattedInteger }
constructor TFormattedInteger.Create(AValue: Integer);
begin
  inherited Create;
  FValue := AValue;
end;
destructor TFormattedInteger.Destroy;
begin
  Form1.Memo1.Lines.Add( 'TFormattedInteger.Destroy');
  inherited Destroy;
end;
function TFormattedInteger.FormattedString: string;
begin
  Result := 'The integer is ' + IntToStr(FVa1ue);
end;
function TFormattedInteger.GetName: string;
begin
  Result := 'TFormattedInteger.GetName';
end;
{ TFormattedHexInteger }
destructor TFormattedHexInteger.Destroy;
begin
  Form1.Memo1.Lines.Add('TFormattedHexInteger.Destroy');
  inherited Destroy;
end;
function TFormattedHexInteger.FormattedString: string;
begin
  Result := 'The hex integer is $' + IntToHex(FValue, 4);
end;
function TFormattedHexInteger.GetName: string;
begin
  Result := 'TFormattedHexInteger.GetName';
end;
end.
```

El Listado 1.5 muestra el código fuente para el formulario principal de *IntfDemo*.

Listado 1.5: Código fuente de IntfDemo - *MainForm.pas*

```
unit MainForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, IntfUnit;
type
  TForm1 = class(TForm)
    btnAsObject: TButton;
    Memo1: TMemo;
    btnAsInterface: TButton;
    btnExplicitDestroy: TButton;
    btnFunction: TButton;
    btnConstFunction: TButton;
    procedure btnAsObjectClick(Sender: TObject);
    procedure btnAsInterfaceClick(Sender: TObject);
    procedure btnExplicitDestroyClick(Sender: TObject);
    procedure btnFunctionClick(Sender: TObject);
    procedure btnConstFunctionClick(Sender: TObject);
  private
    { Private declarations }
    procedure ShowName(Intf: IFormattedNumber);
    procedure ShowNameConst(const Intf: IFormattedNumber);
  public
    { Public dec1arations }
  end;
var
  Form1 : TForm1;
implementation
{$R *.DFM}
procedure TForm1.btnAsObjectClick(Sender: TObject);
var
  MyInt: TFormattedInteger;
begin
  MyInt := TFormattedInteger.Create(12);
  Memo1.Lines.Add(MyInt.FormattedString);
  MyInt.Free;   // Imprescindible destruir MyInt
  Memo1.Lines.Add( 'End of procedure');
  Memo1.Lines.Add('');
end;
procedure TForm1.btnAsInterfaceClick(Sender: TObject);
var
  MyIntf: IFormattedNumber;
begin
  MyIntf := TFormattedHexInteger.Create(1537);
  Memo1.Lines.Add(MyIntf.FormattedString);
  Memo1.Lines.Add('End of procedure');
  Memo1.Lines.Add('');
  // MyIntf es destruido autom□ticamente aqu□
```

```
end;
procedure TForm1.btnExplicitDestroyClick(Sender: TObject);
var
  MyIntf: IFormattedNumber;
begin
  MyIntf := TFormattedInteger.Create(3);
  Memo1.Lines.Add(MyIntf.FormattedString);
  MyIntf := nil;  // Destrucci□n expl□cita del objeto
  Memo1.Lines.Add('End of procedure');
  Memo1.Lines.Add('');
end;
procedure TForm1.ShowName(Intf: IFormattedNumber);
begin
  Memo1.Lines.Add(Intf.GetName);
end;
procedure TForm1.btnFunctionClick(Sender: TObject);
var
  MyInt: TFormattedInteger;
  MyHexIntf: IFormattedNumber;
begin
  MyInt := TFormattedInteger.Create(27);
  ShowName(MyInt);
  // □MyInt es destruido autom□ticamente aqu□!
  // MyInt.Free;  <- esto provocar□a violaci□n de acceso
  MyHexlntf :=  TFormattedHexInteger.Create(2047);
  ShowName(MyHexIntf);
  Memo1.Lines.Add('End of procedure');
  Memo1.Lines.Add('');
end;
procedure TForm1.ShowNameConst(const Intf: IFormattedNumber);
begin
  Memo1.Lines.Add(Intf.GetName);
end;
procedure TForm1.btnconstFunctionClick(Sender: TObject);
var
  MyInt: TFormattedInteger;
begin
  MyInt := TFormattedInteger.Create(27);
  ShowNameConst(MyInt);
  MyInt.Free;
  Memo1.Lines.Add('End of procedure');
  Memo1.Lines.Add('');
end;
end.
```

El Listado 1.6 presenta el código fuente del fichero de proyecto *IntfDemo*.

Listado 1.6: Código fuente de IntfDemo - *IntfDemo.dpr*

```
program IntfDemo;
uses
```

```
  Forms,
  MainForm in 'MainForm.pas' {Form1},
  IntfUnit in IntfUnit.pas'
{$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Este ejemplo muestra varias formas de acceder a las clases *TFormattedInteger* y *TFormattedHexInteger* tanto a través de objetos como de interfaces. Como los objetos e interfaces son creados, usados y destruidos dentro del mismo manejador de eventos, en realidad no estamos mezclando los modelos de referencia en este código. En vez de eso, tratamos de mostrar cómo difiere la gestión del tiempo de vida para objetos e interfaces.

En esta sección hemos aprendido cómo definir e implementar interfaces. Hemos insistido en que, si vamos a usar interfaces, deberíamos utilizarlas como único mecanismo para acceder a nuestros objetos. También hemos visto una o dos artimañas para forzar la coexistencia de los modelos de objeto e interfaz, cuando no encontremos otra solución.

En la próxima sección les presentaremos algunos ejemplos concretos que hacen uso de las interfaces.

Programando con interfaces

Hasta el momento, no hemos visto ejemplos aplicables a la vida real del uso de las interfaces en nuestro código Delphi. Detengámonos en un ejemplo que hace un buen uso de las interfaces a través de un concepto de programación que nos es familiar a todos: la ordenación.

Ejemplo: SortDemo - Algoritmos con interfaces

Muchos programas hacen uso de la ordenación de una u otra forma. Los programas menos sofisticados pueden utilizar técnicas simples como el algoritmo de burbuja, mientras que otros programas podrían requerir el uso de algoritmos mucho más complejos como *Quicksort* o *Mergesort*. Usualmente, cuando implementamos una rutina de ordenación esta queda estrechamente unida a los datos que deseamos ordenar. ¿Cuántas veces ha escrito código como el siguiente?

```
procedure SortEmployees;
// ordenar un array de empleados por orden alfab□tico
var
  I, J: Integer;
```

```
begin
  for I := 1 to MAX_EMPLOYEES 1 do
    for J := I + 1 to MAX_EMPLOYEES do
      if EmployeeArray[J].Name < EmployeeArray[I .Name then begin
        Temp := EmployeeArray[I ;
        EmployeeArray[I] := EmployeeArray[J ;
        EmployeeArray[J] := Temp;
      end;
end;
```

Lo único que hace el procedimiento anterior es ordenar un *array* de registros de empleados. Probablemente justifiquemos la implementación de la ordenación de esta forma con el hecho de que se trata de una ordenación rápida por el método de la burbuja y podemos escribir el código tan rápidamente que no tenemos ni que pensarlo.

¿No sería mejor tener a nuestra disposición una rutina que pudiera ser usada no sólo para ordenar, sino también para organizar elementos en una lista ordenada, o donde quiera que necesitemos una función de comparación? Usando interfaces podemos hacer realidad esto una manera sencilla.

Definamos una interfaz llamada *ICompare*. La función de *ICompare* será comparar un objeto (*ObjectA*) con otro similar (*ObjectB*), y devolver el valor *–1* si *ObjectA* es menor (anterior) que *ObjectB*, cero (0) si los dos objetos son iguales, o *+1* si *ObjectA* es mayor que *ObjectB*. Para hacer más útil este ejemplo, vamos a añadir a la interfaz la capacidad de soportar múltiples métodos de ordenación para un objeto. He aquí la declaración de la interfaz:

```
type
  ICompare = interface
    ['{DDFE0840-E8FB-11D2-9085-0040F6741DE2}']
    function CompareWith(ACompareTo: ICompare; AOrderBy: Integer):
Integer;
  end;
```

El único objetivo de la interfaz es proporcionar a un objeto el mecanismo para que se ordene a sí mismo de alguna manera. Esto ilustra otro punto importante sobre interfaces: cuando creemos una interfaz, hagámoslo lo más específica posible. No debemos intentar crear una interfaz que lo haga todo. Una interfaz multipropósito no será útil en muchas situaciones. Lo más probable es que resulte específica para la aplicación que estamos escribiendo. En vez de utilizar una interfaz de propósito general, cree interfaces más pequeñas, más concretas, que puedan reutilizarse en las más diversas situaciones.

El uso de la interfaz será de esta forma:

```
if MyObject1.CompareWith(MyObject2, SortIndex) < 0 then
  // MyObject1 es menor que MyObject2
```

La intención del parámetro *AOrderBy* es permitir al que implemente la interfaz la definición de distintos métodos de ordenación. Por ejemplo, la clase *TStudent* en un programa que gestiona las matrículas de un colegio podría necesitar ordenar a los estudiantes por nombre o por el número de seguridad social. La clase *TStudent* puede definir que el valor 1 significa "ordenar por nombre" y el valor 2 "ordenar por número de seguridad social".

El Listado 1.7 nos muestra el código fuente de la unidad *IntfUnit*, que declara la interfaz *ICompare* e implementa un algoritmo de ordenación de tipo "burbuja".

Listado 1.7: Código fuente de SortDemo - *IntfUnit.pas*

```
unit IntfUnit;
interface
type
  ICompare = interface
    ['{DDFE0840-E8FB-11D2-9085-0040F6741DE2}']
    function CompareWith(ACompare: ICompare; ASortBy: Integer): Integer;
  end;
procedure SortArray(var A: Array of IUnknown; ASortBy: Integer);
implementation
procedure SortArray(var A: Array of IUnknown; ASortBy: Integer);
var
  I, J: Integer;
  Temp: IUnknown;
begin
  for i := Low(A) to High(A) - 1 do
    for J := I + 1 to High(A) do
      if (A[J] as ICompare).CompareWith(A[I] as ICompare, ASortBy) < 0
then
      begin
        Temp := A[I];
        A[I] := A[J];
        A[J] := Temp;
      end;
end;
end.
```

Observe que el procedimiento *SortArray* recibe un *array* de *IUnknown*. Por ejemplo, si pasáramos un *array* de *TInterfacedObject*, la siguiente línea de código causaría que los objetos sean destruidos prematuramente:

```
if (A[J] as ICompare).CompareWith(A[I], AOrderBy) < 0 then // 
```

El operador **as** haría que Delphi llamase a *_AddRef* y *_Release* sobre la interfaz *ICompare*, lo cual destruiría todos objetos que estamos ordenando ¡en pleno proceso de ordenación! Este es un ejemplo clásico de los posibles problemas que se originan cuando se mezclan los modelos de referencia.

El ejemplo *SortDemo* define dos clases diferentes que implementan la interfaz *ICompare*: *TEmployee* y *TInventoryItem*. Estas clases se definen en las unidades *EmpUnit.pas* e *InvUnit.pas*, respectivamente. El Listado 1.8 muestra el código fuente de la unidad *EmpUnit*.

Listado 1.8: Código fuente de SortDemo - *EmpUnit.pas*

```
unit EmpUnit;
interface
uses
  SysUtils, IntfUnit;
type
  TEmployeeOrder = (eoName, eoSalary);
  IEmployee = interface
    ['{FFCD24F0-4FE8-11D3-B84D-00400F67455FE}']
    function GetName: string;
    function GetSalary: Double;
  end;
  TEmployee = class(TInterfacedObject, IEmployee, ICompare)
  private
    FName: string;
    FSalary: Double;
  public
    constructor Create(AName: string; ASalary: Double);
    function CompareWith(ACompare: ICompare; aSortBy: Integer): Integer;
    function GetName: string;
    function GetSalary: Double;
  end;
implementation
constructor TEmployee.Create(AName: string; ASalary: Double);
begin
  inherited Create;
  FName := AName;
  FSalary := ASalary;
end;
function TEmployee.CompareWith(ACompare: ICompare; aSortBy: Integer):
Integer;
var
  Emp: IEmployee;
begin
  Result := 0;
  Emp := ACompare as IEmployee;
  case ASortBy of
  Ord(eoName):
    Result := CompareStr(FName, Emp.GetName);
  Ord(eoSalary):
    if FSalary < Emp.GetSalary then
      Result := -1
    else if FSalary > Emp.GetSalary then
      Result := 1
    else
```

```
      Result := 0;
  end;
end;
function TEmployee.GetName: string;
begin
  Result := FName;
end;
function TEmployee.GetSalary: Double;
begin
  Result := FSalary;
end;
end.
```

La clase *TEmployee* es una clase simple, que se utiliza para gestionar nombres de empleados y salarios. Los empleados pueden ser ordenados por nombre o por salario. Hemos implementado el tipo enumerativo *TEmployeeOrder* para especificar los métodos de ordenación válidos para la clase *TEmployee*.

El Listado 1.9 muestra el código fuente de la unidad *InvUnit*.

Listado 1.9: Código fuente de SortDemo - *InvUnit.pas*

```
unit InvUnit;
interface
uses
 SysUtils, IntfUnit;
type
  TInventoryItemOrder= (iioPartNumber, iioDescription, iioUnitPrice,
    iioInStock);
  IInventoryItem = interface
    ['{FFCD24F1-4FE8-11D3-B84D-0040F67455FE}']
    function GetPartNumber: string;
    function GetDescription: string;
    function GetInStock: Integer;
    function GetUnitPrice: Double;
  end;
  TInventoryItem = class(TInterfacedObject, IInventoryItem, ICompare)
  private
    FPartNumber: string;
    FDescription: string;
    FInStock: Integer;
    FUnitPrice: Double;
  public
    constructor Create(APartNumber: string; ADescription: string;
                       AInStock: Integer; AUnitPrice: Double);
    function CompareWith(ACompare: ICompare; ASortBy: Integer): Integer;
    function GetPartNumber: string;
    function GetDescription: string;
    function GetInStock: Integer;
    function GetUnitPrice: Double;
  end;
```

```
implementation
constructor TInventoryItem.Create(APartNumber: string; ADescription:
string;
              AInStock: Integer; AUnitPrice: Double);
begin
  inherited Create;
  FPartNumber := APartNumber;
  FDescription := ADescription;
  FInStock := AInStock;
  FUnitPrice := AUnitPrice;
end;
function TInventoryItem.CompareWith(ACompare: ICompare;
           ASortBy: Integer): Integer;
var
  Inv: IInventoryItem;
begin
  Result := 0;
  Inv := ACompare as IInventoryItem;
  case ASortBy of
  Ord(iioPartNumber):
    Result := CompareStr(FPartNumber, Inv.GetPartNumber);
  Ord(iioDescription):
    Result := CompareStr(FDescription, Inv.GetDescription);
  Ord(iioInStock):
    Result := FInStock - Inv.GetInStock;
  Ord(iioUnitPrice):
    if FUnitPrice < Inv.GetUnitPrice then
      Result := -1
    else if FUnitPrice > Inv.GetUnitPrice then
      Result := 1
    else
      Result := 0
  end;
end;
function TInventoryItem.GetDescription: string;
begin
  Result := FDescription;
end;
function TInventoryItem.GetInStock: Integer;
begin
  Result := FInStock;
end;
function TInventoryItem.GetPartNumber: string;
begin
  Result := FPartNumber;
end;
function TInventoryItem.GetUnitPrice: Double;
begin
  Result := FUnitPrice;
end;
end.
```

La clase *TInventoryItem* es similar en naturaleza a *TEmployee*, pero se utiliza para gestionar productos de inventario, que pueden ser ordenados por número de pieza, descripción, precio unitario o cantidad en existencia (*iioPartNumber*, *iioDescription*, *iioUnitPrice*, o *iioInStock*).

La razón para crear estos dos clases es mostrar cómo podemos usar una misma interfaz (*ICompare*) para ordenar conjuntos de datos completamente independientes.

El Listado 1.10 muestra el código del formulario principal de *SortDemo*.

Listado 1.10: Código fuente de SortDemo - *MainForm.pas*

```
unit MainForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  ComCtrls, IntfUnit, EmpUnit, InvUnit, StdCtrls;
const
  MAX_EMPLOYEES = 5;
  MAX_INVENTORYITEMS = 3;
type
  TfrmMain = class(TForm)
    PageControl1: TPageControl;
    tabBubbleSort: TTabSheet;
    tabListView: TTabSheet;
    btnByName: TButton;
    btnBySalary: TButton;
    lbEmployees: TListBox;
    Label1: TLabel;
    lvInventory: TListView;
    Label2: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure btnByNameClick(Sender: TObject);
    procedure btnBySalaryClick(Sender: TObject);
    procedure lvInventoryColumnClick(Sender: TObject; Column:
TListColumn);
    procedure lvInventoryCompare(Sender: TObject;
      Item1, Item2: TListItem; Data: Integer; var Compare: Integer);
  private
    { Private declarations }
    FEmpArray: array [1.. MAX_EMPLOYEES] of IUnknown;
    FInvArray: array [1.. MAX_INVENTORYITEMS] of IUnknown;
    procedure LoadListBox;
    procedure CreateInventoryItem(AIndex: Integer; APartNo,
         ADescription: string; AInStock: Integer; AUnitPrice: Double);
  public
    { Public declarations }
  end;
var
```

```
  frmMain: TfrmMain;
implementation
{$R *.DFM}
procedure TfrmMain.CreateInventoryItem(AIndex: Integer; APartNo,
            ADescription: string; AInStock: Integer; AUnitPrice: Double);
var
  ListItem: TListItem;
begin
  FInvArray[AIndex] := TInventoryItem.Create(APartNo, ADescription,
    AInStock, AUnitPrice);
  ListItem := IvInventory.Items.Add;
  ListItem.Caption := APartNo;
  ListItem.SubItems.Add(ADescription);
  ListItem.SubItems.Add(FloatToStrF(AUnitPrice, ffCurrency, 5, 2));
  ListItem.SubItems.Add(IntToStr(AInStock));
  ListItem.Data := Pointer(FInvArray[AIndex]);
end;
procedure TfrmMain.FormCreate(Sender: TObject);
begin
  // Crear algunos empleados
  FEmpArray[1] := TEmployee.Create('Smith, Tom', 19200.00);
  FEmpArray[2] := TEmployee.Create('Doe, John', 38000.00);
  FEmpArray[3] := TEmployee.Create('Williams, Fred', 26500.00);
  FEmpArray[4] := TEmployee.Create('Jones, Bob', 90000.00);
  FEmpArray[5] := TEmployee.Create('Adams, Tim', 42500.00);
  // Cargar empleados en cuadro de lista
  LoadListBox;
  // Crear algunos productos de inventario
  CreateInventoryItem(1, 'P5409', 'Widget', 35, 1.19);
  CreateInventoryItem(2, 'X1234', 'Gadget', 4, 14.95);
  CreateInventoryItem(3, 'J7749', 'Doodad', 17, 8.79);
end;
procedure TfrmMain.LoadListBox;
var
  Index: Integer;
  Emp: IEmployee;
begin
  lbEmployees.Items.BeginUpdate;
  try
    lbEmployees.Items.Clear;
    for Index := 1 to MAX_EMPLOYEES do begin
      Emp := FEmpArray[Index] as IEmployee;
      lbEmployees.Items.Add(Emp.GetName + #9 +
        FloatToStrF(Emp.GetSalary, ffCurrency, 8, 2));
    end;
  finally
    lbEmployees.Items.EndUpdate;
  end;
end;
procedure TfrmMain.btnByNameClick(Sender: TObject);
begin
  SortArray(FEmpArray, Ord(eoName));
```

```
  LoadListBox;
end;
procedure TfrmMain.btnBySalaryClick(Sender: TObject);
begin
  SortArray(FEmpArray, Ord(eoSalary));
  LoadListBox;
end;
procedure TfrmMain.IvInventoryColumnClick(Sender: TObject;
  Column: TListColumn);
begin
  IvInventory.CustomSort(nil, Column.Index);
end;
procedure TfrmMain.IvInventoryCompare(Sender: TObject;
      Item1, Item2: TListItem; Data: Integer; var Compare: Integer);
var
  I1, I2: IUnknown;
begin
  I1 := IUnknown(Item1.Data);
  I2 := IUnknown(Item2.Data);
  Compare := (I1 as ICompare).CompareWith((I2 as ICompare), Data);
end;
end.
```

El método *FormCreate* de la clase *TfrmMain* crea un *array* de empleados y un *array* de productos de inventario. Por supuesto, una aplicación real tomaría estos datos de una base de datos o de ficheros de algún tipo, pero esto será suficiente para la demostración.

Cuando ejecute la aplicación, verá un *TPageControl* con dos páginas sobre él. La primera página contiene un *TListBox* que es usado para mostrar la lista de empleados.

La Figura 1.3 presenta la primera página de la aplicación *SortDemo*.

Figura 1.3: Aplicación *SortDemo*: lista de empleados.

Por defecto, los nombres de los empleados son mostrados en el orden que han sido creados. Pulsando el botón '*Sort by Name*' o '*Sort by Salary*' se ordena el *array* de

empleados por nombre o por salario, respectivamente. La rutina de ordenación utilizada se apoya en un simple algoritmo de burbuja que ordena los objetos en base al método *CompareWith* de la interfaz *ICompare*. Pulsando sobre la pestaña '*List View*' veremos el inventario de productos en el orden en que fueron creados. Si se pulsa sobre cualquiera de los cuatro encabezamientos de columna '*Part #*', '*Description*', '*Unit Price*', o '*In Stock*', la lista es ordenada según la columna seleccionada. Este código utiliza la interfaz *ICompare* de una forma completamente diferente.

La Figura 1.4 presenta la segunda página de la aplicación *SortDemo*.

Figura 1.4: Aplicación *SortDemo*: listado de productos de inventario.

Si no está familiarizado con las capacidades de ordenación de *TListView*, échele un vistazo al siguiente fragmento de código, extraído del Listado 1.10:

```
procedure TFrmMain.IvInventoryColumnClick(Sender: TObject;
  Column: TListColumn);
begin
  IvInventory.CustomSort(nil, Column.Index);
end;
procedure TfrmMain.IvInventoryCompare(Sender: TObject);
      Item1, Item2: TListItem; Data: Integer; var Compare: Integer);
var
  I1, I2: IUnknown;
begin
  I1 := IUnknown(Item1.Data);
  I2 := IUnknown(Item2.Data);
  Compare := (I1 as ICompare).CompareWith((I2 as ICompare), Data);
end;
```

El método *CustomSort* de *TListView* utiliza el método asignado a su evento *OnCompare* para comparar dos productos cualesquiera de la lista. Debido a que se ha usado la propiedad *Data* de la lista de productos para colocar un puntero a la interfaz *IUnknown* de la clase *TInventoryItem*, podemos simplemente interpretar la propiedad *Data* como una *IUnknown*, y llamar al método *TInventoryItem.CompareWith* para comparar productos.

Obviamente, *TListView* tiene su propio motor interno de ordenación, el cual llama al método *OnCompare* en los momentos apropiados. Lo que es importante comprender aquí es que al declarar la interfaz *ICompare* genéricamente, ésta puede ser usada sobre múltiples instancias. Si en lugar de declarar una interfaz *ICompare*, hubiéramos declarado una interfaz *ISort*, entonces el código no hubiera sido utilizable en este caso particular.

Hasta este punto, hemos cubierto los conceptos básicos relacionados con las interfaces. La próxima sección discute algunos aspectos adicionales de las interfaces que son ligeramente más avanzados.

Tópicos avanzados sobre interfaces

En esta sección examinaremos algunos temas avanzados sobre las interfaces en Delphi, incluyendo:

- Implementando múltiples interfaces en una clase.
- Delegación de interfaces.
- Propiedades en las interfaces.

En la última parte de esta sección crearemos una aplicación gráfica que nos mostrará cómo utilizar algunos de estos tópicos avanzados.

Implementando múltiples interfaces en una clase

Object Pascal no soporta la herencia múltiple, como es el caso de otros lenguajes tales como C++. La principal razón para evitar la herencia múltiple es que múltiples ancestros con nombres de métodos duplicados en el árbol de herencia conducen a una ambigüedad a la hora de determinar qué método asociar a la clase derivada.

En algunas situaciones aisladas la herencia múltiple puede ser provechosa o incluso necesaria. Afortunadamente, con las interfaces podemos obtener la mayoría de los beneficios de la herencia múltiple sin sus dolores de cabeza asociados. La implementación de múltiples interfaces en una clase es sencilla; simplemente liste las interfaces una tras otra en la declaración de la clase, por ejemplo:

```
type
  TMyMultiplyInterfacedClass = class(TInterfacedObject,
    IInterface1, IInterface2, IInterface3);
```

TMyMultiplyInterfacedClass debe implementar los métodos de las interfaces *IInterface1*, *IInterface2* e *IInterface3*.

Múltiples interfaces no es herencia múltiple

Aún cuando la declaración de clase mostrada con anterioridad pudiera sugerirnos herencia múltiple, no lo es. *TMyMultiplyInterfacedClass* tiene una y sólo una clase progenitora, que es *TInterfacedObject*. El hecho que la clase implemente todos los métodos de las interfaces especificadas no cambia este hecho.

Cláusulas de resolución de métodos

¿Que haríamos en el caso de que más de una interfaz definiese un método con el mismo nombre? Consideremos las siguientes interfaces:

```
type
  IInterface1 = interface
    procedure DoIt;
  end;
  IInterface2 = interface
    procedure DoIt;
  end;
```

La declaración de ambas interfaces es completamente legal. Ahora, ¿qué hacer si deseáramos crear una clase que implemente a las dos interfaces? Asumiendo que la implementación de *IInterface1.DoIt* fuera diferente de la de *IInterface2.DoIt*, nuestra clase requeriría dos métodos llamados *DoIt*, lo cual no es posible.

Nota

Si la implementación de *IInterface1.DoIt* e *IInterface2.DoIt* es la misma, podemos implementar un único método *DoIt* en nuestra clase. Delphi mapeará ambas interfaces al mismo método.

Para resolver este problema, Delphi nos brinda una característica conocida como *cláusula de resolución de métodos*. La resolución de métodos nos permite cambiar el nombre de un método de la interfaz cuando ésta es implementada en una clase. Consideremos la siguiente declaración de clase:

```
type
  TMyObject = class (TInterfacedObject, IInterface1, IInterface2)
    procedure IInterface1.DoIt: = DoIt1;
    procedure IInterface2.DoIt: = DoIt2;
    procedure DoIt1;
    procedure DoIt2;
  end;
```

TMyObject mapea el método *IInterface1.DoIt* al procedimiento *DoIt1*, y el método *IInterface2.DoIt* al procedimiento *DoIt2*. El siguiente fragmento de código en realidad llama a *TMyObject.DoIt1*:

```
var
  MyInt1: IInterface1;
begin
  MyInt1 := TMyObject.Create;
  MyInt1.DoIt;
end;
```

No necesitamos renombrar todos los nombres de métodos conflictivos. El siguiente código es perfectamente legal:

```
type
  TMyObject = class (TInterfacedObject, IInterface1, IInterface2)
    procedure IInterface2.DoIt = Dolt2;
    procedure DoIt;
    procedure Dolt2;
  end;
```

En este ejemplo, *IInterface1.DoIt* es mapeado al método *DoIt*, como sería usual, mientras que *IInterface2.DoIt* mapea a *DoIt2*.

Delegación

Supongamos que tenemos una clase bien diseñada, llamada *TObject1*, que implementa la interfaz *IInterface1*. Deseamos crear una clase llamada *TCombinedObject* que implemente las interfaces *IInterface1* e *IInterface2*. Aparentemente tendríamos que reimplementar los métodos de *IInterface1*, posiblemente copiando el código de *TObject1* a *TCombinedObject*, ¿correcto?

No necesariamente. Delphi nos permite *delegar* la implementación de una interfaz a otra clase. La delegación significa que una clase contiene un puntero a otra clase. La clase 'interna' implementa la funcionalidad de una o más interfaces. En lugar de reimplementar las interfaces en cuestión, la clase externa o contenedora simplemente pasa esos métodos a la clase contenida.

El siguiente código implementa la interfaz *IInterface1* en la clase *TObject1*. *TCombinedObject* contiene una referencia a *TObject1*, y *delega* la implementación de *IInterface1* a *FObj1*.

```
type
  IInterface1 = interface
    ['{72B1C43-AO21-1103-B8AC-OO4GF67455FE}']
    procedure DoIt1;
  end;
  IInterface2 = interface
    ['{72B1C431-AO2l -11D3-B8AC-OO4OF67455FE}']
    procedure DoIt2;
  end;
  TObjectl = class (TInterfacedObject, IInterface1)
  protected
    procedure DoItl;
  end;
  TCombinedObject = class (TInterfacedObject, IInterface1, IInterface2)
  protected
    FObj1: IInterfacel;
  public
    constructor Create;
    destructor Destroy; override;
```

```
    procedure DoIt2;
    property MyIntf: IInterfacel read FObj1 implements IInterface1;
  end;
implementation
{ TObject1 }
procedure TObjectl.DoItl;
begin
 ShowMessage('Inside TObject.DoIt1');
end;
{ TCombinedObject }
constructor TCombinedObject.Create;
begin
  inherited Create;
  FObj1 := TObject1.Create;
end;
destructor TCombinedObject.Destroy;
begin
  FObj1 := nil;
  inherited;
end;
procedure TCombinedObject.DoIt2;
begin
  ShowMessage('Inside TCombinedObject.DoIt2');
end;
```

Para usar *TCombinedObject* en su código, deberá hacerlo de la siguiente forma:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I1: IInterface1;
  I2: IInterface2;
begin
   I2 := TCombinedObject.Create;
   I2 := DoIt2;
   I1 := I2 as IInterface1;
   I1.DoItl;
end;
```

Observe que la sentencia *I2 as IInterface1* automáticamente extrae la interfaz *IInterface1* de *TCombinedObject*, aún cuando *IInterface1* está implementada por *TObject1*. En esto reside la belleza de la delegación: nuestro código no necesita ser consciente de cómo la interfaz está implementada realmente.

Nota

Los objetos añadidos requieren un especial cuidado.

Cuando cree un objeto que piense utilizar más adelante como objeto agregado, NO deberá heredar de *TInterfacedObject*. Si lo hace, entonces cuando obtenga una referencia a una interfaz implementada por el objeto agregado, NO podrá usar la interfaz en cuestión para obtener una referencia a una interfaz implementada por el objeto contenedor.

Por ejemplo, la siguiente aplicación no se ejecutará:

```
unit Unitl;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;
type
  IInterfacel = interface
    ['{72B1C43O-AO21-11D3-B8AC-OO4OF67455FE}']
    procedure DoIt1;
  end;
  IInterface2 = interface
    ['{72B1C431-AO21-11D3-B8AC-OO4OF67455FE}']
    procedure DoIt2;
  end;
  TAggObject = class (TInterfacedObject, IInterface1)
  protected
    procedure DoItl;
  end;
  TCombinedObject = class(TInterfacedObject, IInterfacel, IInterface2)
  protected
    FAggObj: IInterfacel;
  public
    constructor Create;
    destructor Destroy; override;

    procedure DoIt2;
    property AggObj: IInterfacel read FAggObj implements IInterfacel;
  end;
  TForm1 = class(TForm)
    Buttonl : TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
 Form1 : TForm1;
implementation
{$R *.DFM}
{ TAggObject }
procedure TAggObject.DoItl;
begin
  ShowMessage('Inside TAggObject.DoIt1');
end;
{ TCombinedObject }
constructor TCombinedObject.Create;
begin
 inherited Create;
  FAggObj := TAggObject.Create;
```

```
end;
destructor TCombinedObject.Destroy;
begin
  FAggObj := nil;
  inherited;
end;
procedure TCombinedObject.DoIt2;
begin
  ShowMessage('Inside TCombinedObject.DoIt2');
end;
procedure TForm1.Button1Click(Sender: TObject);
var
  I1: IInterface1;
  I2: IInterface2;
begin
  I2 := TCombinedObject.Create;
  I2.DoIt2;
  I1 := I2 as IInterface1; // Esto funciona
  I1.DoIt1;
  // Utilizar la interfaz delegada IInterface2 para obtener
  // una referencia a la interfaz IInterface2
  I2 := I1 as IInterface2; // Esto falla
  I2.DoIt2;
end;
end.
```

Delphi lanzará una excepción "*Interface not supported*" (Interfaz no soportada) cuando tratemos de interpretar *I1* como *IInterface2*. La razón es que una vez que el compilador tiene una referencia al objeto agregado, no tiene forma de conocer cómo volver al objeto contenedor.

Delphi brinda específicamente para esta situación la clase *TAggregatedObject*. La aplicación anterior funcionará correctamente si agregamos *ComObj.pas* a la cláusula uses, derivamos *TAggObject* de *TAggregatedObject* en lugar de *TInterfacedObject*, y modificamos el constructor *TCombinedObject.Create* por:

```
constructor TCombinedObject.Create;
begin
 inherited Create;
 FAggObj := TAggObject.Create(self);
end;
```

Observe que el constructor de *TAggregatedObject* requiere una referencia a un objeto contenedor ("*self*" en el fragmento de código anterior). Podríamos tener problemas si tratáramos de usar el mismo objeto como objeto independiente y también como un objeto agregado, dado que no podemos pasar el valor nil al constructor de *TAggregatedObject*.

Hallvard Vossbotn sugiere la siguiente clase como una alternativa a *TAggregatedObject*. Si desea usar la clase como clase independiente, llame al constructor *Create* sin argumentos. Si desea usar la clase como un objeto agregado, llame al constructor *Create* y pase una referencia al objeto contenedor (*self*).

```
type
  TAggregatedOrStandaloneObject = class(TInterfacedObject, IUnknown)
  private
    FController: Pointer;
    function GetController: IUnknown;
  protected
    { IUnknown }
    function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  public
    constructor Create(Contro1ler: IUnknown); overload;
    property Controller: IUnknown read GetController;
  end;
{ TAggregatedOrStandaloneObject }
constructor TAggregatedOrStandaloneObject.Create(Controller: IUnknown);
begin
  FController := Pointer(Controller)
end;
function TAggregatedOrStandaloneObject.GetController: IUnknown;
begin
  if Assigned(FContro11er) then
    Result := IUnknown(FController)
  else
    Result := Self;
end;

{ TAggregatedOrStandaloneObject.IUnknown }
function TAggregatedOrStandaloneObject.QueryInterface(const IID: TGUID;
  out Obj): HResult;
begin
  if Assigned(FController) then
    Result := IUnknown(FController).QueryInterface(IID, Obj)
  else
    Result := inherited QueryInterface(IID, Obj);
end;
function TAggregatedOrStandaloneObject._AddRef: Integer;
begin
  if Assigned(FController) then
    Result := IUnknown(FController)._AddRef
  else
    Result := inherited _AddRef;
 end;
function TAggregatedOrStanda1oneObject._Release: Integer;
begin
  if Assigned(FContro11er) then
    Result := IUnknown(FController)._Release
  else
   Result := inherited _Release;
End;
```

Propiedades en las interfaces

Esto podría no ser inmediatamente obvio, pero podemos definir también propiedades en una interfaz. Podemos definir propiedades de sólo lectura, sólo escritura o de lectura y escritura, pero todos los accesos tienen que realizarse a través de funciones de acceso, debido a que las interfaces no pueden definir almacenamiento.

Por ejemplo, la siguiente declaración de interfaz define una propiedad de sólo lectura, otra de sólo escritura y una tercera de lectura/escritura:

```
type
  IPropertyInterface = interface
    function GetAge: Integer;
    procedure SetShoeSize(value: Double);
    function GetHeight: Double;
    procedure SetHeight(value: Double);
    property Age: Integer read GetAge;
    property ShoeSize: Double write SetShoeSize;
    property Height: Double read GetHeight write SetHeight;
  end;
```

Cuando implemente esta interfaz en una clase, tenga en cuenta que los usuarios de la clase tendrán acceso a las funciones de acceso y a las propiedades. En otras palabras, un usuario podrá llamar a la función *GetAge* o acceder a la propiedad *Age*. Debido a que la mayoría de los lenguajes no soportan propiedades, y las interfaces son diseñadas para que sean portables entre lenguajes, las funciones *GetXxx* y *SetXxx* necesitan ser accesibles a los usuarios de las interfaces.

Ejemplo: Demostración gráfica

El ejemplo final de este capítulo ilustra alguna de las técnicas avanzadas discutidas en esta sección, así como los conceptos expuestos desde principio del mismo. Específicamente, este ejemplo demuestra:

- La herencia de una clase que implementa una interfaz
- La implementación de múltiples interfaces en una clase
- Las propiedades en una interfaz

Muchas industrias utilizan gráficos orientados a objetos en sus aplicaciones. Las aplicaciones tipo CAD pueden dibujar, redimensionar y mover formas gráficas en la pantalla. Las herramientas de generación de prototipos de software permiten presentar varias figuras en pantalla y representar gráficamente las relaciones entre ellas. La lista podría ampliarse con muchos otros ejemplos.

GrphDemo es una aplicación que de forma aleatoria coloca cruces azules y rectángulos negros sobre un formulario. Las cruces y rectángulos tienen en común que ambos tienen color y posición. Los rectángulos se diferencian en que ellos

también tienen un tamaño (las cruces en esta aplicación particular no pueden ser redimensionadas).

GrphDemo utiliza la clase *TInterfaceList* de Delphi para almacenar una lista de figuras. Delphi 3 no incluye *TInterfaceList*, por lo que este ejemplo no podrá ser compilado bajo esa versión.

El Listado 1.11 nos muestra el código para las interfaces definidas en *GrphDemo*. Como puede ver, cada atributo de los gráficos (color, tamaño, y posición) se define en una interfaz independiente. Los objetos gráficos en esta aplicación implementan las interfaces individuales que ellos soportan.

Listado 1.11: GrphDemo – código fuente de *IntfForm.pas*

```
unit IntfUnit;
interface
uses
  Windows, Graphics;
type
  IColor = interface
    ['{EAE054E1-EB94-11D2-9086-0040F6741DE2}']
    function GetColor: TColor;
    procedure SetColor(AColor: TColor);
    property Color: TColor read GetColor write SetColor;
  end;
  IPosition = interface
    ['{162EDFE1-EB8C-11D2-9086-0040F6741DE2}']
    function GetXY: TPoint;
    procedure SetXY(APoint: TPoint);
    procedure NudgeUp;
    procedure NudgeDown;
    procedure NudgeLeft;
    procedure NudgeRight;
    property Point: TPoint read GetXY write SetXY;
  end;
  ISize = interface
    ['{162EDFE0-EB8C-11D2-9086-0040F6741DE2}']
    function GetWidth: Integer;
    function GetHeight: Integer;
    procedure SetWidth(AWidth: Integer);
    procedure SetHeight(AHeight: Integer);
    property Height: Integer read GetHeight write SetHeight;
    property Width: Integer read GetWidth write SetWidth;
  end;
  IDraw = interface
    ['{FFCD24F3-4FE8-11D3-B84D-0040F67455FE}']
    procedure Draw(ACanvas: TCanvas);
  end;
implementation
end.
```

El Listado 1.12 muestra el código de los dos objetos gráficos que implementan las interfaces.

Listado 1.12: GrphDemo - *Shapes.pas*

```
unit Shapes;
interface
uses
  Windows, IntfUnit, Graphics, Dialogs;
type
  TMark = class(TInterfacedObject, IColor, IPosition, IDraw)
  private
    FColor: TColor;
    FPoint: TPoint;
  public
    constructor Create(APoint: TPoint; AColor: TColor);
    function GetColor: TColor;
    procedure SetColor(AColor: TColor);
    function GetXY: TPoint;
    procedure SetXY(APoint: TPoint);
    procedure NudgeUp;
    procedure NudgeDown;
    procedure NudgeLeft;
    procedure NudgeRight;
    procedure Draw(ACanvas: TCanvas);
  end;
  TSquare = class(TMark, IColor, IPosition, ISize, IDraw)
  private
    FHeight: Integer;
    FWidth: Integer;
  public
    constructor Create(AUpLeft: TPoint; AHeight: Integer;
      AWidth: Integer; AColor: TColor);
    function GetWidth: Integer;
    function GetHeight: Integer;
    procedure SetWidth(AWidth: Integer);
    procedure SetHeight(AHeight: Integer);
    procedure Draw(ACanvas: TCanvas);
  end;
implementation
{ TMark }
constructor TMark.Create(APoint: TPoint; AColor: TColor);
begin
  inherited Create;
  FPoint := APoint;
  FColor := AColor;
end;
procedure TMark.Draw(ACanvas: TCanvas);
begin
  ACanvas.Pen.Color := FColor;
  ACanvas.MoveTo(FPoint.X - 2, FPoint.Y - 2);
```

```
  ACanvas.LineTo(FPoint.X + 3, FPoint.Y + 3);
  ACanvas.MoveTo(FPoint.X - 2, FPoint.Y + 2);
  ACanvas.LineTo(FPoint.X + 3, FPoint.Y - 3);
end;
function TMark.GetColor: TColor;
begin
  Result := FColor;
end;
function TMark.GetXY: TPoint;
begin
  Result := FPoint;
end;
procedure TMark.NudgeDown;
begin
  Inc(FPoint.Y);
end;
procedure TMark.NudgeLeft;
begin
  Dec(FPoint.X);
end;
procedure TMark.NudgeRight;
begin
  Inc(FPoint.X);
end;
procedure TMark.NudgeUp;
begin
  Dec(FPoint.Y);
end;
procedure TMark.SetColor(AColor: TColor);
begin
  FColor := AColor;
end;
procedure TMark.SetXY(APoint: TPoint);
begin
  FPoint := APoint;
end;
{ TSquare }
constructor TSquare.Create(AUpLeft: TPoint; AHeight: Integer;
  AWidth: Integer; AColor: TColor);
begin
  inherited Create(AUpLeft, AColor);
  FHeight := AHeight;
  FWidth := AWidth;
end;
procedure TSquare.Draw(ACanvas: TCanvas);
begin
  ACanvas.Pen.Color := FColor;
  ACanvas.Rectangle(FPoint.X, FPoint.Y, FPoint.X + FWidth,
    FPoint.Y + FHeight);
end;
function TSquare.GetHeight: Integer;
```

```
begin
  Result := FHeight;
end;
function TSquare.GetWidth: Integer;
begin
  Result := FWidth;
end;
procedure TSquare.SetHeight(AHeight: Integer);
begin
  FHeight := AHeight;
end;
procedure TSquare.SetWidth(AWidth: Integer);
begin
  FWidth := AWidth;
end;
end.
```

El Listado 1.13 presenta el código fuente del formulario principal, mostrado en la Figura 1.5.

Figura 1.5: Formulario principal de *GrphDemo*.

Listado 1.13: Código fuente de GrphDemo – *MainForm.pas*

```
unit MainForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, ExtCtrls, StdCtrls, IntfUnit;
type
  TForm1 = class(TForm)
    btnNewSquare: TButton;
    btnNewMark: TButton;
    btnClear: TButton;
```

```
    btnNudgeLeft: TButton;
    btnNudgeRight: TButton;
    btnNudgeUp: TButton;
    btnNudgeDown: TButton;
    btnGrow: TButton;
    Bevel1: TBevel;
    PaintBox1: TPaintBox;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure btnNewSquareClick(Sender: TObject);
    procedure btnNewMarkClick(Sender: TObject);
    procedure btnClearClick(Sender: TObject);
    procedure btnNudgeLeftClick(Sender: TObject);
    procedure btnNudgeRightClick(Sender: TObject);
    procedure btnNudgeUpClick(Sender: TObject);
    procedure btnNudgeDownClick(Sender: TObject);
    procedure btnGrowClick(Sender: TObject);
    procedure PaintBox1Paint(Sender: TObject);
  private
    { Private declarations }
    FShapes: TInterfaceList;
    procedure Clear;
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
uses
  Shapes;
{$R *.DFM}
procedure TForm1.FormCreate(Sender: TObject);
begin
  FShapes := TInterfaceList.Create;
end;
procedure TForm1.FormDestroy(Sender: TObject);
begin
  Clear;
  FShapes.Free;
end;
procedure TForm1.btnNewSquareClick(Sender: TObject);
var
  Square: IUnknown;
begin
  Square := TSquare.Create(Point(10 + Random(200), 10 + Random(200)),
    40 + Random(20), 40 + Random(20), clBlack);
  FShapes.Add(Square);
  PaintBox1.Invalidate;
end;
procedure TForm1.btnNewMarkClick(Sender: TObject);
var
  Mark: IUnknown;
```

```
begin
  Mark := TMark.Create(Point(10 + Random(200), 10 + Random(200)),
    clBlue);
  FShapes.Add(Mark);
  PaintBox1.Invalidate;
end;
procedure TForm1.Clear;
begin
  // TInterfaceList.Clear libera autom□ticamente todos los objetos
  // asociados a las interfaces en la lista destruida
  FShapes.Clear;
end;
procedure TForm1.btnClearClick(Sender: TObject);
begin
  Clear;
  PaintBox1.Invalidate;
end;
procedure TForm1.PaintBox1Paint(Sender: TObject);
var
  Index: Integer;
  Shape: IUnknown;
begin
  for Index := 0 to FShapes.Count - 1 do begin
    Shape := FShapes[Index];
    (Shape as IDraw).Draw(PaintBox1.Canvas);
  end;
end;
procedure TForm1.btnNudgeLeftClick(Sender: TObject);
var
  Index: Integer;
  Shape: IUnknown;
begin
  for Index := 0 to FShapes.Count - 1 do begin
    Shape := IUnknown(FShapes[Index]);
    (Shape as IPosition).NudgeLeft;
  end;
  PaintBox1.Invalidate;
end;
procedure TForm1.btnNudgeRightClick(Sender: TObject);
var
  Index: Integer;
  Shape: IUnknown;
begin
  for Index := 0 to FShapes.Count - 1 do begin
    Shape := IUnknown(FShapes[Index]);
    (Shape as IPosition).NudgeRight;
  end;
  PaintBox1.Invalidate;
end;
procedure TForm1.btnNudgeUpClick(Sender: TObject);
var
  Index: Integer;
```

```
  Shape: IUnknown;
begin
  for Index := 0 to FShapes.Count - 1 do begin
    Shape := IUnknown(FShapes[Index]);
    (Shape as IPosition).NudgeUp;
  end;
  PaintBox1.Invalidate;
end;
procedure TForm1.btnNudgeDownClick(Sender: TObject);
var
  Index: Integer;
  Shape: IUnknown;
begin
  for Index := 0 to FShapes.Count - 1 do begin
    Shape := IUnknown(FShapes[Index]);
    (Shape as IPosition).NudgeDown;
  end;
  PaintBox1.Invalidate;
end;
procedure TForm1.btnGrowClick(Sender: TObject);
var
  Index: Integer;
  Shape: IUnknown;
  SizeIntf: ISize;
begin
  for Index := 0 to FShapes.Count - 1 do begin
    Shape := IUnknown(FShapes[Index]);
    if Shape.QueryInterface(ISize, SizeIntf) = S_OK then begin
      SizeIntf.Width := SizeIntf.Width + 1;
      SizeIntf.Height := SizeIntf.Height + 1;
    end;
  end;
  PaintBox1.Invalidate;
end;
end.
```

El Listado 1.14 muestra el código fuente del fichero de proyecto GrphDemo.

Listado 1.14: GrphDemo – *grphdemo.dpr*

```
Program GrphDemo;
uses
  Forms,
  MainForm in 'MainForm.pas' {Form1},
  IntfUnit in 'IntfUnit.pas',
  Shapes in 'Shapes.pas';
{$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Resumen

En este capítulo hemos examinado las interfaces en detalle. Hemos mostrado cómo podemos usar interfaces internamente en nuestras aplicaciones Delphi, y hemos sentado las bases para el resto de los capítulos de este libro. Por ahora debe conocer cómo:

- Declarar una interfaz
- Implementar una interfaz en una clase.
- Implementar la funcionalidad requerida de *IUnknown*.
- Evitar el conteo de referencias para desactivar la destrucción automática de los objetos.
- Obtener un puntero a una interfaz, dada otra interfaz.
- Implementar múltiples interfaces en nuestras clases.
- Delegar la implementación de una interfaz a un objeto contenido.
- Definir e implementar propiedades en una interfaz.

A este nivel debemos tener claro cómo podemos usar las interfaces en nuestras aplicaciones Delphi. En el próximo capítulo discutiremos cómo las interfaces se relacionan con la programación COM.

Resumen

En este capítulo hemos examinado las interfaces en detalle. Hemos mostrado cómo podemos usar interfaces internamente en nuestras aplicaciones Delphi, y hemos sentado las bases para el resto de los capítulos de este libro. Por ahora debe conocer cómo:

- Declarar una interfaz
- Implementar una interfaz en una clase.
- Implementar la funcionalidad requerida de *IUnknown*.
- Evitar el conteo de referencias para desactivar la destrucción automática de los objetos.
- Obtener un puntero a una interfaz, dada otra interfaz.
- Implementar múltiples interfaces en nuestras clases.
- Delegar la implementación de una interfaz a un objeto contenido.
- Definir e implementar propiedades en una interfaz.

A este nivel debemos tener claro cómo podemos usar las interfaces en nuestras aplicaciones Delphi. En el próximo capítulo discutiremos cómo las interfaces se relacionan con la programación COM.

2

Las interfaces y COM

En el capítulo anterior examinamos las interfaces como elementos del lenguaje. Aprendimos cómo las interfaces pueden ser usadas internamente para desarrollar una aplicación. Sin embargo, el propósito principal de las interfaces está relacionado con el desarrollo de aplicaciones COM.

En este capítulo examinaremos cómo las interfaces son usadas en los objetos COM. Abordaremos los dos tipos de servidores COM y construiremos algunos ejemplos pequeños, pero muy útiles.

Los ejemplos de servidores COM presentados en este capítulo son atípicos, pues carecen de bibliotecas de tipos. En un esfuerzo por introducir sólo un concepto cada vez, hemos pospuesto la discusión de las bibliotecas de tipos hasta el Capítulo 3 "Bibliotecas de Tipos". Como resultado, los servidores COM que crearemos en este capítulo son útiles solamente si creamos el servidor y la aplicación cliente en Delphi (u otro lenguaje que soporte completamente las tablas de métodos virtuales). Después que hayamos introducido bibliotecas de tipos, mostraremos cómo crear servidores COM que puedan ser utilizados desde casi cualquier plataforma de programación disponible en la actualidad.

Los GUIDs y COM

En el Capítulo 1, "Las Interfaces en Delphi", definimos qué es un GUID. Planteamos que la mayoría de las interfaces internas a una aplicación y *TODAS* las interfaces COM requieren un GUID. Detengámonos un momento para entender el rol que juegan los GUIDs dentro de la programación COM.

En el Registro de Windows hay una clave llamada *HKEY_CLASSES_ROOT\CLSID*. Si abrimos este nodo y nos desplazamos varias páginas hacia abajo, veremos numerosas líneas ocupadas por GUIDs. La Figura 2.1 muestra una parte de la sección *CLSID* del Registro de Windows de mi ordenador.

Nota

Un *CLSID* ("class ID", identificador de clase) no es más que un nombre de un tipo particular de GUID.

Figura 2.1: El Registro de Windows contiene un listado de las clases COM registradas.

Cada *CLSID*, o *GUID* representa una implementación de una interfaz COM. Por ejemplo, el primer *CLSID* listado en mi ordenador es {*00000010-0000-0010-8000-00AA006D2EA4*}. Este *CLSID* suministra la interfaz al motor de datos *Microsoft Data Access Objects* (*DAO*). La clave *InprocServer32* bajo el *GUID* contiene información que es usada por Windows para localizar la DLL correspondiente al motor DAO en mi ordenador. Los servidores internos a procesos ('*internos al proceso*') son examinados posteriormente en detalle en este capítulo, en la sección titulada "Servidores COM internos al proceso".

La clave *ProgID* especifica un nombre asequible al ser humano mediante el cual la interfaz puede ser referida para fines de automatización. La automatización se estudia en detalle en el Capítulo 4, "Automatización".

Objetos COM y Fábricas de Clases

Como mencionamos previamente, un objeto COM no es más que un objeto que implementa una o más interfaces. Los objetos COM son implementados en ficheros ejecutables o en DLLs para ser usados por una aplicación cliente. Todas las clases escritas en el Capítulo 1 fueron usadas únicamente dentro de nuestras aplicaciones Delphi. En este capítulo crearemos objetos que pueden ser usados por cualquier lenguaje que soporte COM. Bueno, al menos en teoría: Visual Basic, por ejemplo, no comprende verdaderamente los métodos de la programación orientada a objetos (la programación en Visual Basic no es realmente orientada a objetos), así que los objetos creados en este capítulo no serán funcionales para programadores de VB. En el Capítulo 4 nos centraremos en escribir servidores COM que puedan ser usados no sólo por Delphi, C++ y

Visual Basic, sino también por aplicaciones como Microsoft Word o Internet Explorer. Pero iremos paso a paso...

Un objeto COM reside dentro de una DLL o de un EXE. Los objetos COM que residen dentro de una DLL son denominados servidores internos al proceso (*in-process servers*), mientras que los que residen dentro de EXEs son denominados servidores externos al proceso (*out-of-process servers*). Más adelante en este capítulo examinaremos los servidores internos y externos.

Los servidores COM pueden contener uno o más objetos COM. Los objetos COM son examinados en la siguiente sección.

Objetos COM

En el Capítulo 1 derivamos la mayoría de nuestras clases de *TInterfacedObject*. En lugar de ello, en este capítulo derivaremos de *TComObject*, ya que *TInterfacedObject* no brinda la funcionalidad necesaria para implementar objetos COM.

He aquí la declaración de *TComObject*, obtenida del fichero *ComObj.pas*:

```
type
  TComObject = class(TObject, IUnknown, ISupportErrorInfo)
  private
    FController: Pointer;
    FFactory: TComObjectFactory;
    FNonCountedObject: Boolean;
    FRefCount: Integer;
    FServerExceptionHandler: IServerExceptionHandler;
    function GetController: IUnknown;
  protected
    { IUnknown }
    function IUnknown.QueryInterface = ObjQueryInterface;
    function IUnknown._AddRef = ObjAddRef;
    function IUnknown._Release = ObjRelease;
    { M□todos de IUnknown para otras interfaces }
    function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
    { ISupportErrorInfo }
    function InterfaceSupportsErrorInfo(const iid: TIID): HResult;
stdcall;
  public
    constructor Create;
    constructor CreateAggregated(const Controller: IUnknown);
    constructor CreateFromFactory(Factory: TComObjectFactory;
      const Controller: IUnknown);
    destructor Destroy; override;
    procedure Initialize; virtual;
    function ObjAddRef: Integer; virtual; stdcall;
    function ObjQueryInterface(const IID: TGUID; out Obj): HResult;
```

```
    virtual; stdcall;
  function ObjRelease: Integer; virtual; stdcall;
  function SafecallException(ExceptObject: TObject;
    ExceptAddr: Pointer): HResult; override;
  property Controller: IUnknown read GetController;
  property Factory: TComObjectFactory read FFactory;
  property RefCount: Integer read FRefCount;
  property ServerExceptionHandler: IServerExceptionHandler
    read FServerExceptionHandler write FServerExceptionHandler;
end;
```

Observe que *TComObject* se deriva de *TObject*, no de *TInterfacedObject*, por lo que debe suministrar una implementación para *_AddRef*, *_Release* y *QueryInterface*. *TComObject* usa técnicas de resolución de métodos para mapear estos métodos a *ObjAddRef*, *ObjRelease* y *ObjQueryInterface*, respectivamente.

Nota

TComObject suministra dos implementaciones de *_AddRef*, *_Release* y *QueryInterface*. La interfaz *IUnknown* es mapeada a *ObjAddRef*, *ObjRelease* y *ObjQueryInterface*, según hemos explicado antes. *_AddRef*, *_Release* y *QueryInterface* dirigen el control a través del puntero interno *FController* (que es realmente una referencia a otra interfaz, usada para soportar una especie de agregación híbrida). Es extremadamente raro que un servidor COM soporte este tipo de agregación híbrida, por lo que no la abordaremos con más detalle en este libro.

En caso de que le haya surgido la duda acerca de por qué *TComObject* no desciende de *TInterfacedObject*, la respuesta es muy simple: los arquitectos de Delphi crearon *TComObject* mucho antes que *TInterfacedObject*. *TInterfacedObject* fue introducida cuando se hizo evidente que una clase "ligera" que utilizara conteo de referencias era necesaria para implementaciones no relacionadas con COM.

TComObject parece complicada, pero afortunadamente no necesitamos lidiar con ella, ya que Borland nos proporciona una implementación. Todo lo que necesitamos hacer es derivar nuestros objetos COM de *TComObject*, y la VCL (y COM) hará el resto.

HResult y OleCheck

En la programación COM, la mayoría de las funciones (con la excepción de *_AddRef* y *_Release*) devuelven un valor de tipo *HResult*. Un *HResult* es un valor especial que indica si la llamada a la función fue exitosa o no, además de contener un código de error en caso de fallo. En la programación COM tradicional es común ver código parecido al siguiente:

```
if Failed(MyComObject.SomeFunction) then begin
  ShowMessage('Imposible ejecutar SomeFunction');
  Exit;
end;
if Failed(MyComObject.SomeOtherFunction) then begin
  ShowMessage('Imposible ejecutar SomeOtherFunction');
  Exit;
```

```
end;
```

Como puede ver, la mayoría del código está destinado al control de los errores que pudieran ocurrir durante una llamada a función. La filosofía de Delphi consiste en no mezclar los valores de retorno con el manejo de excepciones. Está claro que el código anterior no es consecuente con esa filosofía. Afortunadamente, Delphi proporciona un procedimiento llamado *OleCheck*, que nos permite reescribir el código anterior de la siguiente forma:

```
OleCheck(MyComObject.SomeFunction);
OleCheck(MyComObject.SomeOtherFunction);
```

Obviamente este código es corto y más fácil de escribir y de entender. Utilizaremos *OleCheck* cuando realicemos llamadas a funciones COM que devuelvan un *HResult*.

Fábricas de clases

Los objetos COM no son instanciados directamente en las aplicaciones. En lugar de ello**,** COM usa una fábrica de clases (*class factory*) para crear el objeto. Una fábrica de clases es un objeto cuyo propósito es crear otros objetos. Cada objeto COM tiene una fábrica de clases asociado. La fábrica de clases es responsable de la creación del objeto COM, que será implementado en el servidor.

Nota

Clases duales en las fábricas de clases

Podríamos preguntarnos por qué existe esta dualidad de clases. Podría parecer excesivo que se requiera una fábrica de clases para todos y cada uno de los objetos COM. Sin embargo, COM no requiere una fábrica de clases diferente para cada objeto. Una fábrica de clases puede manejar la creación de múltiples objetos COM. De hecho, en Delphi hay sólo una instancia de fábrica de clases que COM ve. Esta fábrica de clases posee una lista de todos los objetos de la fábrica de clase presentes en el servidor COM. Cuando COM llama a la fábrica de clases para solicitar una instancia de un nuevo objeto COM, ésta recorre una lista interna de clases auxiliares buscando una coincidencia.

Las fábricas de clases aíslan a COM del proceso de construcción de un objeto. Si no fuera por la fábricas de clases, COM tendría que hacer llamadas directas a los constructores de los objetos para crearlos. COM no impone restricciones a cómo son implementados los objetos COM, y la construcción es una parte integral del proceso de implementación, así que es importante que COM no tenga conocimiento directo del proceso de construcción del objeto.

Aunque las DLLs pueden exportar funciones estándar que COM podría llamar para obtener una instancia de un objeto, los EXEs no pueden. Por ejemplo, una DLL podría exportar una función llamada *ConstructMyCOMObject*, e indicarle a COM que llame a esa función cada vez que necesite crear una instancia de *MyCOMObject*.

Los EXEs deben registrar sus fábricas de clases cuando son creados, y COM hace llamadas a la interfaz de la fábrica de clases para crear los objetos COM. En aras de la consistencia, las DLLs crean y registran las fábricas de clases de igual forma que los EXEs.

Las fábricas de clases soportan la interfaz *IClassFactory*, que se define de la siguiente forma:

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

Truco

Cuando vea un GUID compuesto por números muy pequeños, como los usados en *IClassFactory* (['{*00000001-0000-0000-C000-000000000046*}']), puede asumir que la interfaz probablemente forme parte del núcleo de interfaces COM definidas por Microsoft. Los GUIDs generados por terceras partes, como Borland o cualquier otro, serán más aleatorios, y no estarán formados principalmente por ceros.

Podemos encontrar la definición de las interfaces de Microsoft en el código fuente de *ActiveX.pas*. Sin embargo, si necesita información adicional sobre las interfaces definidas por Microsoft, la mejor referencia es Microsoft Developer's Network (MSDN). El MSDN está disponible online en *http://msdn.microsoft.com*, o a través de la compra a su distribuidor habitual (o en Danysoft). El contenido online está más actualizado que el de los CDs, pero si no dispone de una conexión rápida a Internet, podría llevar mucho tiempo encontrar la información buscada.

Como podemos ver, *IClassFactory* define únicamente dos funciones: *CreateInstance* y *LockServer*. *CreateInstance* es la función responsable de la creación de una instancia del objeto COM al que esta fábrica hace referencia. Normalmente no llamaremos a esta función. Como veremos en un momento, COM se encarga de llamar a la función por nosotros.

Normalmente, los servidores COM son descargados de memoria cuando no existen clientes activos haciendo uso de ellos. La función *LockServer* puede ser llamada para forzar que el servidor permanezca en memoria. Una llamada *LockServer(True)* incrementa el contador interno de bloqueos, mientras que una llamada *LockServer(False)* decrementa dicho contador. Cuando el contador de bloqueos es cero el servidor puede ser descargado de memoria si no hay clientes haciendo referencia al mismo.

Observe que las llamadas a *LockServer(True)* y *LockServer(False)* deben estar balanceadas para que todo funcione correctamente. Una llamada a *LockServer(True)* sin la correspondiente llamada a *LockServer(False)* proporciona una manera segura de mantener el servidor COM cargado en memoria para siempre.

Nota

Probablemente nunca necesitará llamar a *LockServer*. *LockServer* es un remanente de los antiguos días de los documentos OLE embebidos, cuando los servidores COM eran EXEs separados que tardaban mucho tiempo en cargarse (por ejemplo, Microsoft Word). Por

razones de eficiencia, era beneficioso mantener el servidor cargado en memoria aún cuando no estuviera siendo usado.

Cuando usamos los asistentes de Delphi para crear objetos COM, Delphi añade automáticamente el código necesario para crear la fábrica de clases correspondiente al objeto.

Ahora ya tenemos un conocimiento aceptable de qué son los objetos COM y las fábricas de clases. La próxima sección describe en detalle los servidores COM internos al proceso.

Servidores COM internos al proceso

El primer servidor COM que vamos a analizar es un servidor interno al proceso. Los servidores internos toman su nombre porque son implementados dentro de una DLL, y por consiguiente, ocupan el mismo espacio de direcciones (proceso) que la aplicación que está haciendo uso de ellos. Todos los servidores internos al proceso exportan cuatro funciones estándar: *DllRegisterServer*, *DllUnregisterServer*, *DllGetClassObject*, y *DllCanUnloadNow*. Borland proporciona una implementación de estas funciones en *ComServ.pas*. Por consiguiente, no necesitamos codificar estas funciones, pero sí deberíamos comprender sus objetivos.

- **DllRegisterServer.** *DllRegisterServer* es llamada automáticamente por dos vías diferentes. La primera es a través de la opción "*Register Active X Server*" (Registrar Servidor ActiveX) del menú del entorno integrado de Delphi. La segunda es a través de la utilidad de Windows *RegSvr32* (o la utilidad equivalente de Borland, *TRegSvr*). Independientemente de la vía de llamada, *DllRegisterServer* registra nuestros objetos COM en el Registro de Windows. Al inicio del capítulo hemos presentado una pantalla que mostraba un cierta cantidad de objetos COM en el Registro. Los servidores COM internos al proceso disponen además de una subclave llamada *InprocServer32*, que contiene la ruta completa del servidor COM.
- **DllUnregisterServer.** *DllUnregisterServer* efectúa el proceso inverso a *DllRegisterServer*. Esta función elimina todas las entradas asociadas al objeto COM en el Registro de Windows, que fueron colocadas allí por *DllRegisterServer*.
- **DllGetClassObject.** *DllGetClassObject* es responsable de suministrar una fábrica de clases a un COM, dado un objeto COM a crear. Recordemos que cada servidor COM implementará una fábrica de clases para cada objeto COM que exporte.
- **DllCanUnloadNow.** COM es responsable de llamar a *DllCanUnloadNow* para ver si es posible descargar el servidor COM de memoria. Si alguna aplicación mantiene referencias a algún objeto COM del servidor en

cuestión, *DllCanUnloadNow* devuelve *S_FALSE*. Si no quedan referencias a los objetos COM del servidor, entonces *DllCanUnloadNow* devuelve *S_TRUE*, y COM elimina el servidor de la memoria.

Soporte de múltiples hilos

El soporte multihilos tiene relación solamente con los servidores internos al proceso, y no es aplicable a los servidores externos. Los servidores internos al proceso pueden adherirse a uno de entre varios modelos de multihilado. El modelo asociado a un servidor interno se almacena en el Registro de Windows. Los modelos de multihilado que puede soportar un servidor interno al proceso son:

- *Single* (de un solo hilo): Los Objetos COM que sopo no necesitamos preocuparnos de que múltiples hilos intenten acceder al servidor al mismo tiempo. Todos los accesos al servidor COM ocurren sobre el hilo en el que el servidor COM fue creado.
- rtan el modelo de un solo hilo realmente no ofrecen en absoluto soporte para el multihilado. Todos los accesos al servidor COM son serializados por Windows, así que
- *Apartment* (compartimentado): Los objetos COM que soportan el modelo compartimentado (o de un solo hilo compartido, como es a veces llamado) sólo pueden procesar las peticiones recibidas desde el hilo en que han sido creados. Un servidor puede exportar varios objetos COM, y cada objeto COM podría ser creado por un hilo diferente. Por esta razón, el acceso a cualquier dato global definido en el servidor tiene que ser sincronizado a través de un objeto de exclusión mutua (*mutex*), evento, sección crítica o algún otro método de sincronización.
- *Free* (libre): El modelo de servidores de multihilado libre elimina la restricción impuesta por el modelo compartimentado. En el modelo de multihilado libre, múltiples hilos pueden estar activos simultáneamente sobre cualquier objeto COM. Por esta razón, no sólo el acceso a los datos globales tiene que ser sincronizado, sino que también el acceso a los datos locales debe ser sincronizado para el acceso de múltiples hilos.
- *Apartment* & *Free* (compartimentado & libre): Los servidores COM que se adhieren a esta opción cumplen a la vez el modelo compartimentado y el libre. Este es el modelo más difícil de soportar, ya que los servidores COM que soportan ambos modelos, deben sincronizar sus propios accesos a las instancias de objetos compartimentadas y serializar (*marshal*) los parámetros que traspasen las fronteras de hilos.

En la mayoría de los casos, es conveniente escribir servidores COM que cumplan con el modelo compartimentado. La biblioteca de tiempo de ejecución y las clases COM de Delphi 5 (implementadas en *ComServ.pas* y *ComObj.pas*) soportan todos los modelos de multihilado, incluyendo el multihilado libre. El modelo libre ofrece un

notable incremento del rendimiento en servidores COM alojados por MTS e IIS, casos en el que el huésped administra cierta cantidad de hilos de trabajo para gestionar un gran número de peticiones de clientes, y donde la mayoría de los métodos COM utilizados realizan accesos de lectura en lugar de escritura.

En la mayoría de las situaciones, incluyendo aplicaciones estándar y controles visuales, no hay ninguna ventaja al utilizar el modelo hilo libre. Por consiguiente, el trabajo adicional requerido para hacer seguras nuestras clases no será rentable. Por otro lado, los servidores COM de un solo hilo son comúnmente muy restrictivos, al no permitir el acceso desde diferentes hilos. A menos que necesite la ganancia de rendimiento asociada a la implementación del multihilado libre en servidores COM bajo MTS o IIS, o tenga razones apremiantes para permanecer con un servidor COM de hilo único, deberá escribir servidores COM que utilicen el modelo compartimentado.

Registrando el servidor

Todos los servidores COM necesitan ser registrados ante Windows para que funcionen adecuadamente. El proceso de registro implica añadir las entradas necesarias al Registro de Windows, de tal forma que Windows conozca la localización y tipo de servidor (interno o externo al proceso).

Como programador de Delphi, puede registrar servidores COM mediante la opción *Run|Register ActiveX Server* (Registrar Servidor ActiveX) desde el menú principal de Delphi. Por supuesto, los usuarios finales de una aplicación no tendrán instalado Delphi en sus ordenadores. Windows nos brinda una utilidad de registro llamada *RegSvr32*. Para registrar un servidor COM podemos simplemente escribir

```
RegSvr32 <NombreDelServidor>
```

Sin embargo, un enfoque aún más sencillo consiste en registrar el servidor a través de su programa de instalación. La mayoría de los programas de instalación del tipo InstallShield, suministran un mecanismo para registrar los servidores COM como parte del proceso de instalación.

Independientemente del método utilizado para registrar el servidor, los detalles son los mismos. El programa (*Regsvr32*, *InstallShield*, etc.) simplemente llama a la función *DllRegisterServer* que reside dentro de la DLL que implementa el servidor COM. La función *DllRegisterServer* es responsable de incorporar las entradas necesarias al Registro para que el servidor interno al proceso funcione de forma adecuada.

Como mencionamos anteriormente, la implementación de *DllRegisterServer* en Delphi lo hace todo por nosotros. Si desea investigar los detalles de esta función, puede encontrarla en *ComServ.pas*.

Si eliminamos un servidor interno al proceso de un ordenador, podemos desregistrarlo desde menú principal de Delphi seleccionando *Run|Unregister ActiveX Server*, o ejecutando:

```
RegSvr32 -u  <NombreDelServidor>
```

La mayoría de los programas de instalación tienen la capacidad de eliminar un servidor interno al proceso durante el proceso de desinstalación.

Constructores a medida

No debemos intentar redefinir el constructor de un objeto COM. Todos los constructores definidos por *TComObject* llaman al método virtual *Initialize*. Si necesitáramos ejecutar código de inicialización en nuestros objetos COM, la solución consiste en redefinir el método *Initialize* en lugar del constructor. *Initialize* se define de la siguiente forma:

```
procedure Initialize; virtual;
```

La razón para colocar código en *Initialize* y no en el constructor es que las clases base de los objetos COM en Delphi contienen varios constructores no virtuales. Dependiendo del escenario, la fábrica de clases llamará a diferentes constructores en diferentes instancias. El método *Initialize* es virtual y es el único método que es llamado independientemente del constructor utilizado para crear el objeto COM.

Creando una instancia de un objeto COM interno al proceso.

Cuando deseemos crear una instancia de un objeto COM interno al proceso en nuestro código cliente de Delphi, normalmente usaremos la función *CreateComObject*, declarada en *ComObj.pas* como:

```
function CreateComObject(const classID: TGUID): IUnknown;
```

CreateComObject recibe como parámetro el GUID del objeto COM que deseamos crear, y retorna un puntero a la interfaz *IUnknown* del objeto. Si el GUID solicitado no puede ser localizado en el Registro de Windows, *CreateComObject* lanza una excepción.

Tomemos un momento para analizar la implementación de *CreateComObject*, que es muy corta:

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

Como puede ver, todo lo que hace la función *CreateComObject* es llamar a la función de Windows *CoCreateInstance*, pasando ciertos valores por defecto como parámetros. *CoCreateInstance* tiene cinco parámetros, y se define de la siguiente forma:

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

El primer parámetro, *clsid*, es el GUID del servidor COM que deseamos crear. Este es el primer y único parámetro que pasamos específicamente a *CreateComObject*. El parámetro siguiente, *unkOuter*, es usado sólo si el objeto COM es parte de una agregación. Ya discutimos brevemente la agregación en el Capítulo 1, y no nos detendremos más en ello. Si está interesado en la agregación de objetos COM, consulte la clase VCL *TAggregatedObject*, definida en *ComObj.pas*.

El tercer parámetro, *dwClsContext*, determina el tipo de servidor que deseamos crear. *CreateComObject* permite solicitar automáticamente un servidor interno al proceso (*CLSCTX_INPROC_SERVER*) o un servidor local externo al proceso (*CLSCTX_LOCAL_SERVER*). El otro identificador que es a veces utilizado con esta función es *CLSCTX_REMOTE_SERVER*. Este valor se utiliza cuando se está trabajando con DCOM. Examinaremos DCOM en detalle en el Capítulo 5, "Controles ActiveX y ActiveForms".

El cuarto parámetro, *iid*, es la interfaz del objeto cuya referencia deseamos obtener. Delphi siempre solicita una referencia a la interfaz *IUnknown*, ya que la mayoría de las fábricas de clase fallarán si preguntamos por cualquier otra. (El objetivo de este parámetro es utilizarlo en futuras extensiones de Microsoft). El último parámetro, *pv*, recibirá un puntero a la interfaz *IUnknown*.

Nota

Algo que de lo que debemos ser conscientes es que *CoCreateInstance* crea internamente una instancia de la fábrica de clases responsable de crear el objeto COM, y después usa la fábrica de clases para crear el objeto. Una vez que el objeto COM ha sido creado, la fábrica de clases es destruida.

Por supuesto, esto será muy ineficiente cuando se desee crear numerosas instancias del mismo objeto COM. En estos casos, podemos crear manualmente una instancia de la fábrica de clases y utilizar su método *CreateInstance* para crear los objetos COM, para finalmente eliminarla.

Como mencionamos con anterioridad, *CreateComObject* siempre devuelve un puntero a *IUnknown*. Para obtener un puntero a la interfaz que nos interesa debemos usar el operador **as**, de la siguiente forma:

```
MyIntf := CreateComObject(CLSID_MyServer) as IMyInterface;
```

Esta técnica será mostrada en los ejemplos del capítulo.

Nota

Tabla de métodos virtuales

Delphi implementa las interfaces como tablas de métodos independientes, que aparecen en memoria a continuación de la tabla de métodos virtuales (VMT). Como ejemplo, consideremos las siguientes definiciones:

```
type
  IInterface1 = interface
    procedure DoInt1;
    procedure GetName1: WideString;
  end;
  IInterface2 = interface
    procedure DoInt2;
    procedure DoSomethingElse;
  end;
  TMyClass = class(TComObject, IInterface1, IInterface2)
  protected
    // implementaci□n de IInterface1
    procedure DoInt1;
    function GetName1: WideString:
    // implementaci□n de IInterface2
    procedure DoInt2;
    procedure DoSomeThingElse;
    // m□todos virtuales de TMyClass
    procedure WorkHard; virtual;
    function IsDone: Boolean; virtual;
  end;
```

Para abreviar, los GUIDs han sido eliminados del código.

La disposición en memoria de las tablas de métodos de la clase se muestra en la Figura 2.2:

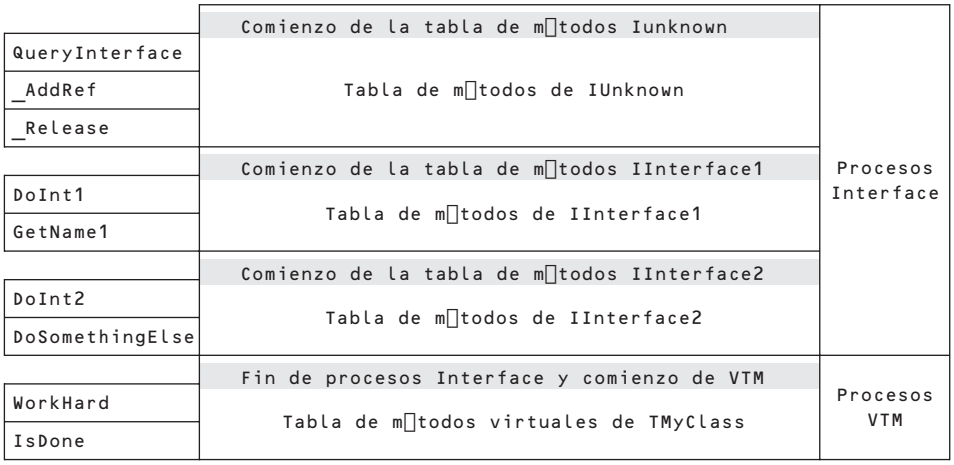


Figura 2.2: Las tablas de métodos de las interfaces se colocan en memoria secuencialmente, y la VMT se coloca a continuación.

Dado que diferentes tablas de interfaces ocupan zonas diferentes de memoria, no podemos simplemente convertir de una interfaz a otra. El siguiente código:

```
MyIntf1 := IInterface(CreateComObject(CLSID_MyServer));
```

fallará miserablemente al tratar de convertir una interfaz *IUnknown* a *IInterface1*. Utilice siempre el operador as para convertir de una interfaz a otra, como describimos en el Capítulo 1.

Ahora que conocemos qué es un servidor COM interno al proceso, la próxima sección nos conducirá paso a paso en la creación de un servidor COM interno al proceso funcional.

Ejemplo: Distribución unidimensional óptima

Suponga que tenemos una aplicación que hemos escrito para la industria de transporte. Una de las tareas de la aplicación es cargar cada camión tan cerca como sea posible de su capacidad máxima, sin exceder un peso límite predeterminado. Pongamos por caso que el máximo peso que admiten los camiones es de 10.000 libras.

Es responsabilidad de la aplicación transportar todos los productos utilizando el menor número de camiones posible. Por supuesto, si tenemos 40.000 libras de materiales para transportar, se requerirá más de un camión.

Este problema es un ejemplo de problema *NP-completo*. Los problemas NP-completos tienen la dudosa distinción de no tener una solución óptima que pueda ser calculada en un tiempo lineal. La única forma de aproximarse a la mejor solución es probar cada combinación individual de productos distribuidos sobre los camiones. A no ser que se trate de una cantidad pequeña de productos, el tiempo requerido para probar cada combinación es excesivo.

Consideremos el caso de que tengamos 100 productos a transportar. Si se requieren cinco camiones para transportar los productos, entonces existen 5^{100} combinaciones posibles para distribuir los productos en los camiones. Claramente, no hay tiempo para considerar cada combinación, especialmente si esperamos transportar los productos en un futuro cercano.

Lo que necesitamos, entonces, es una *heurística*, o una forma de determinar una solución casi óptima en un período de tiempo razonable. Existen varios algoritmos para esto. El objetivo de este capítulo no es enseñarnos algoritmos de distribución, sino más bien ilustrar cómo se puede utilizar un servidor COM para implementar uno o más algoritmos de distribución.

La aplicación que estamos escribiendo está sometida a severas presiones de tiempo (¿qué aplicación no lo está?), y por tanto no tenemos tiempo para investigar varios algoritmos y determinar cuál es el mejor. Después de pensarlo unos segundos,

hemos llegado a una solución factible, aunque lejos de ser óptima. Para entender cómo trabaja este algoritmo, asumamos que tenemos que transportar los productos listados en la Tabla 2.1:

Tabla 2.1: Listado de los productos a transportar

Producto	Peso	Cantidad
Sofá	300	4
Mesa	200	6
Cama	150	4
Silla	50	20

El peso total de estos productos es de 4.000 libras. Para simplificar, asumamos que un camión puede llevar sólo 1.000 libras. Para implementar nuestro algoritmo, recorreremos cada producto de la lista. Si el camión puede cargar el producto, lo colocamos. En caso contrario, se considera que el camión está lleno y se comienza la carga de un nuevo camión.

Tomemos el primer producto, un sofá. Necesitamos comenzar un nuevo camión para colocar el sofá. El segundo producto, otro sofá, puede ser también colocado en el camión, al igual que el tercero. Todavía nos resta un sofá, cuyo peso excedería la capacidad de carga del camión. El primer camión es enviado a su destino, y el segundo camión es llamado para cargar el último sofá.

Obviamente, al primer camión todavía le restaban 100 libras de capacidad de carga, y podría haber llevado dos de las 20 sillas. Esta es una limitación del algoritmo *NextFit* que desarrollaremos en este capítulo.

La Tabla 2.2 muestra cómo se producirá la distribución de los productos.

Tabla 2.2: Resultado de la distribución usando el algoritmo *NextFit*

Camión #	Carga
1	3 sofás
2	1 sofá, 3 mesas
3	3 mesas, 2 camas
4	2 camas, 14 sillas
5	6 sillas

Como puede calcularse, el número óptimo de camiones necesarios para esta relación de productos es de 4 (4.000 libras – 1.000 libras por camión), pero este algoritmo ha requerido del uso de 5 camiones. Nuestros clientes nos han dado el

visto bueno a esta implementación por limitaciones de tiempo, pero nos han pedido que en el futuro tratemos de implementar un algoritmo mejor.

Creando un servidor COM que implementa una distribución unidimensional

Para crear un servidor COM que implemente este algoritmo, seleccione *File|New*. En la ventana de diálogo *NewItems* que aparece, seleccione la pestaña *ActiveX*. La página *ActiveX* se muestra en la Figura 2.3.

Figura 2.3: La página *ActiveX* del Depósito de Objetos.

Haga doble clic sobre el elemento *ActiveX Library*, o seleccione el elemento *ActiveX Library* y después pulse OK. Esto creará un servidor COM implementado en una DLL:

```
library Project1;
uses
 ComServ;
exports
 DllGetClassObject,
 DllCanUnLoadNow,
 DllRegisterServer,
 DllUnregisterServer;
{$R  *.RES}
begin
end.
```

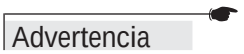
Observe que Delphi ha añadido de forma automática *DllGetClassObject*, *DllCanUnloadNow*, *DllRegisterServer* y *DllUnregisterServer* a la sección **exports** del módulo. No tenemos que escribir ningún código para soportar estas funciones.

Guarde el fichero, seleccionando *File|Save* del menú. Llamémosle *Bin1Srv*.

Ahora necesitamos crear la declaración de la interfaz, que llamaremos *IOneDBin*. Definiremos la interfaz *IOneDBin* como:

```
type
  IOneDBin = interface
    ['{7856B7E3-EF75-11D2-B3AB-0040F67455FE' ]
    procedure SetMaxValue(AMaxValue: Integer);
    procedure AddItem(AQuantity: Integer; ADescription: Integer;
      AValue: Integer);
    procedure Optimize;
    function NextBin: Boolean;
    function NextItem(var ADescription: Integer; var AValue: Integer);
  end;
```

Una vez más, el GUID en la declaración de la interfaz anterior fue generado presionando **Ctrl+May+G**. Nunca debemos copiar un GUID de otro objeto.



Advertencia

Me verá repetir una y otra vez en este libro que cada interfaz que creemos requiere de un GUID único. No nos debe preocupar la idea de que se agoten los GUIDs en nuestro ordenador: podemos generar un nuevo GUID cada segundo ¡sin que se agoten en un millón de años!

Seleccione *File|New* y la pestaña *ActiveX* en el cuadro de diálogo *New Items*. Seleccione el icono '*COM Object*' y pulse OK.

Nota

Delphi 3 no proporciona un asistente para crear objetos COM. Si utiliza Delphi 3, deberá escribir el código generado por el asistente. La cantidad de código generado por el asistente es mínimo, y se muestra en el Listado 2.1.

Delphi 3 tampoco soporta directamente múltiples hilos en objetos COM, y como resultado, la llamada a *TComObjectFactory.Create* en Delphi 3 es ligeramente diferente que en versiones posteriores. No incluya el parámetro final, *tmApartment*, si está utilizando Delphi 3.

Delphi nos muestra el asistente para objetos COM. Rellene los campos como se muestra en la Figura 2.4, y luego pulse OK. Podemos ignorar la opción *Instancing*, pues sólo se relaciona con los servidores externos al proceso. En *Options*, mantenga desmarcada la opción '*Include Type Library*' (Incluir Biblioteca de Tipos). Las bibliotecas de tipos se examinarán en detalle en el Capítulo 3.

Figura 2.4: Asistente de Delphi para Objetos COM

Guarde este fichero como NextFit.pas. El código de NextFit.pas se muestra en el listado 2.1.

Listado 2.1: *NextFit.pas*

```
unit NextFit;
interface
uses
  Windows, ActiveX, ComObj;
type
  TNextFit = class(TComObject, IOneDBin)
  protected
    {Declare IOneDBin methods here}
  end;
const
  Class_NextFit: TGUID = '{7856B7E2-EF75-11D2-B3AB-0040F67455FE}';
implementation
uses ComServ;
initialization
  TComObjectFactory.Create(ComServer, TNextFit, Class_NextFit,
   'NextFit', 'Next-fit algorithm', clMultiInstance, tmApartment);
end.
```

TNextFit es la clase que ha sido generada para nosotros. Recuerden que en el asistente debemos especificar el nombre de la clase como *NextFit*. Delphi añade automáticamente la 'T' como prefijo.

La constante *Class_NextFit* es un GUID que representa el servidor COM. Cada interfaz implementada en el servidor también tendrá su propio GUID.

La sección de inicialización de la unidad contiene una única pero compleja llamada a un constructor. Esta llamada instala la fábrica de clases responsable de la creación del objeto COM *TNextFit*. *TComObjectFactory.Create* se define en *ComObj.pas*:

```
constructor Create(ComServer: TComServerObject; ComClass: TComClass;
   const ClassID: TGUID; const ClassName, Description: string;
   Instancing: TClassInstancing; ThreadingModel: TThreadingModel =
tmSingle);
```

Echemos un vistazo de cerca a esta llamada. En el 99% de los casos, simplemente pasaremos el objeto global *ComServer* como el parámetro *ComServer*. No discutiremos otros posibles usos de este parámetro es este libro.

El segundo parámetro, *ComClass*, recibe la clase que será creada por la fábrica de clases. En este ejemplo deseamos que la fábrica de clases cree instancias de *TNextFit*. Por su parte, *ClassID* recibe el GUID asignado a la clase *TNextFit*. El asistente de Delphi automáticamente crea uno para nosotros: *Class_NextFit*. A continuación, pasamos el nombre de la clase, *NextFit*, y una descripción textual de la misma.

El parámetro *Instancing* se aplica sólo a servidores externos al proceso, y lo ignoraremos por el momento. El asistente de objetos COM de Delphi asigna a este parámetro un valor por defecto, *ciMultiInstance*. Por último, el parámetro final acepta un valor que indica el modelo de multihilado soportado por el objeto. Observe que asignar simplemente el valor *tmApartment* a este parámetro no significa que automáticamente el objeto COM soporte el modelo compartimentado. Es nuestra responsabilidad asegurarnos que el código que escribamos cumpla las especificaciones del modelo compartimentado. Este identificador simplemente determina cómo será publicado el objeto en el Registro de Windows.

Movamos la declaración de la interfaz *IOneDBin* a una unidad llamada *BinIntf.pas* para hacer más sencilla nuestra referencia a ella desde el código del cliente (que escribiremos en la siguiente sección). Los Listados 2.2 al 2.4 muestran el código fuente completo del servidor *Bin1Srv*, después de crear la implementación de la clase *TNextFit*.

Listado 2.2: Bin1Srv - *Bin1Srv.dpr*

```
library Bin1Srv;
uses
  ComServ,
  NextFit in 'NextFit.pas',
  Bin1Srv_TLB in 'Bin1Srv_TLB.pas',
  BinIntf in 'BinIntf.pas';
exports
  DllGetClassObject,
  DllCanUnloadNow,
  DllRegisterServer,
  DllUnregisterServer;
{$R *.TLB}
{$R *.RES}
begin
end.
```

Listado 2.3: Bin1Srv - *BinIntf.pas*

```
unit BinIntf;
interface
type
  IOneDBin = interface
    ['{7856B7E3-EF75-11D2-B3AB-0040F67455FE}']
    procedure SetMaxValue(AMaxValue: Integer);
    procedure AddItem(AQuantity: Integer; ADescription: WideString;
      AValue: Integer);
    procedure Optimize;
    function NextBin: Boolean;
    function NextItem(var ADescription: WideString;
      var AValue: Integer): Boolean;
  end;
const
  Class_NextFit: TGUID = '{7856B7E2-EF75-11D2-B3AB-0040F67455FE}';
implementation
end.
```

Listado 2.4: Bin1Srv - *NextFit.pas*

```
unit NextFit;
interface
uses
  Windows, ActiveX, ComObj, Classes, BinIntf;
type
  TBinItem = class
  protected
    FDescription: string;
    FValue: Integer;
  end;
  TBin = class
  protected
    FValue: Integer;
    FItems: TList;
  public
    constructor Create;
    destructor Destroy; override;
  end;
  TNextFit = class(TComObject, IOneDBin)
  protected
    FMaxValue: Integer;
    FItems: TList;
    FBins: TList;
    FCurrentBin: TBin;
    FBinIndex: Integer;
    FItemIndex: Integer;
  public
    procedure Initialize; override;
    destructor Destroy; override;
    {Declare IOneDBin methods here}
```

```
    procedure SetMaxValue(AMaxValue: Integer);
    procedure AddItem(AQuantity: Integer; ADescription: WideString;
      AValue: Integer);
    procedure Optimize;
    function NextBin: Boolean;
    function NextItem(var ADescription: WideString;
      var AValue: Integer): Boolean;
  end;
implementation
uses ComServ;
{ TBin }
constructor TBin.Create;
begin
  FItems := TList.Create;
end;
destructor TBin.Destroy;
begin
  FItems.Free;
end;
{ TNextFit }
procedure TNextFit.AddItem(AQuantity: Integer; ADescription: WideString;
  AValue: Integer);
var
  Item: TBinItem;
  Index: Integer;
begin
  for Index := 1 to AQuantity do begin
    Item := TBinItem.Create;
    Item.FDescription := ADescription;
    Item.FValue := AValue;
    FItems.Add(Item);
  end;
end;
destructor TNextFit.Destroy;
var
  Index: Integer;
begin
  for Index := 0 to FBins.Count - 1 do
    TBin(FBins[Index]).Free;
  FBins.Free;
  for Index := 0 to FItems.Count - 1 do
    TBinItem(FItems[Index]).Free;
  FItems.Free;
end;
procedure TNextFit.Initialize;
begin
  FItems := TList.Create;
  FBins := TList.Create;
end;
function TNextFit.NextBin: Boolean;
begin
  if FBinIndex < FBins.Count - 1 then begin
```

```
    Inc(FBinIndex);
    FCurrentBin := TBin(FBins[FBinIndex]);
    FItemIndex := -1;
    Result := True;
  end
  else
    Result := False;
end;
function TNextFit.NextItem(var ADescription: WideString;
  var AValue: Integer): Boolean;
begin
  if FItemIndex < FCurrentBin.FItems.Count - 1 then begin
    Inc(FItemIndex);
    ADescription := TBinItem(FCurrentBin.FItems[FItemIndex]).FDescription;
    AValue := TBinItem(FCurrentBin.FItems[FItemIndex]).FValue;
    Result := True;
  end else
    Result := False;
end;
procedure TNextFit.Optimize;
var
  Index: Integer;
  Item: TBinItem;
begin
  FCurrentBin := nil;
  for Index := 0 to FItems.Count - 1 do begin
    Item := TBinItem(FItems[Index]);
    if (FCurrentBin = nil)
      or (FCurrentBin.FValue + Item.FValue > FMaxValue) then begin
      FCurrentBin := TBin.Create;
      FBins.Add(FCurrentBin);
    end;
    FCurrentBin.FItems.Add(Item);
    FCurrentBin.FValue := FCurrentBin.FValue + Item.FValue;
  end;
  FBinIndex := -1;
end;
procedure TNextFit.SetMaxValue(AMaxValue: Integer);
begin
  FMaxValue := AMaxValue;
end;
initialization
{$IFDEF VER100}
  TComObjectFactory.Create(ComServer, TNextFit, Class_NextFit,
    'NextFit', 'Next-fit algorithm', ciMultiInstance);
{$ELSE}
  TComObjectFactory.Create(ComServer, TNextFit, Class_NextFit,
    'NextFit', 'Next-fit algorithm', ciMultiInstance, tmApartment);
{$ENDIF}
end.
```

Será valioso tomar unos momentos para explicar el código de este fichero.

TNextFit implementa el algoritmo *IOneDBin* discutido previamente. La aplicación cliente hará sucesivas llamadas a *TNextFit.AddItem* para especificar los productos a optimizar. Por ejemplo, para las piezas de mobiliario listadas en la Tabla 2.1, la aplicación cliente incluiría el siguiente código:

```
NextFit.AddItem(4, 'Sofa', 300);
NextFit.AddItem(6, 'Table', 200);
NextFit.AddItem(4, 'Bed', 150);
NextFit.AddItem(20, 'Chair', 50);
```

En la medida en que los productos van siendo añadidos, *TNextFit* crea un objeto *TBinItem* para cada producto. Por ejemplo, *TNextFit* creará cuatro objetos *TBinItem* para representar los cuatro sofás. Seguidamente, la aplicación cliente asignará el tamaño del contenedor (en nuestro caso, la capacidad del camión):

```
NextFit.SetMaxValue(1000);
```

En este punto, los productos están listos para ser optimizados, así que la aplicación cliente llama a *TNextTFit.Optimize*, que lleva a cabo todo el trabajo de optimización de las piezas.

Optimize utiliza una clase adicional de ayuda durante el proceso de optimización. La clase *TBin* es usada para representar el camión o cualquier contenedor donde son colocados los productos. *TNextFit.Optimize* crea un nuevo objeto *TBin* cada vez que necesite un nuevo contenedor.

Cada uno de los objetos *TBin* contiene un objeto de la clase *TList*, *FItems*, en el que se alojan los productos situados en el contenedor. Los objetos *TBinItem* sólo existen una vez, pero son almacenados simultáneamente en la lista *TNextFit.FItems* y en la lista *TBin.FItems* para preservar memoria.

Después que los productos han sido optimizados, es muy sencillo recorrer los contenedores y listar el contenido de cada uno. El siguiente bucle nos muestra cómo hacerlo:

```
while NextFit.NextBin do begin
  // Comenzar un nuevo contenedor
  while NextFit.NextItem(Description, Value) do begin
    // Hacer algo con la descripci□n y el valor
  end;
end;
```

Antes de poder utilizar el servidor COM, debemos registrarlo. Seleccione *Run|Register ActiveX Server* en el menú principal de Delphi. Al cabo de un momento recibirá un mensaje de confirmación similar al mostrado en la Figura 2.5. Su unidad de disco y directorio podrían diferir del presentado en la figura.

Figura 2.5: Registro exitoso de *Bin1Srv*

En este momento, Delphi crea también una biblioteca de tipos *Bin1Srv.tlb* y un fichero de importación de biblioteca de tipos *Bin1Srv_TLB.pas*. Podríamos pensar que esto es extraño, pues no solicitamos biblioteca de tipos cuando creamos el objeto COM, *TNextFit*. Sin embargo, esta biblioteca de tipos contiene información sobre el servidor COM *Bin1Srv*, y no sobre el objeto COM *TNextFit*. El Listado 2.5 muestra el código fuente de *Bin1Srv_TLB.pas*.

El Capítulo 3 examina las bibliotecas de tipos con más detalle.

Listado 2.5: Bin1Srv - *Bin1Srv_TLB.pas*

```
unit Bin1Srv_TLB;
// ************************************************************************ //
// WARNING                                                                  //
//                                                                          //
// The types declared in this file were generated from data read from a     //
// Type Library. If this type library is explicitly or indirectly (via      //
// another type library referring to this type library) re-imported, or     //
// the 'Refresh' command of the Type Library Editor activated while         //
// editing the Type Library, the contents of this file will be              //
// regenerated and all manual modifications will be lost.                   //
// ************************************************************************ //
// PASTLWTR : $Revision:   1.11.1.75  $
// File generated on 11/11/2000 10:50:01 AM from Type Library described     //
// below.                                                                   //
// ************************************************************************ //
// Type Lib: C:\Book\samples\chap02\Bin1Srv\Bin1Srv.tlb
// IID\LCID: {E97BFEF5-F664-11D2-9099-0040F6741DE2}\0
// Helpfile:
// HelpString: Bin1Srv Library
// Version:    1.0
// ************************************************************************ //
interface
uses Windows, ActiveX, Classes, Graphics, OleCtrls, StdVCL;
// ************************************************************************//
// GUIDS declared in the TypeLibrary. Following prefixes are used:          //
//   Type Libraries     : LIBID_xxxx                                        //
//   CoClasses          : CLASS_xxxx                                        //
//   DISPInterfaces     : DIID_xxxx                                         //
//   Non-DISP interfaces: IID_xxxx                                          //
// ************************************************************************//
const
```

```
  LIBID_Bin1Srv: TGUID = '{E97BFEF5-F664-11D2-9099-0040F6741DE2}';
implementation
uses ComObj;
end.
```

En la próxima sección crearemos una aplicación cliente para manejar el servidor.

Creando la aplicación cliente para la distribución unidimensional

Ahora podemos escribir el programa cliente para probar el servidor. La Figura 2.6 nos muestra la interfaz de usuario del programa de prueba.

El uso de este programa es simple. Introduciremos la cantidad, descripción y valor de un producto y pulsaremos el botón *Add* (Añadir). El programa emitirá un pitido, comunicándonos de una forma rudimentaria que el producto ha sido registrado. Después de haber introducido todos los productos, pulsaremos el botón *Optimize* (Optimizar) para colocar los productos dentro de los contenedores del tamaño especificado en el control de edición *Bin Size* (Tamaño del contenedor).

Figura 2.6: *Bin1Cli* en acción.

La aplicación rellena una vista en árbol indicando cuántos contenedores son necesarios y qué productos van ser colocados en cada contenedor. No hemos implementado ningún control de errores en la aplicación ni en el servidor COM, así que el programa no chequeará cosas como intentar colocar 2.000 libras en un contenedor de 1.000.

Los Listados 2.6 y 2.7 muestran el código fuente de la aplicación *Bin1Cli*.

Listado 2.6: Bin1Cli - *Bin1Cli.dpr*

```
program Bin1Cli;
```

```
uses
  Forms,
  MainForm in 'MainForm.pas' {Form1},
  BinIntf in '..\Bin1Srv2\BinIntf.pas';
{$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Listado 2.7: Bin1Cli - *MainForm.pas*

```
unit MainForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  ComObj, StdCtrls, ComCtrls, BinIntf;
type
  TForm1 = class(TForm)
    grpItems: TGroupBox;
    Label4: TLabel;
    ecQuantity: TEdit;
    Label5: TLabel;
    ecDescription: TEdit;
    Label6: TLabel;
    ecValue: TEdit;
    btnAdd: TButton;
    btnOptimize: TButton;
    treeBins: TTreeView;
    Label1: TLabel;
    ecBinSize: TEdit;
    procedure btnAddClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure btnOptimizeClick(Sender: TObject);
  private
    { Private declarations }
    FOneD: IOneDBin;
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.btnAddClick(Sender: TObject);
begin
  FOneD.AddItem(StrToInt(ecQuantity.Text), ecDescription.Text,
    StrToInt(ecValue.Text));
  Beep;
  ActiveControl := ecQuantity;
```

```
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  FOneD := CreateComObject(Class_NextFit) as IOneDBin;
end;
procedure TForm1.btnOptimizeClick(Sender: TObject);
var
  Desc: String;
  Value: Integer;
  BinCount: Integer;
  Node: TTreeNode;
begin
  FOneD.SetMaxValue(StrToInt(ecBinSize.Text));
  FOneD.Optimize;
  BinCount := 0;
  treeBins.Items.BeginUpdate;
  try
    while FOneD.NextBin do begin
      Inc(BinCount);
      Node := treeBins.Items.AddChild(nil, 'Bin ' + IntToStr(BinCount));
      while FOneD.NextItem(Desc, Value) do
        treeBins.Items.AddChild(Node, Desc);
    end;
  finally
    treeBins.Items.EndUpdate;
  end;
  btnAdd.Enabled := False;
  btnOptimize.Enabled := False;
end;
end.
```

Actualizando el servidor

Han pasado algunos meses, y nuestra aplicación está funcionando perfectamente. Nuestro cliente está satisfecho con la aplicación, sin embargo el algoritmo *NextFit* en muchos casos proporciona resultados inferiores a los óptimos. A los clientes les gustaría que implementáramos un algoritmo de distribución más óptimo. Después de algunas búsquedas, hemos encontrado un algoritmo más eficiente llamado *FirstFit*. *FirstFit* trabaja de forma muy semejante a *NextFit*, con la excepción de que no "cierra" el contenedor cuando encuentra un producto que no se ajuste a él. En cambio, este algoritmo considera todos los contenedores disponibles cuando se intenta colocar un producto. Se "abre" un nuevo contenedor sólo si ninguno de los contenedores ya en uso tiene capacidad para colocar el nuevo producto.

Para el ejemplo, voy a añadir el nuevo objeto *TFirstFit* al servidor ya existente, en lugar de crear un nuevo servidor. Mientras no cambiemos la interfaz, somos libres de modificar el código dentro de la DLL. Por supuesto, si se modifica el código,

habrá que probarlo en profundidad para garantizar que no enviamos una actualización defectuosa.

La mayor parte del código de *TFirstFit* será exactamente el mismo que para *TNextFit*. Por consiguiente, voy a crear una clase abstracta llamada *TAbstractBinPacker*, que implementará la mayor parte de la funcionalidad básica. Derivaremos *TNextFit* y *TFirstFit* de *TAbstractBinPacker*. Lo que quiero mostrar aquí es que se puede cambiar la implementación tanto como se desee sin afectar el código existente. Mientras no se toque la interfaz, las aplicaciones clientes que utilicen el servidor seguirán funcionando exactamente igual.

Por supuesto, como es frecuente, hemos pensado en un par de mejoras que nos gustaría hacer a la interfaz *IOneDBin*. Desearía añadir una rutina que informe cuántos contenedores son necesarios, y otra que nos comunique el porcentaje de espacio no utilizado. De esta forma, podremos ejecutar ambos algoritmos y comparar los resultados. El algoritmo que genere una menor cantidad de espacio residual será el más eficiente para un conjunto dado de datos de entrada.

Nota

Realmente, en el código hemos implementado *TFirstFit* y *TBestFit*, debido a que el código adicional requerido para implementar *TBestFit* es trivial.

Debido a que una interfaz es inmutable una vez publicada, no podemos de forma arbitraria modificar la interfaz *IOneDBin*. En lugar de ello, crearemos una segunda interfaz *IOneDBin2* (contenedor para distribución, versión 2). Hemos modificado el servidor para hacer exactamente esto. Los Listados 2.8 al 2.13 presentan el código fuente para el servidor COM actualizado.

Para actualizar el servidor, ejecutamos el asistente de Delphi para objetos COM dos veces más, creando los objetos COM *TFirstFit* y *TBestFit*, respectivamente. Delphi coloca estos objetos en ficheros separados. Acto seguido, copiamos los GUID de estos dos ficheros y los pegamos en el código fuente del fichero *BinIntf.pas*, de tal forma que la aplicación cliente pueda hacer referencia a ellos.

Seguidamente, creamos el fichero *BinBase.pas*, que define la clase abstracta usada internamente por el servidor. Debido a que la mayor parte de la funcionalidad de los tres métodos de distribución (*NextFit*, *FirstFit* y *BestFit*) es idéntica, sería lógico tener implementadas las funcionalidades comunes en una clase ancestro común.

Los ficheros *NextFit.pas*, *FirstFit.pas* y *BestFit.pas* implementan los métodos *GetName* y *Optimize*, que son diferentes para cada método de distribución.

Listado 2.8: Bin1Srv2 - *Bin1Srv.dpr*

```
library Bin1Srv;
uses
  ComServ,
  Bin1Srv_TLB in 'Bin1Srv_TLB.pas',
  BinIntf in 'BinIntf.pas',
  BinBase in 'BinBase.pas',
  NextFit in 'NextFit.pas',
  FirstFit in 'FirstFit.pas',
  BestFit in 'BestFit.pas';
exports
  DllGetClassObject,
  DllCanUnloadNow,
  DllRegisterServer,
  DllUnregisterServer;
{$R *.TLB}
{$R *.RES}
begin
end.
```

Listado 2.9: Bin1Srv2 - *BinIntf.pas*

```
unit BinIntf;
interface
type
  IOneDBin = interface
    ['{7856B7E3-EF75-11D2-B3AB-0040F67455FE}']
    procedure SetMaxValue(AMaxValue: Integer);
    procedure AddItem(AQuantity: Integer; ADescription: WideString;
      AValue: Integer);
    procedure Optimize;
    function NextBin: Boolean;
    function NextItem(var ADescription: WideString;
      var AValue: Integer): Boolean;
  end;
  IOneDBin2 = interface(IOneBin)
    ['{15B382F0-F0E0-11D2-908E-0040F6741DE2}']
    function GetName: WideString;
    function NumBins: Integer;
    function PercentWaste: Double;
  end;
const
  Class_NextFit: TGUID = '{7856B7E2-EF75-11D2-B3AB-0040F67455FE}';
  Class_FirstFit: TGUID = '{E97BFEF7-F664-11D2-9099-0040F6741DE2}';
  Class_BestFit: TGUID = '{E97BFEF8-F664-11D2-9099-0040F6741DE2}';
implementation
end.
```

Listado 2.10: Bin1Srv2 - *BinBase.pas*

```
unit BinBase;
interface
```

```
uses
  Windows, ActiveX, ComObj, Classes, BinIntf;
type
  TBinItem = class
  protected
    FDescription: string;
    FValue: Integer;
  public
    property Description: string read FDescription;
    property Value: Integer read FValue;
  end;
  TBin = class
  protected
    FValue: Integer;
    FItems: TList;
  public
    constructor Create;
    destructor Destroy; override;
    property Items: TList read FItems;
    property Value: Integer read FValue write FValue;
  end;
  TAbstractOneDBin = class(TComObject, IOneDBin, IOneDBin2)
  protected
    FMaxValue: Integer;
    FItems: TList;
    FBins: TList;
    FCurrentBin: TBin;
    FBinIndex: Integer;
    FItemIndex: Integer;
  public
    procedure Initialize; override;
    destructor Destroy; override;
    {Declare IOneDBin methods here}
    procedure SetMaxValue(AMaxValue: Integer);
    procedure AddItem(AQuantity: Integer; ADescription: WideString;
      AValue: Integer);
    procedure Optimize; virtual; abstract;
    function GetName: WideString; virtual; abstract;
    function NextBin: Boolean;
    function NextItem(var ADescription: WideString;
      var AValue: Integer): Boolean;
    function NumBins: Integer;
    function PercentWaste: Double;
  end;
implementation
uses ComServ;
{ TBin }
constructor TBin.Create;
begin
  FItems := TList.Create;
end;
destructor TBin.Destroy;
```

```
begin
  FItems.Free;
end;
{ TAbstractOneDBin }
procedure TAbstractOneDBin.AddItem(AQuantity: Integer;
  ADescription: WideString; AValue: Integer);
var
  Item: TBinItem;
  Index: Integer;
begin
  for Index := 1 to AQuantity do begin
    Item := TBinItem.Create;
    Item.FDescription := ADescription;
    Item.FValue := AValue;
    FItems.Add(Item);
  end;
end;
destructor TAbstractOneDBin.Destroy;
var
  Index: Integer;
begin
  for Index := 0 to FItems.Count - 1 do
    TBinItem(FItems[Index]).Free;
  FItems.Free;
  for Index := 0 to FBins.Count - 1 do
    TBin(FBins[Index]).Free;
  FBins.Free;
end;
procedure TAbstractOneDBin.Initialize;
begin
  FItems := TList.Create;
  FBins := TList.Create;
end;
function TAbstractOneDBin.NextBin: Boolean;
begin
  if FBinIndex < FBins.Count - 1 then begin
    Inc(FBinIndex);
    FCurrentBin := TBin(FBins[FBinIndex]);
    FItemIndex := -1;
    Result := True;
  end
  else
    Result := False;
end;
function TAbstractOneDBin.NextItem(var ADescription: WideString;
  var AValue: Integer): Boolean;
begin
  if FItemIndex < FCurrentBin.FItems.Count - 1 then begin
    Inc(FItemIndex);
    ADescription := TBinItem(FCurrentBin.FItems[FItemIndex]).FDescription;
    AValue := TBinItem(FCurrentBin.FItems[FItemIndex]).FValue;
    Result := True;
```

```
  end
  else
    Result := False;
end;
function TAbstractOneDBin.NumBins: Integer;
begin
  Result := FBins.Count;
end;
function TAbstractOneDBin.PercentWaste: Double;
var
  TotalWeight: Integer;
  UsedWeight: Integer;
  BinIndex: Integer;
  Bin: TBin;
  ItemIndex: Integer;
  Item: TBinItem;
begin
  TotalWeight := 0;
  UsedWeight := 0;
  for BinIndex := 0 to FBins.Count - 1 do begin
    Inc(TotalWeight, FMaxValue);
    Bin := TBin(FBins[BinIndex]);
    for ItemIndex := 0 to Bin.FItems.Count - 1 do begin
      Item := TBinItem(Bin.FItems[ItemIndex]);
      Inc(UsedWeight, Item.FValue);
    end;
  end;
  Result := (1.0 - UsedWeight / TotalWeight) * 100.0;
end;
procedure TAbstractOneDBin.SetMaxValue(AMaxValue: Integer);
begin
  FMaxValue := AMaxValue;
end;
end.
```

Listado 2.11: Bin1Srv2 - *NextFit.pas*

```
unit NextFit;
interface
uses
  Windows, ActiveX, ComObj, classes, BinIntf, BinBase;
type
  TNextFit = class(TAbstractOneDBin)
    procedure Optimize; override;
    function GetName: WideString; override;
  end;
implementation
uses
  ComServ;
{ TNextFit }
function TNextFit.GetName: WideString;
begin
```

```
  Result := 'Next Fit';
end;
procedure TNextFit.Optimize;
var
  Index: Integer;
  Item: TBinItem;
begin
  FCurrentBin := nil;
  for Index := 0 to FItems.Count - 1 do begin
    Item := TBinItem(FItems[Index]);
    if (FCurrentBin = nil)
      or (FCurrentBin.Value + Item.Value > FMaxValue) then begin
      FCurrentBin := TBin.Create;
      FBins.Add(FCurrentBin);
    end;
    FCurrentBin.Items.Add(Item);
    FCurrentBin.Value := FCurrentBin.Value + Item.Value;
  end;
  FBinIndex := -1;
end;
initialization
{$IFDEF VER100}
  TComObjectFactory.Create(ComServer, TNextFit, Class_NextFit,
    'NextFit', 'Next Fit algorithm', ciMultiInstance);
{$ELSE}
  TComObjectFactory.Create(ComServer, TNextFit, Class_NextFit,
    'NextFit', 'Next Fit algorithm', ciMultiInstance, tmApartment);
{$ENDIF}
end.
```

Listado 2.12: Bin1Srv2 - *FirstFit.pas*

```
unit FirstFit;
interface
uses
  Windows, ActiveX, ComObj, Classes, BinIntf, BinBase;
type
  TFirstFit = class(TAbstractOneDBin)
    procedure Optimize; override;
    function GetName: WideString; override;
  end;
implementation
uses
  ComServ;
{ TFirstFit }
function TFirstFit.GetName: WideString;
begin
  Result := 'First Fit';
end;
procedure TFirstFit.Optimize;
var
  Index: Integer;
```

```
    Item: TBinItem;
    BinFound: Boolean;
    BinIndex: Integer;
  begin
    FCurrentBin := nil;
    for Index := 0 to FItems.Count - 1 do begin
      Item := TBinItem(FItems[Index]);
      // Encontrar un almacén con espacio suficiente
      BinFound := False;
      BinIndex := 0;
      while (not BinFound) and (BinIndex < FBins.Count) do begin
        FCurrentBin := TBin(FBins[BinIndex]);
        if FCurrentBin.Value + Item.Value <= FMaxValue then
          BinFound := True
        else
          Inc(BinIndex);
      end;
      // Si no hay un almacén disponible, crear uno nuevo
      if not BinFound then begin
        FCurrentBin := TBin.Create;
        FBins.Add(FCurrentBin);
      end;
      FCurrentBin.Items.Add(Item);
      FCurrentBin.Value := FCurrentBin.Value + Item.Value;
    end;
    FBinIndex := -1;
  end;
  initialization
  {$IFDEF VER100}
    TComObjectFactory.Create(ComServer, TFirstFit, Class_FirstFit,
      'FirstFit', 'First Fit algorithm', ciMultiInstance);
  {$ELSE}
    TComObjectFactory.Create(ComServer, TFirstFit, Class_FirstFit,
      'FirstFit', 'First Fit algorithm', ciMultiInstance, tmApartment);
  {$ENDIF}
  end.
```

Listado 2.13: Bin1Srv2 - *BestFit.pas*

```
unit BestFit;
interface
uses
  Windows, ActiveX, ComObj, classes, BinIntf, BinBase;
type
  TBestFit = class(TAbstractOneDBin)
    procedure Optimize; override;
    function GetName: WideString; override;
  end;
implementation
uses
  ComServ;
{ TBestFit }
```

```
function TBestFit.GetName: WideString;
begin
  Result := 'Best Fit';
end;
procedure TBestFit.Optimize;
var
  Index: Integer;
  Item: TBinItem;
  BinIndex: Integer;
  BestBinIndex: Integer;
  BestValue: Integer;
begin
  FCurrentBin := nil;
  for Index := 0 to FItems.Count - 1 do begin
    Item := TBinItem(FItems[Index]);
    // Encontrar un almacén con espacio suficiente
    BestBinIndex := -1;
    BestValue := 0;
    for BinIndex := 0 to FBins.Count - 1 do begin
      FCurrentBin := TBin(FBins[BinIndex]);
      if FCurrentBin.Value + Item.Value <= FMaxValue then begin
        if (BestBinIndex = -1) or
           (FCurrentBin.Value > BestValue) then begin
          BestBinIndex := BinIndex;
          BestValue := FCurrentBin.Value;
        end;
      end;
    end;
    // Si no hay un almacén disponible, crear uno nuevo
    if BestBinIndex = -1 then begin
      FCurrentBin := TBin.Create;
      FBins.Add(FCurrentBin);
    end;
    FCurrentBin.Items.Add(Item);
    FCurrentBin.Value := FCurrentBin.Value + Item.Value;
  end;
  FBinIndex := -1;
end;
initialization
{$IFDEF VER100}
  TComObjectFactory.Create(ComServer, TBestFit, Class_BestFit,
    'BestFit', 'Best Fit algorithm', ciMultiInstance);
{$ELSE}
  TComObjectFactory.Create(ComServer, TBestFit, Class_BestFit,
    'BestFit', 'Best Fit algorithm', ciMultiInstance, tmApartment);
{$ENDIF}
end.
```

Actualizando el Cliente

Hemos hecho algunos ajustes menores a la aplicación cliente para ejecutar *TNextFit* y *TFirstFit*, y mostrar los resultados del algoritmo que produzca los mejores resultados. Los Listados 2.14 y 2.15 muestran el código fuente de la aplicación cliente actualizada.

Listado 2.14: BinCli2 - *BinCli.dpr*

```
program Bin1Cli;
uses
  Forms,
  MainForm in 'MainForm.pas' {Form1},
  BinIntf in '..\Bin1Srv2\BinIntf.pas';
{$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Listado 2.15: BinCli2 - *MainForm.pas*

```
unit MainForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, ComObj, StdCtrls, ComCtrls, BinIntf;
const
  MAXITEMS = 10;
type
  TForm1 = class(TForm)
    grpItems: TGroupBox;
    Label4: TLabel;
    ecQuantity: TEdit;
    Label5: TLabel;
    ecDescription: TEdit;
    Label6: TLabel;
    ecValue: TEdit;
    btnAdd: TButton;
    btnOptimize: TButton;
    treeBins: TTreeView;
    Label1: TLabel;
    ecBinSize: TEdit;
    lblAlgorithm: TLabel;
    lblWaste: TLabel;
    procedure btnAddClick(Sender: TObject);
    procedure btnOptimizeClick(Sender: TObject);
  private
    { Private declarations }
    FQty: array [1 .. MAXITEMS] of Integer;
    FDesc: array [1 .. MAXITEMS] of string;
    FValue: array [1 .. MAXITEMS] of Integer;
```

```
    FItemCount: Integer;
    FBestIntf: IOneDBin2;
    FBestPercentWaste: Double;
    procedure TestPackingMethod(MethodID: TGUID);
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.btnAddClick(Sender: TObject);
begin
  if FItemCount = MAXITEMS then begin
    ShowMessage('This demo accepts only ' + IntToStr(MAXITEMS) + '
items.');
    Exit;
  end;
  Inc(FItemCount);
  FQty[FItemCount] := StrToInt(ecQuantity.Text);
  FDesc[FItemCount] := ecDescription.Text;
  FValue[FItemCount] := StrToInt(ecValue.Text);
  Beep;
  ActiveControl := ecQuantity;
end;
procedure TForm1.TestPackingMethod(MethodID: TGUID);
var
  TestIntf: IOneDBin2;
  Index: Integer;
begin
  TestIntf := CreateComObject(MethodID) as IOneDBin2;
  TestIntf.SetMaxValue(StrToInt(ecBinSize.Text));
  for Index := 1 to FItemCount do
    TestIntf.AddItem(FQty[Index], FDesc[Index], FValue[Index]);
  TestIntf.Optimize;
  if TestIntf.PercentWaste < FBestPercentWaste then begin
    FBestPercentWaste := TestIntf.PercentWaste;
    FBestIntf := TestIntf;
  end;
end;
procedure TForm1.btnOptimizeClick(Sender: TObject);
var
  BinCount: Integer;
  Node: TTreeNode;
  Desc: WideString;
  Value: Integer;
begin
  FBestPercentWaste := 100.0;
  TestPackingMethod(Class_NextFit);
  TestPackingMethod(Class_FirstFit);
  TestPackingMethod(Class_BestFit);
  BinCount := 0;
```

```
    lblAlgorithm.Caption := FBestIntf.GetName;
    lblWaste.Caption :=
      FloatToStrF(FBestIntf.PercentWaste, ffFixed, 8, 2) + '% waste';
    treeBins.Items.BeginUpdate;
    try
      while FBestIntf.NextBin do begin
        Inc(BinCount);
        Node := treeBins.Items.AddChild(nil, 'Bin ' +
          IntToStr(BinCount));
        while FBestIntf.NextItem(Desc, Value) do
          treeBins.Items.AddChild(Node, Desc);
      end;
    finally
      treeBins.Items.EndUpdate;
    end;
    btnAdd.Enabled := False;
    btnOptimize.Enabled := False;
  end;
  end.
```

Como podemos observar en el Listado 2.15, el procedimiento *btnOptimizeClick* crea una instancia de cada servidor COM y ejecuta su método *Optimize*. Para cargar el control de vista en árbol se utilizan los datos producidos por el servidor COM que nos indique la menor cantidad de espacio inutilizado.

La Figura 2.7 muestra la nueva versión de *Bin1Cli* en tiempo de ejecución.

Figura 2.7: Aplicación Bin1Cli modificada.

La aplicación cliente modificada opera en gran medida de modo similar a la original. Sin embargo, la lista de productos aún no procesados se almacena en un *array* hasta que pulse el botón *Optimize*. En ese momento, el programa conecta secuencialmente con cada servidor COM (*NextFit*, *FirstFit*, *BestFit*), optimizando la

distribución de los productos. El control de vista en árbol será rellenado con los datos generados por el servidor que produzca la menor cantidad de espacio inutilizado.

Tenga en cuenta que existen numerosos algoritmos de distribución unidimensional, tales como *WorstFit*, *LastFit* y otros. No cansaremos al lector con el tedio de implementar todos los algoritmos disponibles. Simplemente pretendíamos mostrar cómo un servidor COM puede ser mejorado después de haber salido al mercado.

Recapitulación del ejemplo

Dado que ésta ha sido una sección larga, recapitulemos los pasos que hemos dado y los conceptos que hemos reforzado a través de este ejercicio.

1. Primero, hemos creado una aplicación servidora.
2. Después de crear el servidor, hemos creado un cliente para acceder a él.
3. A continuación, hemos mejorado el servidor proporcionando soporte para diferentes algoritmos (*FirstFit* y *BestFit*), e introducido los nuevos métodos *PercentWaste* y *NumBins*.
4. Finalmente, hemos actualizado la aplicación cliente para que ejecute los tres algoritmos de distribución implementados y determine cuál es el mejor para un conjunto de datos dado.

Los siguientes conceptos fueron reforzados durante la implementación del paso 3:

- La interfaz *IOneDBin* estaba ya en funcionamiento, por lo que no podríamos modificarla sin alterar la aplicación cliente existente.
- La interfaz *IOneDBin2* hereda de *IOneDBin*. Necesita un nuevo GUID, ya que es una nueva interfaz, pero no tenemos que redeclarar los métodos ya declarados en *IOneDBin*.
- Aún cuando la estructura de los ficheros fuentes cambiara (por ejemplo, hemos creado la nueva clase *TAbstractOneDBin*), mientras que la nueva arquitectura implemente completamente las interfaces definidas, las aplicaciones cliente existentes continuarán funcionando correctamente.
- Hemos implementado las interfaces *IOneDBin* e *IOneDBin2* en la clase abstracta *TAbstractOneDBin*, sin tener que implementar dos veces los métodos comunes a ambas interfaces. Las interfaces *IOneDBin* e *IOneDBin2* soportan el método *Optimize*, pero ambas mapean el método a la misma implementación de *Optimize* en la clase *TAbstractOneDBin*.

Dejando atrás los servidores internos al proceso, la próxima sección examina los servidores COM externos al proceso.

Servidores COM externos al proceso

Los servidores COM externos al proceso son implementados en un EXE, en lugar de una DLL. Consecuentemente, se ejecutan en un espacio de direcciones diferente que la aplicación cliente. Esto tiene algunas implicaciones, que examinaremos posteriormente en detalle.

Los servidores COM externos al proceso no exportan las cuatro funciones requeridas por un servidor COM interno al proceso. Por consiguiente, los dos tipos de servidores usan diferentes métodos para registrarse en el Registro de Windows. Para registrar un servidor externo al proceso, simplemente ejecute el servidor, añadiendo */regserver* a la línea de comandos. Delphi registrará el servidor y los objetos COM, y luego terminará. Para desregistrar el servidor, utilice el parámetro de línea de comandos */unregserver*.

Delphi también registrará el servidor si lo ejecutamos normalmente, sin opciones de línea de comandos. Sin embargo, la aplicación servidora continuará ejecutándose.

Instanciación

Los servidores COM externos al proceso pueden soportar uno de los tres métodos de *instanciación* diferentes. La instanciación tiene que ver con la cantidad de instancias de objetos COM que se crean para satisfacer las peticiones de clientes.

- *Single Instance* (única instancia) significa que se permite sólo una instancia del objeto COM por aplicación. Cada aplicación que solicite una instancia del objeto COM provocará la creación de una nueva copia del servidor COM.
- *Multiple Instance* (múltiples instancias) significa que el servidor COM es capaz de crear múltiples copias de un objeto COM. Cuando un cliente solicite una instancia de un objeto COM, no se arrancará una nueva copia del servidor COM (a menos que éste no estuviera ejecutándose). En vez de eso, el servidor que se está ejecutando creará una nueva instancia del objeto COM.
- *Internal Only* (sólo interno) es usado sólo para objetos COM que no van a ponerse a disposición de las aplicaciones cliente. La única aplicación que podrá crear el objeto COM es el servidor COM en el cual éste reside.

En general, necesitaremos crear servidores COM que soporten instanciación múltiple. Por ejemplo, suponga que hemos escrito un servidor COM que controla el acceso a un puerto serie. Dos aplicaciones cliente que están ejecutándose al mismo tiempo necesitan enviar datos al puerto (a través del servidor COM). Un servidor COM que soporte instanciación múltiple puede abrir el puerto y dar servicio a ambas aplicaciones cliente simultáneamente.

Si el servidor COM fuera un servidor de instancia única, el primer cliente en solicitar acceso al servidor provoca que se inicie una instancia del mismo. Cuando

el segundo cliente solicitara acceso al servidor, una nueva copia del proceso servidor sería lanzada para darle servicio. Debido a que la primera instancia del servidor tiene el puerto serie abierto, la segunda instancia del servidor no sería capaz de acceder al puerto.

Por supuesto, hay casos en los cuales éste podría ser el efecto deseado. En tales casos, cree servidores COM de instancia única. Sin embargo, en el caso general, a menos que se tengan razones para lo contrario, cree servidores de instancias múltiples. Como veremos en la próxima sección, cuando creamos un servidor COM usando el Asistente para Objetos COM de Delphi, éste asume por defecto que crearemos un servidor con instanciación múltiple.

Nota

No necesariamente estaremos comprometidos con un método de instanciación una vez lo hayamos seleccionado. Después de haber creado el servidor, podemos fácilmente cambiar el método de instanciación alterando una simple línea de código.

Creando una instancia de un objeto COM externo al proceso

El método para crear un objeto COM que reside dentro de un servidor externo al proceso es el mismo que para crear uno que reside dentro de un servidor interno al proceso. Podemos llamar a *CreateComObject*, pasando el GUID del objeto COM que deseamos crear.

Delphi lleva a cabo un procesamiento diferente en este caso. En lugar de usar la clave de registro *InProcServer32*, utiliza la clave *LocalServer32* para obtener la ruta completa del servidor EXE. A continuación, *CreateComObject* inicia la aplicación servidora. Como parte de la llamada a *Application.Initialization*, el servidor registra sus fábricas de clases dentro de la *tabla de objetos en ejecución* interna de Windows. La tabla de objetos en ejecución controla los objetos COM activos. Examinaremos con algo más de detalle la tabla de objetos en ejecución en el Capítulo 4.

No hemos mostrado cómo crear un servidor COM externo al proceso en este capítulo, ya que para usar servidores COM externos al proceso eficientemente necesitamos crear bibliotecas de tipos. Las bibliotecas de tipos se analizan en el Capítulo 3. En el Capítulo 4 presentaremos algunos ejemplos concretos de servidores COM externos al proceso.

Serialización de datos (marshaling)

Cuando una aplicación hace uso de un servidor COM externo al proceso, ésta es cargada en una dirección de memoria y el servidor COM es cargado en otra diferente. Las variables declaradas en la aplicación son cargadas en una dirección de memoria dada, que representa una dirección en memoria virtual**.** Por ejemplo,

supongamos que la aplicación cliente declara una variable entera *MyInt*, cuya dirección de memoria es *$00442830*. El servidor COM externo al proceso no tiene acceso a esa dirección de memoria.

Debido a que un programa ejecutable no tiene acceso directo al espacio de direcciones de otro ejecutable, Windows transfiere los datos entre la aplicación lanzada y el servidor externo al proceso a través de un mecanismo de serialización conocido como *marshaling*. Windows puede transferir automáticamente los siguientes tipos de datos de Delphi: *Smallint*, *Integer, Single*, *Double*, *Currency*, *TDateTime*, *WideString*, *IDispatch*, *SCODE*, *WordBool*, *OleVariant*, *IUnknown*, *Shortint*, *Byte*, *Word*, *UINT*, *int64*, *Largeuint*, *SYSINT*, *SYSUINT*, *HResult*, *Pointer*, *SafeArray*, *PChar* y *PWideChar*.

De los tipos anteriores, *Smallint*, *Integer*, *Single*, *Double*, *Currency*, *TDateTime*, *WideString*, *IDispatch*, *SCODE*, *WordBool*, *OleVariant*, *IUnknown*, *Shortint* y *Byte* son compatibles con la automatización COM, lo que significa que pueden ser usados de forma segura en los servidores de automatización COM. Los servidores de automatización serán examinados en detalle en el Capítulo 4.

Observe que los registros (**record**) y *arrays* están ausentes de la lista anterior. Windows no puede serializar automáticamente variables de esos tipos. ¿Y si tenemos necesidad de pasar a servidor un registro o *array*? Algo que podemos hacer es proporcionar de forma manual un soporte para transmitir la estructura en cuestión. Para ello implementaríamos la interfaz *IMarshal* para nuestros datos. La implementación de *IMarshal* es un tópico avanzado que está fuera del alcance de este libro. Si necesita más información sobre la interfaz *IMarshal*, quizás la mejor referencia sea Microsoft Developer's Network (MSDN).

En el Capítulo 6, "DCOM", desarrollaremos un ejemplo que nos muestra cómo utilizar *arrays* variantes para transmitir registros a través de las fronteras de procesos. Los *arrays* variantes son examinados en la siguiente sección.

Arrays variantes

Antes de abordar el tema de los servidores COM externos al proceso, tomaremos algún tiempo para introducir otro elemento del lenguaje llamado *arrays variantes*. Una *variante* es una variable que puede almacenar un valor de varios tipos de datos diferentes. Normalmente cuando definimos una variable, especificamos exactamente el tipo de dato que será almacenado en la misma, como en el siguiente caso:

```
var
  I: Integer;
```

La declaración de la variable *I* especifica que almacenará un valor entero. No importa cuánto lo intentemos: no podremos almacenar en la variable *I* un número de punto flotante o una cadena de caracteres.

Una variante, por otra parte, puede almacenar diferentes tipos de datos. Más aún, puede indicarnos qué tipo de datos almacena en un momento dado. Una variante ocupa 16 bytes de memoria. Podríamos pensar en una variante como en una entidad similar a un registro variante (**record case**) del Pascal tradicional, aunque las variantes ofrecen ventajas adicionales, tales como la capacidad de forzar la conversión de un valor a un tipo de datos diferente**.**

Podemos determinar el tipo de datos almacenado en una variante usando la función *VarType*. La unidad *System* define los tipos legales de variante:

```
{ Variant type codes (wtypes.h) }
const
  varEmpty    = $0000; { vt_empty       }
  varNull     = $0001; { vt_null        }
  varSmallint = $0002; { vt_i2          }
  varInteger  = $0003; { vt_i4          }
  varSingle   = $0004; { vt_r4          }
  varDouble   = $0005; { vt_r8          }
  varCurrency = $0006; { vt_cy          }
  varDate     = $0007; { vt_date        }
  varOleStr   = $0008; { vt_bstr        }
  varDispatch = $0009; { vt_dispatch    }
  varError    = $000A; { vt_error       }
  varBoolean  = $000B; { vt_bool        }
  varVariant  = $000C; { vt_variant     }
  varUnknown  = $000D; { vt_unknown     }
                       { vt_decimal $e  }
                       { undefined  $f  }
                       { vt_i1      $10 }
  varByte     = $0011; { vt_ui1         }
                       { vt_ui2     $12 }
                       { vt_ui4     $13 }
                       { vt_i8      $14 }
  { if adding new items, update varLast, BaseTypeMap and OpTypeMap }
  varStrArg   = $0048; { vt_clsid    }
  varString   = $0100; { Pascal string; not OLE compatible }
  varAny      = $0101;
  varTypeMask = $0FFF;
  varArray    = $2000;
  varByRef    = $4000;
```

El valor *varArray* es usado para indicar que la variante almacena un *array* (un *array variante*). El valor *varByRef* indica que la variante contiene un puntero al valor – el verdadero valor no se almacena directamente en la variante.

Para determinar el tipo de dato almacenado en una variante, podemos utilizar el siguiente código:

```
VType := VarType(V) and varTypeMask;
```

Realizar la operación lógica **and** entre el tipo y la constante *varTypeMask* elimina las banderas *varArray* y *varByRef*, si estuviesen presentes. Una variante que almacene un *array* de valores de tipo *Double* tendría un tipo de *$2005* (*varDouble or varArray*). Una variante que almacene un *Double* tendría un tipo de *$0005* (*varDouble*).

```
var
  V: Variant;
begin
  V := 'I love Delphi';
  V := 5;
  if VarType(V) and varTypeMask = varInteger then
    V := V + 10;
  ShowMessage(IntToStr(V));
  V := 3.5;
end;
```

Observe que podemos asignar valores de diferentes tipos de datos a una misma variante. Además, observe que se pueden realizar operaciones aritméticas sobre variantes, siempre que almacenen valores numéricos. Si tratamos de sumar un número con una variante no numérica (como por ejemplo una cadena), recibiremos en tiempo de ejecución el mensaje de error que muestra la Figura 2.8:

Figura 2.8: No trate de sumar un número a una variante no numérica.

Debemos puntualizar que es posible sumar dos variantes cuando una contenga una cadena y la otra un número. Por ejemplo, el siguiente código se ejecutará satisfactoriamente:

```
var
  V1: Variant;
  V2: Variant;
begin
  V1 := '123'; // cadena
  V2 := 5;     // entero
  ShowMessage(IntToStr(V1 + V2));
end;
```

En este ejemplo, *V1* será convertida automáticamente a entero antes de ser adicionada a *V2*. Sin embargo, si el valor almacenado en *V1* no pudiera ser convertido a un valor numérico, recibiremos el mismo mensaje de error mostrado en la Figura 2.8.

Ahora que ya conocemos qué es una variante, es fácil deducir qué es un *array variante*: es simplemente un *array* de variantes. Los *arrays* variantes no tienen por qué contener datos homogéneos. Cada elemento en el *array* puede contener un tipo de datos diferente. Haremos uso de este hecho en el Capítulo 6, pero por ahora demostraremos cómo crear *arrays* variantes.

Creando arrays variantes

Hay dos formas básicas de crear un *array* variante. Examinaremos inicialmente el método más flexible, que tiene que ver con la utilización de la función *VarArrayCreate*.

El segundo método, *VarArrayOf*, debe ser resuelto en tiempo de compilación, por lo que es mucho menos flexible.

VarArrayCreate

La forma básica de crear un *array* variante es a través de la función *VarArrayCreate*. *VarArrayCreate* está definida en *System.pas* de la manera siguiente:

```
function VarArrayCreate(const Bounds: array of Integer;
  varType: Integer): Variant;
```

El primer parámetro define los límites inferior y superior del *array*. *VarType*, por su parte, determina qué tipo de datos será almacenado en el *array*. Para crear un *array* de enteros, podríamos escribir el siguiente código:

```
MyIntArray := VarArrayCreate([1, 10], varInteger);
```

Esta instrucción crea un *array* unidimensional, con una capacidad para almacenar 10 enteros. Si deseamos crear un *array* bidimensional, podemos escribir algo como esto:

```
MyIntArray := VarArrayCreate([1, 10, 1, 5], varInteger);
```

Esta instrucción produce un *array* de 10 "columnas" y 5 "filas".

Uno de los usos más comunes del *array* variante es para crear un *array* de variantes, de la siguiente forma:

```
MyVarArray := VarArrayCreate([1, 10], varVariant);
```

Aquí se crea un *array* de 10 elementos, cada uno de los cuales es una variante. Cualquier elemento individual del *array* puede a su vez contener un *array*:

```
MyIntArray[1] := VarArrayCreate([1, 5], varVariant);
```

Por esta vía podemos crear un *array* de *arrays*, con el que podríamos modelar prácticamente cualquier estructura de datos, tales como listas, árboles, etc.

VarArrayOf

Como la creación de un *array* unidimensional es tan común, Delphi proporciona la función *VarArrayOf*, que puede ser usada para crear un *array* variante unidimensional. *VarArrayOf* está definida en *System.pas* de la siguiente forma:

```
function VarArrayOf(const Values: array of Variant): Variant;
```

Podemos usar *VarArrayOf* de esta forma:

```
MyVarArray := VarArrayOf([10, 100, 1000, 10000]);
```

Este código construye un *array* de 4 elementos que contiene los números 10, 100, 1000 y 10000.

Algo que debemos recordar es que internamente *VarArrayOf* crea un *array* variante. Por ejemplo, no podemos usar la función *VarArrayOf* para crear un *array* de enteros. Sin embargo, podemos crear un *array* de valores de diferentes tipos:

```
MyVarArray := VarArrayOf(['John', 'Smith', 4', 80000.00, 'Main Street',
                          'West Palm-Beach', 'FL']);
```

Si está pensando que el código anterior se asemeja a la definición de un registro, está ciertamente en el camino correcto. De hecho, el código de la VCL para la gestión de bases de datos hace un buen uso de los *array* variantes en algunos casos.

Accediendo a un array variante

Después de crear un *array* variante, necesitaremos acceder a sus elementos individuales. El mecanismo de acceso a los elementos de un *array* variante es muy similar al de los *arrays* estándar de Delphi. Consideremos la siguiente declaración:

```
MyIntArray := VarArrayCreate([1, 10, 1, 5], varInteger);
```

Podemos acceder a un elemento del *array* mediante la notación:

```
I := MyIntArray[1, 1];
```

De la misma forma que podemos usar las funciones *Low* y *High* para calcular los límites de un *array* estándar en Delphi, podemos usar *VarArrayLowBound* y *VarArrayHighBound* para calcular los límites de un *array* variante. *VarArrayDimCount* puede ser utilizada para calcular el número de dimensiones de un *array*:

```
procedure TraverseArray(V: Variant);
var
  Index: Integer;
begin
  ShowMessage('Array has '+ IntToStr(VarArrayDimCount(V)+ ' dimensions');
  for Index := VarArrayLowBound(V) to VarArrayHighBound(V) do
    ShowMessage(V[Index]);
end;
procedure TForm1.Button1Click(Sender: TObject);
var
  MyArray: Variant;
begin
```

```
  MyArray := VarArrayOf['Jones', 'Smith', 'Adams'];
  TraverseArray(MyArray);
end;
```

Mejorando el rendimiento de los *arrays* variantes

Como hemos mencionado anteriormente, el uso de *arrays* variantes provoca una penalización del rendimiento. Si necesitamos realizar un número determinado de accesos a un *array* variante (por ejemplo, si recorremos los elementos del *array* en un bucle), podríamos usar *VarArrayLock* y *VarArrayUnlock* para acceder de una forma rápida al *array*.

VarArrayLock devuelve un *array* de Delphi con las mismas características del *array* variante pasado como parámetro. Por esta razón, el *array* variante debe haber sido declarado como un *array* de elementos de alguno de los tipos de datos estándar de Delphi. En particular, no es posible usar *VarArrayLock* y *VarArrayUnlock* sobre *arrays* de variantes. Esto significa que no podemos hacer uso de ellas sobre un *array* creado mediante una llamada a *VarArrayOf*.

```
procedure TraverseArray2(V: Variant);
type
  PIntArray = ^TIntArray;
  TIntArray = array [1..100] of Integer;
var
  Index: Integer;
  PI: PIntArray;
begin
  ShowMessage('Array has '+ IntToStr(VarArrayDimCount(V))+ ' dimensions');
 PI := VarArrayLock(V);
  try
    for Index := VarArrayLowBound(V) to VarArrayHighBound(V) do
      ShowMessage(IntToStr(PI^[Index]));
  finally
    VarArrayUnlock(V);
  end;
end;
procedure TForm1.Button1Click(Sender: TObject);
var
  MyArray: Variant;
begin
  MyArray := VarArrayCreate([1, 3], varInteger);
  MyArray[1] := 100;
  MyArray[2] := 200;
  MyArray[3] := 300;
  TraverseArray2(MyArray);
end;
```

Observe que debemos tener presente llamar a *VarArrayUnlock* después de haber llamado a *VarArrayLock*. Para ello lo más conveniente es utilizar un bloque **try-finally** que asegura que *VarArrayUnlock* será llamada.

Ejemplo: *Array* variante

Construyamos un programa de ejemplo que acceda a un *array* variante a través de las tres formas descritas. El ejemplo además nos permitirá medir el rendimiento de las primitivas de manipulación del *array*.

Los Listados 2.16 y 2.17 presentan el código fuente de la aplicación *VarDemo*.

Listado 2.16: VarDemo - *VarDemo.dpr*

```
program VarDemo;
uses
  Forms,
  MainForm in 'MainForm.pas' {Form1};
{$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Listado 2.17: VarDemo - *MainForm.pas*

```
unit MainForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ExtCtrls;
type
  TForm1 = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Label6: TLabel;
    Label7: TLabel;
    Label8: TLabel;
    Label9: TLabel;
    lblArrayCreate: TLabel;
    lblVariantCreate: TLabel;
    lblLockedVariantCreate: TLabel;
    lblLockedVariantLock: TLabel;
    lblArrayWrite: TLabel;
    lblVariantWrite: TLabel;
    lblLockedVariantWrite: TLabel;
    lblArrayRead: TLabel;
    lblVariantRead: TLabel;
    lblLockedVariantRead: TLabel;
    lblLockedVariantUnlock: TLabel;
    lblArrayDestroy: TLabel;
    lblVariantDestroy: TLabel;
```

```
    lblLockedVariantDestroy: TLabel;
    btnRun: TButton;
    Bevel1: TBevel;
    Bevel2: TBevel;
    Bevel3: TBevel;
    Label10: TLabel;
    ecArraySize: TEdit;
    procedure btnRunClick(Sender: TObject);
  private
    FArraySize: Integer;
    procedure TestArray;
    procedure TestLockedVariant;
    procedure TestVariant;
    { Private declarations }
  public
    { Public declarations }
  end;
  TTestArray = array [1 .. 1000000] of Integer;
  PTestArray = ^TTestArray;
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.TestArray;
var
  T1, T2: DWord;
  A: PTestArray;
  Index: Integer;
  I: Integer;
  Count: Integer;
begin
  T1 := GetTickCount;
  GetMem(A, FArraySize * sizeof(Integer));
  T2 := GetTickCount;
  lblArrayCreate.Caption := IntToStr(T2 - T1);
  T1 := GetTickCount;
  for Count := 1 to 10000 do
    for Index := 1 to FArraySize do
      A^[Index] := Index;
  T2 := GetTickCount;
  lblArrayWrite.Caption := IntToStr(T2 - T1);
  T1 := GetTickCount;
  for Count := 1 to 10000 do
    for Index := 1 to FArraySize do
      I := A^[Index];
  T2 := GetTickCount;
  lblArrayRead.Caption := IntToStr(T2 - T1);
  T1 := GetTickCount;
  FreeMem(A, FArraySize * sizeof(Integer));
  T2 := GetTickCount;
  lblArrayDestroy.Caption := IntToStr(T2 - T1);
end;
```

```
procedure TForm1.TestVariant;
var
  T1, T2: DWord;
  Index: Integer;
  I: Integer;
  V: Variant;
  Count: Integer;
begin
  T1 := GetTickCount;
  V := VarArrayCreate([1, FArraySize], varInteger);
  T2 := GetTickCount;
  lblVariantCreate.Caption := IntToStr(T2 - T1);
  T1 := GetTickCount;
  for Count := 1 to 10000 do
    for Index := 1 to FArraySize do
      V[Index] := Index;
  T2 := GetTickCount;
  lblVariantWrite.Caption := IntToStr(T2 - T1);
  T1 := GetTickCount;
  for Count := 1 to 10000 do
    for Index := 1 to FArraySize do
      I := V[Index];
  T2 := GetTickCount;
  lblVariantRead.Caption := IntToStr(T2 - T1);
  T1 := GetTickCount;
  VarClear(V);
  T2 := GetTickCount;
  lblVariantDestroy.Caption := IntToStr(T2 - T1);
end;
procedure TForm1.TestLockedVariant;
var
  T1, T2: DWord;
  A: PTestArray;
  Index: Integer;
  I: Integer;
  V: Variant;
  Count: Integer;
begin
  T1 := GetTickCount;
  V := VarArrayCreate([1, FArraySize], varInteger);
  T2 := GetTickCount;
  lblLockedVariantCreate.Caption := IntToStr(T2 - T1);
  T1 := GetTickCount;
  A := VarArrayLock(V);
  T2 := GetTickCount;
  lblLockedVariantLock.Caption := IntToStr(T2 - T1);
  T1 := GetTickCount;
  for Count := 1 to 10000 do
    for Index := 1 to FArraySize do
      A^[Index] := Index;
  T2 := GetTickCount;
  lblLockedVariantWrite.Caption := IntToStr(T2 - T1);
```

```
  T1 := GetTickCount;
  for Count := 1 to 10000 do
    for Index := 1 to FArraySize do
      I := A^[Index];
  T2 := GetTickCount;
  lblLockedVariantRead.Caption := IntToStr(T2 - T1);
  T1 := GetTickCount;
  VarArrayUnlock(V);
  T2 := GetTickCount;
  lblLockedVariantUnlock.Caption := IntToStr(T2 - T1);
  T1 := GetTickCount;
  VarClear(V);
  T2 := GetTickCount;
  lblLockedVariantDestroy.Caption := IntToStr(T2 - T1);
end;
procedure TForm1.btnRunClick(Sender: TObject);
begin
  FArraySize := StrToInt(ecArraySize.Text);
  if (FArraySize < 2) or (FArraySize > 1000000) then begin
    ShowMessage('Please specify an array size between 2 and 1000000');
    Exit;
  end;
  Screen.Cursor := crHourglass;
  try
    TestArray;
    TestVariant;
    TestLockedVariant;
  finally
    Screen.Cursor := crDefault;
  end;
end;
end.
```

La Figura 2.9 muestra la salida después de ejecutar *VarDemo* en un ordenador Pentium II a 450 Mhz.

Figura 2.9: *VarDemo* muestra la velocidad relativa de los *arrays* de Delphi, *arrays* variantes y *array* variantes bloqueados.

Como puede verse, la interfaz de usuario de esta aplicación es sencilla. Escriba el número de elementos del *array*, y pulse el botón '*Run All Tests*' para ejecutar todas las pruebas automáticamente. No invertiremos ningún tiempo discutiendo el código fuente de esta aplicación, ya que en la sección anterior hemos explicado todos los conceptos relacionados con el ejemplo. *VarDemo* simplemente unifica todos esos conceptos.

Tomemos unos momentos para examinar los tiempos producidos por la aplicación. Ante todo, debemos observar que el tiempo requerido para crear y destruir tanto un *array* estándar de Delphi como un *array* variante es insignificante.

El acceso a un *array* variante es muy costoso, comparado con el acceso a un *array* estándar. La penalidad en el rendimiento es de cerca de *30x* en mi ordenador. Sin embargo, observe qué ocurre cuando bloqueamos el *array* variante antes de usarlo. Una vez bloqueado el *array*, los tiempos de accesos son prácticamente los mismos que para cualquier *array* estándar de Delphi.

Tenga en cuenta además que el tiempo requerido para bloquear y desbloquear el *array* variante es despreciable. Bloquear un *array* variante no crea una copia de éste; simplemente activa una bandera que impide que otros usuarios realicen modificaciones en él mientras esté protegido.

Usando objetos COM con otros lenguajes

En este punto, estaremos preguntándonos cómo los usuarios de otros lenguajes podrán acceder a los servidores COM que creemos. Todo el código es Delphi, así que, si necesitáramos una interfaz con Visual C++, por ejemplo, ¿cómo podríamos hacerlo?

Como mencionamos anteriomente en este capítulo, los objetos COM pueden ser usados por cualquier lenguaje que soporte tablas de métodos virtuales (o *vtable*, en el lenguaje COM). El problema radica en que para usar los objetos escritos, necesitaríamos recrear la definición de sus interfaces en el lenguaje específico, lo cual en general puede no ser una tarea trivial.

Adicionalmente, la serialización automática de datos requiere de la existencia de bibliotecas de tipos para identificar que parámetros son pasados a los métodos de las interfaces y cómo serializarlos. En el próximo capítulo se examinarán las bibliotecas de tipos, que son una forma neutral al lenguaje para declarar las interfaces de nuestros objetos COM.

Resumen

En este capítulo hemos aprendido qué es exactamente un GUID y por qué éstos son tan importantes en el desarrollo de aplicaciones COM. También hemos aprendido cómo crear un GUID para nuestro propio uso.

Hemos examinado en detalle los servidores internos al proceso, y hemos mostrado cómo crear y utilizar un servidor interno al proceso en Delphi.

También hemos aprendido qué es un *array* variante, por qué son útiles y cómo se utilizan. Hemos mostrado cómo se puede bloquear un *array* variante para incrementar el rendimiento de los accesos al *array*.

Posteriormente, hemos introducido los servidores COM externos al proceso, aunque no hemos creado ninguno en este capítulo. En el Capítulo 4 mostraremos cómo crear servidores COM externos al proceso a través de múltiples ejemplos. Hemos examinado el *marshaling*, porque es importante para los servidores externos al proceso, y los tipos de datos que Windows permite serializar automáticamente.

Por último, hemos destacado que sólo los lenguajes orientados a objetos que "entienden" *vtables* podrán acceder a los servidores COM creados en este capítulo.

En el próximo capítulo examinaremos las bibliotecas de tipos, que constituyen una forma independiente del lenguaje de exponer los objetos en nuestros servidores COM. Posteriormente haremos un profundo análisis de los servidores de automatización, que pueden ser utilizados prácticamente por cualquier lenguaje de programación moderno.

Resumen

En este capítulo hemos aprendido qué es exactamente un GUID y por qué éstos son tan importantes en el desarrollo de aplicaciones COM. También hemos aprendido cómo crear un GUID para nuestro propio uso.

Hemos examinado en detalle los servidores internos al proceso, y hemos mostrado cómo crear y utilizar un servidor interno al proceso en Delphi.

También hemos aprendido qué es un *array* variante, por qué son útiles y cómo se utilizan. Hemos mostrado cómo se puede bloquear un *array* variante para incrementar el rendimiento de los accesos al *array*.

Posteriormente, hemos introducido los servidores COM externos al proceso, aunque no hemos creado ninguno en este capítulo. En el Capítulo 4 mostraremos cómo crear servidores COM externos al proceso a través de múltiples ejemplos. Hemos examinado el *marshaling*, porque es importante para los servidores externos al proceso, y los tipos de datos que Windows permite serializar automáticamente.

Por último, hemos destacado que sólo los lenguajes orientados a objetos que "entienden" *vtables* podrán acceder a los servidores COM creados en este capítulo.

En el próximo capítulo examinaremos las bibliotecas de tipos, que constituyen una forma independiente del lenguaje de exponer los objetos en nuestros servidores COM. Posteriormente haremos un profundo análisis de los servidores de automatización, que pueden ser utilizados prácticamente por cualquier lenguaje de programación moderno.

Resumen

En este capítulo hemos aprendido qué es exactamente un GUID y por qué éstos son tan importantes en el desarrollo de aplicaciones COM. También hemos aprendido cómo crear un GUID para nuestro propio uso.

Hemos examinado en detalle los servidores internos al proceso, y hemos mostrado cómo crear y utilizar un servidor interno al proceso en Delphi.

También hemos aprendido qué es un *array* variante, por qué son útiles y cómo se utilizan. Hemos mostrado cómo se puede bloquear un *array* variante para incrementar el rendimiento de los accesos al *array*.

Posteriormente, hemos introducido los servidores COM externos al proceso, aunque no hemos creado ninguno en este capítulo. En el Capítulo 4 mostraremos cómo crear servidores COM externos al proceso a través de múltiples ejemplos. Hemos examinado el *marshaling*, porque es importante para los servidores externos al proceso, y los tipos de datos que Windows permite serializar automáticamente.

Por último, hemos destacado que sólo los lenguajes orientados a objetos que "entienden" *vtables* podrán acceder a los servidores COM creados en este capítulo.

En el próximo capítulo examinaremos las bibliotecas de tipos, que constituyen una forma independiente del lenguaje de exponer los objetos en nuestros servidores COM. Posteriormente haremos un profundo análisis de los servidores de automatización, que pueden ser utilizados prácticamente por cualquier lenguaje de programación moderno.

3

Bibliotecas de tipos

En el Capítulo 2, "Las Interfaces y COM", mostramos cómo crear servidores COM. Como hemos estudiado, los servidores COM creados en ese capítulo pueden ser usados solamente por lenguajes que "entiendan" las tablas de métodos virtuales, tales como Delphi y C++. No solo eso, el desarrollador de una aplicación cliente deberá obtener de algún modo una copia de las interfaces implementadas por el servidor COM para poder utilizarlo.

En los casos en que vayamos a crear tanto el servidor como el cliente en Delphi, podremos simplemente hacer referencia a la unidad en la que se declara la interfaz del servidor COM en la cláusula **uses** de la aplicación cliente. Sin embargo, en muchos casos, necesitaremos utilizar un servidor COM escrito en otro lenguaje como C o C++, o deberemos crear un servidor COM para distribuirlo a desarrolladores que no trabajen en Delphi. Existe por tanto la necesidad de compartir de alguna forma la declaración de las interfaces entre programadores que utilizan diferentes lenguajes.

En este capítulo examinamos las bibliotecas de tipos, que constituyen una forma independiente del lenguaje de definir interfaces y documentar servidores COM.

Definición de Biblioteca de Tipos

Como mencionamos en la sección anterior, las bibliotecas de tipos ofrecen un mecanismo para documentar los objetos COM. Las bibliotecas de tipos contienen datos tales como las interfaces que son implementadas por un objeto, las propiedades y métodos definidos en una interfaz, y la cantidad y tipos de argumentos para cada método.

En general, las bibliotecas de tipos pueden ser integradas en los propios servidores COM en forma de un recurso, o pueden ser distribuidas separadamente como un fichero con extensión .TLB. Delphi añade automáticamente una biblioteca de tipos como recurso a los servidores COM, y también genera un fichero .TLB.

Delphi (como la mayoría de los compiladores modernos) puede leer la información almacenada en una biblioteca de tipos y crear automáticamente un *fichero de*

importación, que podremos utilizar para desarrollar las aplicaciones clientes. Por esa razón, generalmente sólo es necesario distribuir el servidor COM al programador que va a desarrollar una aplicación cliente; en caso de que encontremos a alguien que no pueda leer la biblioteca de tipos, podemos entregarle también el fichero .TLB.

Razones para Usar Bibliotecas de Tipos

Las bibliotecas de tipos son imprescindibles para ciertos tipos de servidores COM, tales como los servidores de automatización y los controles ActiveX. Los servidores de automatización se examinan en el Capítulo 4, "Automatización", y los controles ActiveX en Capítulo 5, "Controles ActiveX y ActiveForms".

Incluso en el caso de que estemos desarrollando un servidor COM que no requiera de biblioteca de tipos, podríamos desear crear una. Las bibliotecas de tipos ofrecen múltiples ventajas, entre las que podemos incluir:

- *Enlace temprano* cuando creemos controladores de automatización (los conceptos de enlace *temprano* y *tardío* se examinan en detalle en el Capítulo 4)
- Muchos compiladores generan automáticamente código específico para un lenguaje a partir de una biblioteca de tipos.
- Las utilidades conocidas como *visores de bibliotecas de tipos* pueden ser utilizadas para leer y mostrar información de servidores COM que contienen una biblioteca de tipos.
- Serialización (*marshaling*) automática de parámetros entre clientes y servidores COM. En el Capítulo 2 hemos discutido el soporte que proporciona Windows para la serialización. Las bibliotecas de tipos permiten que Windows realice la serialización sin ningún esfuerzo especial por parte del servidor o del cliente.

TTypedComObject

Delphi proporciona soporte para las bibliotecas de tipos a través de la clase *TTypedComObject* y sus descendientes. Si recordamos el Capítulo 2, al utilizar el Asistente de Objetos COM, éste nos preguntaba si deseamos crear una biblioteca de tipos. Para su referencia, el Asistente de Objetos COM se muestra en la Figura 3.1.

Si se activa la opción '*Include Type Library*', Delphi genera un código similar al del Listado 3.1.

Nota

Recuerde que Delphi 3 no proporciona un Asistente de Objetos COM. Si es Ud. un usuario de Delphi 3, deberá escribir el código que se muestra en el Listado 3.1.

Figura 3.1: Asistente de Objetos COM

Listado 3.1: Código generado por el Asistente de Objetos COM

```
unit Unit2;
interface
uses
  Windows, ActiveX, ComObj, Project1_TLB, StdVcl;
type
  TTest = class(TTypedComObject, ITest)
  protected
    {Declare ITest methods here}
  end;
implementation
uses ComServ;
initialization
  TTypedComObjectFactory.Create(ComServer, TTest, Class_Test,
    ciMultiInstance, tmApartment);
end.
```

El código que se muestra en el Listado 3.1 es muy similar al código que Delphi genera cuando no se solicita una biblioteca de tipos. Existen tres diferencias fundamentales:

- Delphi crea automáticamente el fichero de importación *Project1_TLB.pas* que se muestra en el Listado 3.2.
- El objeto COM (en este caso *TTest*) es derivado de *TTypedComObject* en lugar de *TComObject*.
- En la sección de inicialización de la unidad, Delphi llama a *TTypedComObjectFactory.Create* en lugar de *TComObjectFactory.Create*.

He aquí la definición completa de la clase *TTypedComObject*, definida en *ComObj.pas*.

```
type
  TTypedComObject = class(TComObject, IProvideClassInfo)
  protected
```

```
    { IProvideClassInfo }
    function GetClassInfo(out TypeInfo: ITypeInfo): HResult; stdcall;
  end;
```

Como podemos ver, no es mucha la diferencia con su ancestro *TComObject*. Únicamente *TTypedComObject* implementa la interfaz *IProvideClassInfo*, que sólo define un método. La interfaz *IProvideClassInfo* simplemente proporciona un mecanismo de acceso a la interfaz *ITypeInfo* que examinaremos posteriormente en este capítulo.

Listado 3.2: Código generado por el Asistente de Objetos COM

```
unit Project1_TLB;
// ************************************************************************ //
// WARNING                                                                  //
// □□□-                                                                     //
// The types declared in this file were generated from data read from       //
// a Type Library. If this type library is explicitly or indirectly         //
// (via another type library referring to this type library)                //
// re-imported, or the 'Refresh' command of the Type Library Editor         //
// activated while editing the Type Library, the contents of this file      //
// will be regenerated and allmanual modifications will be lost.            //
// ************************************************************************ //
// PASTLWTR : $Revision:   1.11.1.75  $
// File generated on 8/10/1999 2:45:39 PM from Type Library described
below.
// ************************************************************************ //
// Type Lib: C:\Book\Chap03\Proyect1.tlb (1)
// IID\LCID: {91ADB6E6-4F4D-11D3-B84B-0040F67455FE}\0
// Helpfile:
// HelpString:  Project1 Library
// Version: 1.0
// ************************************************************************ //
interface
uses Windows, ActiveX, Classes, Graphics, OleCtrls, StdVCL;
// ************************************************************************ //
// GUIDS declared in the TypeLibrary. Following prefixes are used:          //
//   Type Libraries     : LIBID_xxxx                                        //
//   CoClasses          : CLASS_xxxx                                        //
//   DISPInterfaces     : DIID_xxxx                                         //
//   Non-DISP interfaces: IID_xxxx                                          //
// ************************************************************************ //
const
  // TypeLibrary Major and minor versions
  LIBID_DCPTypeServer: TGUID = '{91ADB6E6-4F4D-11D3-B84B-0040F67455FE}';
  IID_ITest: TGUID = '{91ADB6E7-4F4D-11D3-B84B-0040F67455FE}';
  CLASS_Test: TGUID = '{91ADB6E9-4F4D-11D3-B84B-0040F67455FE}';
type
// ************************************************************************ //
// Forward declaration of types defined in Type Library                     //
// ************************************************************************ //
```

```
  ITest = interface;
// ********************************************************************* //
// Declaration of CoClasses defined in Type Library                      //
// (NOTE: Here we map each CoClass to its Default Interface)             //
// ********************************************************************* //
  Test = ITest;
// ********************************************************************* //
// Interface: ITest
// Flags:     (0)
// GUID:      {91ADB6E7-4F4D-11D3-B84B-0040F67455FE}
// ********************************************************************* //
  ITest = interface (IUnknown)
    ['{91ADB6E7-4F4D-11D3-B84B-0040F67455FE}']
  end;
  CoTest = class
    class function Create: ITest;
    class function CreateRemote(const MachineName: string): ITest;
  end;
implementation
uses ComObj;
class function CoTest.Create: ITest;
begin
  Result := CreateComObject(CLASS_Test) as ITest;
end;
class function CoTest.CreateRemote(const MachineName: string): ITest;
begin
  Result := CreateRemoteComObject(MachineName, CLASS_Test) as ITest;
end;
end.
```

Ya conocemos qué es una biblioteca de tipos y para qué es usada. En las siguientes secciones les guiaremos a través de los pasos básicos para crear una biblioteca de tipos en Delphi.

Creando una biblioteca de tipos en Delphi

Antes de la aparición de Delphi, la mayoría de los lenguajes de programación nos obligaban a escribir las bibliotecas de tipos manualmente, utilizando un lenguaje conocido como *Interface Definition Language* (IDL). Consideremos la siguiente declaración de una interfaz:

```
type
  ITest = interface
    procedure DoSomething;
    function GetValue: Integer;
  end;
```

El código IDL para un servidor COM que implementa la interfaz *ITest* se muestra en el Listado 3.3.

Listado 3.3: Código IDL para la interfaz *ITest*

```
[
  uuid(1F415FD5-F132-11D2-B3B0-0040F67455FE),
  version(1.0),
  helpstring("Project1 Library")

library Project1
{
  importlib("STDOLE2.TLB");
  importlib("STDVCL50.TLB");
  [
    uuid(1F415FD6-F132-11D2-B3B0-0040F67455FE),
    version(1.0),
    helpstring("Interface for Test Object")

  interface ITest: IUnknown
  {
    [id(0x00000001)
    void _stdcall DoSomething ( void );
    [id(0x00000002)
    long _stdcall GetValue( void );

  [
    uuid(1F415FD8-F132-11D2-B3B0-0040F67455FE),
    version(1.0),
    helpstring('Test Object')

  coclass Test
  {
    [default  interface ITest;
   ;
 ;
```

Enmascarada en medio del código podemos distinguir la declaración de la interfaz *ITest*, que repetiremos nuevamente para facilitar la referencia.

```
interface ITest: IUnknown
{
  [id(0x00000001)]
  void _stdcall DoSomething(void):
  [id(0x00000002)]
  long _stdcall GetValue(void);
};
```

Como podemos ver del Listado 3.3, el desarrollo de aplicaciones COM (o al menos, de bibliotecas de tipos) antes de Delphi era una tarea compleja, a menos que se dominara IDL de forma fluida (¡lo cual yo no hacía!).

Nota

El código IDL anterior fue generado usando el Editor de Bibliotecas de Tipos de Delphi.

Usando el Editor de Bibliotecas de Tipos

Entra en escena el Editor de Bibliotecas de Tipos de Delphi. Una buena parte de este capítulo se centra en cómo utilizar el Editor de Bibliotecas de Tipos, que hace de la creación y edición de bibliotecas de tipos una tarea sencilla. Manteniendo su filosofía fundamental de herramientas bidireccionales, Delphi mantiene nuestro código fuente actualizado cuando creamos declaraciones de interfaces mediante el Editor de Bibliotecas de Tipos.

El Editor de Bibliotecas de Tipos ha sufrido cambios sustanciales de Delphi 3 a Delphi 4, y de éste a Delphi 5. Las ventanas mostradas en este capítulo pertenecen a Delphi 5, así que si está usando una versión anterior de Delphi, nuestras pantallas podrían no coincidir exactamente con lo que verá en su monitor. Sin embargo, aún en este caso deberá ser capaz de seguir adelante, pues puntualizaremos en las mayores diferencias entre versiones cuando sea apropiado.

Nota

Este libro no pretende ser una referencia detallada del Editor de Bibliotecas de Tipos. Explicaremos lo suficiente para adquirir los conocimientos básicos necesarios en la práctica diaria de programación. No obstante, el Editor de Bibliotecas de Tipos es una herramienta compleja. Si desea aprender más acerca de las opciones o características avanzadas, presione F1 para acceder a la ayuda, o simplemente experimente sobre un servidor COM de prueba.

Creemos un nuevo servidor COM interno al proceso mediante la opción *File | New* del menú principal. Seleccione la pestaña *ActiveX* del Depósito de Objetos, y haga doble clic sobre el icono *ActiveX Library*. Nuevamente, seleccione *File | New* en el menú principal, pero esta vez pulse dos veces sobre el icono *COM Object* de la página *ActiveX* del Depósito de Objetos.

Rellene los campos del Asistente de Objetos COM, como se muestra en la Figura 3.2:

Figura 3.2: Creación de un servidor COM con una biblioteca de tipos.

Asegúrese de marcar la opción '*Include Type Library*' (debe estar marcada por defecto). Cuando pulse el botón OK, Delphi generará la ya familiar unidad para el objeto COM, y luego lanzará automáticamente el Editor de Bibliotecas de Tipos. Si alguna vez necesita iniciar el Editor de Bibliotecas de Tipos manualmente, simplemente seleccione *View | Type Library* desde el menú principal de Delphi. La Figura 3.3 muestra el Editor de Bibliotecas de Tipos cuando es llamado por Delphi para el objeto COM *TUnitAuto* recién creado.

Figura 3.3: El Editor de Bibliotecas de Tipos.

El Editor de Bibliotecas de Tipos está dividido en tres secciones. En la parte superior de la ventana se encuentra una barra de herramientas que se usa para

añadir interfaces, métodos y propiedades al servidor COM. En el lado izquierdo de la ventana principal se encuentra una vista de árbol que muestra las interfaces, métodos, etc. que hemos definido para el servidor. Esta vista se conoce como la *lista de objetos*. La lista de objetos es conceptualmente muy similar al explorador de código de Delphi, ya que muestra la declaración de interfaces, métodos e información de otro tipo contenida en la biblioteca de tipos.

El resto del Editor de Bibliotecas de Tipos está compuesto por un control multipáginas que muestra la información asociada al nodo seleccionado en el árbol. Es extraño que el Editor de Bibliotecas de Tipos no tenga un menú principal - toda la funcionalidad se proporciona a través de la barra de herramientas y de menús contextuales, que aparecen cuando se pulsa el botón derecho del ratón sobre la lista de objetos.

Truco

Puede activar los nombres de los botones de la barra de herramientas pulsando el botón derecho del ratón sobre ésta y seleccionando la opción '*Text Labels*' del menú contextual.

Nuestra nueva biblioteca de tipos tiene tres nodos en su lista de objetos: *Project2*, *IUnitAuto* y *UnitAuto*. *Project2* representa el proyecto en el que estamos trabajando. Lo primero que haremos será renombrar el proyecto. Puede hacer clic sobre el nodo *Project2*, o pulsar el cuadro de edición *Name* en el panel de información. Cambie el nombre a *DCPTypeServer*; también cambie la cadena de ayuda por '*DCP TypeInfo Server*'.

El próximo nodo en la lista de objetos, *IUnitAuto*, se refiere a la interfaz que ha sido creada automáticamente para nosotros. Los valores que se asignan por defecto a ésta son aceptables.

El último nodo, *UnitAuto*, tiene que ver con la *CoClass* o clase derivada de *TTypedComObject* que implementa la interfaz *IUnitAuto*. Los valores por defecto para este nodo son también aceptables.

Examinaremos cada botón de la barra de herramientas en las siguientes subsecciones (la Figura 3.4 muestra la apariencia de la barra de herramientas).

Figura 3.4: Barra de Herramientas del Editor de Bibliotecas de Tipos.

La barra de herramientas del Editor de Bibliotecas de Tipos tiene la apariencia que se muestra en la Figura 3.4. En ella se destacan los botones que se analizan en las siguientes secciones.

Nota

Como estamos presentando el Editor de Bibliotecas de Tipos, vamos a crear una biblioteca de tipos que será utilizada en el primer ejemplo del Capítulo 4. La razón para crear una biblioteca de tipos legítima es que más adelante en este capítulo mostraremos cómo escribir un programa que lee e interpreta una biblioteca de tipos. Aunque ya tendrá varias bibliotecas de tipos en su ordenador (se haya dado cuenta de ello o no), no tengo forma de conocer qué está instalado en su ordenador particular; la creación de esta biblioteca nos permitirá mantenernos en la misma página.

New Interface (Nueva interfaz)

Pulse este botón para añadir una interfaz a una biblioteca de tipos. Nuestro servidor de ejemplo sólo va a soportar la interfaz *IUnitAuto*, pero podría soportar interfaces adicionales si así lo deseáramos. Delphi generará automáticamente un GUID para cada nueva interfaz que creemos.

New Dispinterface (Nueva disp-interfaz)

Una *disp-interfaz* es muy similar a una interfaz, pero utiliza un mecanismo de despacho diferente para llamar a los métodos del servidor. Examinaremos las disp-interfaces en el Capítulo 4, así que no añadiremos ninguna a nuestro servidor por ahora.

New CoClass (Nueva co-clase)

CoClass (clase COM) es el término bajo el que se nombra al objeto COM que implementa una o varias interfaces. En este ejemplo sólo estamos escribiendo una co-clase, *UnitAuto*, que Delphi ha creado automáticamente.

New Enumeration (Nueva enumeración)

Una *enumeración* en una biblioteca de tipos es un concepto análogo a un tipo enumerado en Delphi. Las enumeraciones se representan mediante identificadores enteros, en lugar de tipos conjunto. Añadamos una enumeración a nuestra biblioteca de tipos.

1. Pulse el botón '*New Enumeration*' de la barra de herramientas. Delphi crea una nueva enumeración llamada *Enum1*.
2. Como *Enum1* no es nombre muy descriptivo, cambiemos éste a *AreaUnit*. Esta enumeración contendrá unidades de medida de áreas.

Delphi asigna automáticamente un GUID y un número de versión (1.0) a la nueva enumeración. Simplemente creando una enumeración no especificamos sus

valores. Necesitamos añadir constantes a la enumeración, como se explica a continuación.

New Constant (Nueva constante)

Obviaremos algunos botones y examinaremos ahora el botón '*New Constant*'. Este botón se activa únicamente cuando está seleccionada una enumeración en la lista de objetos. Añadamos algunos valores a la enumeración *AreaUnit*.

1. Seleccione el enumerador *AreaUnit* en la lista de objetos.
2. Pulse el botón *New Constant* para añadir un valor a la enumeración *AreaUnit*.
3. Cambie el nombre de la constante por *auSquareMeters*.
4. Añada otra constante y llámela *auSquareCentimeters*.
5. Añada las constantes *auSquareYards*, *auSquareFeet*, *auSquareInches*, *auSquareKilometers*, *auSquareMiles* y *auAcres*.

En la medida en que se añaden las constantes, Delphi automáticamente asignará el valor 0 a *auSquareMeters*, 1 a *auSquareCentimeters*, y así sucesivamente.

Observe que yo prefiero utilizar prefijos de dos o tres letras en las enumeraciones (en el ejemplo, "*au*" para *Area Unit*). Esta es una preferencia estrictamente personal y ni COM, ni el Editor de Bibliotecas de Tipos ni Delphi obligan a ello.

New Alias (Nuevo alias)

Un *alias* en una biblioteca de tipos es usado para definir un tipo que deseamos incluir en un registro o unión. La mayoría de las bibliotecas de tipos no incluyen esta información, y no haremos más referencia a ella en este libro.

New Record (Nuevo registro)

Pulse este botón para definir una estructura de registro en la biblioteca de tipos. La mayoría de las bibliotecas de tipos no definen estructuras de registro, y no haremos más referencia a esto en el libro.

New Union (Nueva unión)

Pulse este botón para definir una *unión* en la biblioteca de tipos. Una unión es el equivalente a un registro variante en Pascal. La mayoría de las bibliotecas de tipos no definen uniones, y no haremos más referencia a esto en el libro.

New Module (Nuevo módulo)

Pulsemos este botón para crear un nuevo *módulo* en la biblioteca de tipos. Un módulo es un conjunto de métodos y constantes. No encontraremos definiciones de módulos en la mayoría de las bibliotecas de tipos, y no discutiremos sobre ello en el resto del libro.

New Method (Nuevo método)

Si selecciona una interfaz (o disp-interfaz) en la lista de objetos, el botón '*New Method*' de la barra de herramientas se activará. Para añadir un nuevo método a la interfaz *IUnitAuto* haremos lo siguiente:

1. Seleccione la interfaz *IUnitAuto* en la lista de objetos.
2. Pulse el botón '*New Method*' para añadir un nuevo método a la interfaz *IUnitAuto*.
3. Cambie el nombre del nuevo método a *Convert*. El Editor de Bibliotecas de Tipos automáticamente asigna al método *Convert* un identificador (*ID*) de 1. Los IDs de métodos son examinados en el Capítulo 4.
4. Seleccione la página '*Parameters*' en el panel de información. Observe que el tipo de retorno se define como *HResult* (ver Figura 3.5).
5. Seleccione *double* en el cuadro del tipo de retorno.

Figura 3.5: Parámetros por defecto del método.

Ahora vamos a añadir parámetros al método *Convert*. *Convert* aceptará tres parámetros: la cantidad que deseamos convertir, la unidad de medida actual y la unidad de medida a la cual deseamos realizar la conversión. Antes de añadir los parámetros, analicemos brevemente los *modificadores* de parámetros.

Recuerde que en el Capítulo 2 hablamos de la serialización. La serialización exige de cierto trabajo detallado por parte de Windows. Obviamente, mientras menos información tenga que ser transferida entre fronteras de proceso, la llamada al

método se ejecutará más eficientemente. Es por esta razón que debemos considerar cuidadosamente los parámetros que serán pasados al método. Algunos parámetros sólo necesitan ser enviados al servidor. El servidor no cambiará el valor del parámetro, así que éstos no necesitan ser devueltos al cliente. Si el servidor no hace ningún cambio a los parámetros, ¿para qué entonces enviarlos de vuelta al cliente? El cliente ya conoce el valor de los parámetros, después de todo, él los envió al servidor. A estos parámetros se les conoce como parámetros *in* (de entrada).

El otro lado de la moneda lo constituyen los valores que son calculados por el servidor y que sólo necesitan ser devueltos al cliente. Los valores no necesitan ser enviados del cliente al servidor, después de todo. A estos parámetros se les conoce como parámetros *out* (de salida).

Por último, tenemos los parámetros que tienen un valor del lado del cliente, y son modificados en el servidor. Estos parámetros deben ser serializados en ambas direcciones, y se conocen como parámetros *var* (de entrada y salida).

Consideremos el método *Convert*. La cantidad pasada a la función obviamente debe ser enviada del cliente al servidor. El servidor utilizará el valor pero no lo modificará, así que no tiene que enviarlo de vuelta al cliente (la cantidad será convertida a una unidad de medida diferente, pero será devuelta al cliente como resultado de la función). Por lo tanto, estamos en presencia de un parámetro *in*.

Ahora que hemos comprendido los modificadores de parámetros, añadamos los tres parámetros al método *Convert*:

1. Pulse el botón *Add* del panel de información. El Editor de Bibliotecas de Tipos creará un nuevo parámetro llamado *Param1*.
2. Pulse sobre el nombre del parámetro y cambie éste a *Quantity*.
3. Pulse sobre el tipo('*Type*') y cambie éste a **double**. El modificador tiene como valor por defecto *in*, así que podemos dejarlo como está.
4. Pulse sobre botón *Add* nuevamente, y añada un nuevo parámetro de tipo *int* y nombre *InUnit*.
5. Por último, añada otro parámetro entero llamado *OutUnit*.

Cuando finalice, el Editor de Bibliotecas de Tipos tendrá una apariencia similar a la que se muestra en la Figura 3.6.

Figura 3.6: Añadiendo un método a una Biblioteca de Tipos

New Property (Nueva propiedad)

Si pulsamos sobre una interfaz (o disp-interfaz) en el árbol, el botón '*New Property*' de la barra de herramientas se activará. Este botón se utiliza para añadir una nueva propiedad a la interfaz.

Podemos crear propiedades de sólo lectura, de sólo escritura y de lectura y escritura, tal como es posible hacerlo en las clases de Delphi.

La interfaz *IUnitAuto* que crearemos en el capítulo 4 no requiere ninguna propiedad. Sin embargo, las propiedades en COM son tan comunes que desearía mostrar cómo crear una:

1. Seleccione el nodo *IUnitAuto* en la lista de objetos.
2. Despliegue las opciones que ofrece el botón '*New Property*'. Encontrará opciones que permiten seleccionar entre una propiedad de sólo lectura, de sólo escritura o de lectura y escritura.
3. Seleccione la opción '*Read/Write*', para crear una propiedad que permita lectura y escritura. El Editor de Bibliotecas de Tipos añadirá dos propiedades bajo el nodo *IUnitAuto*, ambas llamadas *Property1*. La primera instancia de *Property1* se refiere al método de acceso de lectura (*Get*) de la propiedad, y la segunda instancia al método de acceso de escritura (*Put*).

4. Cambie el nombre de una de las instancias de la propiedad a *Name*. El Editor de Bibliotecas de Tipos automáticamente cambiará el nombre de la otra instancia.
5. Seleccione en la lista de objetos la instancia de la propiedad *Name* que corresponde al método de acceso de escritura (*Put*).
6. En la página *Parameters* del panel de información, especifique *void* (nulo) como valor de retorno.
7. También sobre la página *Parameters*, asigne el tipo del parámetro generado automáticamente como *BSTR* (si está usando Delphi 3 ó 4, utilice *WideString* en lugar de *BSTR*).
8. Seleccione en la lista de objetos la instancia de la propiedad *Name* que corresponde al método de acceso de lectura (*Get*).
9. En la página *Parameters* del panel de información, especifique *BSTR* como tipo del valor de retorno. Además tendrá que escribir la palabra BSTR, ya que no aparecerá en el cuadro de combinación. (De nuevo, los usuarios de Delphi 3 y 4 deberán usar WideString en vez de BSTR).

La Figura 3.7 muestra el estado del Editor de Bibliotecas de Tipos después de añadida la propiedad *Name*:

Figura 3.7: Añadiendo propiedades a una biblioteca de tipos.

Refresh Implementation (Actualizar implementación)

Normalmente Delphi mantiene nuestro código actualizado a partir de la información que se introduce en el Editor de Bibliotecas de Tipos. Puede pulsar este botón para obligar a Delphi a actualizar los ficheros que contienen el código fuente.

Para asegurarse de que el código fuente está sincronizado con el Editor de Bibliotecas de Tipos, haga clic ahora sobre el botón '*Refresh Implementation*'.

Register Type Library (Registrar biblioteca de tipos)

Cuando se pulsa sobre este botón, Delphi compila el servidor COM y lo registra automáticamente ante Windows. Los mismos resultados pueden obtenerse manualmente, compilando el proyecto y luego seleccionando *Run | Register ActiveX Server* en el menú principal de Delphi.

Export to IDL (Exportar a IDL)

Pulse este botón para generar código fuente en formato MIDL o CORBA. Esto puede ser de utilidad si desea inspeccionar el IDL que Delphi genera, o si está utilizando otra herramienta que requiera la presencia de un fichero IDL.

Ahora ya tenemos creada una biblioteca de tipos válida. Cierre el Editor de Bibliotecas de Tipos y seleccione *File | Save All* en el menú principal de Delphi. Guarde *Unit1* (el fichero de implementación del objeto COM) como *AreaIntf.pas*. Guarde *Project2* (el fichero principal del proyecto) como *TIServer.dpr*. Delphi automáticamente creará un fichero llamado *TIServer.tlb*, que es la biblioteca de tipos en sí. No olvide que la biblioteca de tipos también estará incluida en *TIServer.dll* como un recurso (para construir *TIServer.dll*, seleccione la opción *Project | Build* del menú principal de Delphi).

Si echamos un vistazo al código fuente de *AreaIntf.pas*, veremos que incluye tres funciones (*TUnitAuto.Convert*, *TUnitAuto.Get_Name* y *TUnitAuto.Set_Name*). Si estuviésemos codificando un servidor COM real, por supuesto que tendríamos que proporcionar una implementación para dichos métodos. Sin embargo, en este capítulo lo que estamos estudiando es cómo crear una biblioteca de tipos, así que ignoraremos la inplementacion de estos métodos.

En la próxima sección crearemos una utilidad que permite leer y mostrar el contenido de la biblioteca de tipos creada.

Nota

Los servidores COM externos al proceso y las bibliotecas de tipos

Habrá notado que en el Capítulo 2 no ha habido ningún ejemplo de servidor COM externo al proceso. La razón es simple: los servidores externos al proceso requieren soporte de serialización (*marshaling*) para funcionar adecuadamente. Como hemos indicado en el último

capítulo, es posible proporcionar mediante código el soporte de serialización necesario, pero se trata de un tema que va más allá de los objetivos de este libro. Los servidores de automatización serializan automáticamente los tipos de datos compatibles con la automatización COM. Estos tipos fueron enumerados en el Capítulo2, y serán discutidos nuevamente en el Capítulo 4.

¿Qué hacer si deseamos escribir un servidor externo al proceso que no sea un servidor de automatización? Existe un método poco conocido para crear tales proyectos. Si vamos a la página *Flags* (en Delphi 3, a la página *Attributes*) del Editor de Bibliotecas de Tipos, veremos una casilla de verificación llamada *OleAutomation*. La selección de esta casilla obliga a Windows a utilizar su mecanismo de serialización automático para la interfaz. Sin embargo, si se va a tomar el trabajo de añadir una biblioteca de tipos a su proyecto, sería preferible que creara además un verdadero servidor de automatización, lo cual es mucho más sencillo, como verá en el Capítulo 4. (Delphi 5 simplifica algo este proceso. En la parte inferior del Asistente para Objetos COM verá una casilla de verificación con el nombre '*Mark Interface OleAutomation*'. Activando esta casilla la interfaz será 'marcada' automáticamente como de automatización).

Si no activa la casilla '*OleAutomation*', no podrá obtener ninguna referencia a ninguna interfaz del objeto COM excepto *IUnknown*. Por ejemplo, una llamada a *CreateCOMObject* seguida de un intento de convertir el resultado a la interfaz *ITest* provocará el siguiente error:

Este mensaje de error puede provocar confusión, dado que la interfaz *ITest* está soportada por el servidor COM. El problema es que Windows no sabe cómo serializar la interfaz *ITest*, dado que Ud. no ha suministrado un mecanismo de serialización personalizado.

Creando un Visor de Bibliotecas de tipos

En esta sección construiremos un programa que puede leer la información de tipos de una biblioteca de tipos y mostrarla en una vista de árbol. La cantidad de información que se almacena en una biblioteca de tipos es enorme, y la documentación para obtener información es poca e imprecisa. Por estas razones, este ejemplo sólo interpretará los datos que más frecuentemente se encuentran en un editor de bibliotecas de tipos: clases COM, interfaces, disp-interfaces y enumeradores. Si desea mejorar el ejemplo, Microsoft Developer's Network (MSDN) es la mejor fuente de información.

Por supuesto, el Editor de Bibliotecas de Tipos de Delphi es capaz de leer, mostrar y editar toda la información que se puede encontrar en una biblioteca de tipos, así que este ejemplo perseguirá fines estrictamente educativos.

La Figura 3.8 muestra el programa *TIViewer* en tiempo de ejecución.

Figura 3.8: *TIViewer*.

Los Listados 3.4 y 3.5 presentan el código fuente completo de la Aplicación *TIViewer*.

Listado 3.4: Aplicación TIViewer - *MainForm.pas*

```
unit MainForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, Menus, ComCtrls, ExtCtrls, ActiveX;
type
  TForm1 = class(TForm)
    MainMenu1: TMainMenu;
    treeTI: TTreeView;
    StatusBar1: TStatusBar;
    FileMenu: TMenuItem;
    FileExit1: TMenuItem;
    FileOpen1: TMenuItem;
    OpenDialog1: TOpenDialog;
    procedure FileExit1Click(Sender: TObject);
    procedure FileOpen1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
    FLib: ITypeLib;
    FEnumRoot: TTreeNode;
    FDispatchRoot: TTreeNode;
    FInterfaceRoot: TTreeNode;
    FCoClassRoot: TTreeNode;
    FUnknownRoot: TTreeNode;
    procedure LoadTypeLibrary;
    procedure LoadEnum(TypeInfo: ITypeInfo; TypeAttr: PTypeAttr);
    procedure LoadInterface(TypeInfo: ITypeInfo; TypeAttr: PTypeAttr);
    procedure LoadCoClass(TypeInfo: ITypeInfo; TypeAttr: PTypeAttr);
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
uses
  ComObj;
{$R *.DFM}
procedure TForm1.FileExit1Click(Sender: TObject);
begin
  Close;
end;
procedure TForm1.LoadEnum(TypeInfo: ITypeInfo; TypeAttr: PTypeAttr);
var
  TypeLibNode: TTreeNode;
  AName: WideString;
  ADocString: WideString;
  AHelpContext: LongInt;
  ValuesNode: TTreeNode;
  ValueNode: TTreeNode;
```

```
  ValueIndex: Integer;
  VarDesc: PVarDesc;
begin
  TypeInfo.GetDocumentation(-1, @AName, @ADocString, @AHelpContext, nil);
  TypeLibNode := treeTI.Items.AddChild(FEnumRoot, AName);
  treeTI.Items.AddChild(TypeLibNode, 'Version ' +
    IntToStr(TypeAttr.wMajorVerNum) + '.' +
IntToStr(TypeAttr.wMinorVerNum));
  if ADocString  '' then
    treeTI.Items.AddChild(TypeLibNode, 'Help string: ' + ADocString);
  if AHelpContext  0 then
    treeTI.Items.AddChild(TypeLibNode, 'Help context: ' +
      IntToStr(AHelpContext));
  ValuesNode := treeTI.Items.AddChild(TypeLibNode, 'Values');
  for ValueIndex := 0 to TypeAttr.cVars - 1 do begin
    TypeInfo.GetVarDesc(ValueIndex, VarDesc);
    try
      TypeInfo.GetDocumentation(VarDesc.memid, @AName, @ADocString,
        @AHelpContext, nil);
      ValueNode := treeTI.Items.AddChild(ValuesNode, AName);
      if ADocString  '' then
        treeTI.Items.AddChild(ValueNode, 'Help string: ' +
          ADocString);
      if AHelpContext  0 then
        treeTI.Items.AddChild(ValueNode, 'Help context: ' +
          IntToStr(AHelpContext));
    finally
      TypeInfo.ReleaseVarDesc(VarDesc);
    end;
  end;
end;
procedure TForm1.LoadInterface(TypeInfo: ITypeInfo;
  TypeAttr: PTypeAttr);
var
  TypeLibNode: TTreeNode;
  AName: WideString;
  ADocString: WideString;
  AHelpContext: LongInt;
  FunctionsNode: TTreeNode;
  FunctionNode: TTreeNode;
  FuncIndex: Integer;
  FuncDesc: PFuncDesc;
  ParametersNode: TTreeNode;
  ParamIndex: Integer;
  Names: PBStrList;
  cNames: Integer;
begin
  TypeInfo.GetDocumentation(-1, @AName, @ADocString, @AHelpContext, nil);
  if TypeAttr.typeKind = TKIND_DISPATCH then
    TypeLibNode := treeTI.Items.AddChild(FDispatchRoot, AName)
  else
    TypeLibNode := treeTI.Items.AddChild(FInterfaceRoot, AName);
```

```
  treeTI.Items.AddChild(TypeLibNode, GUIDToString(TypeAttr.GUID));
  treeTI.Items.AddChild(TypeLibNode, 'Version ' +
    IntToStr(TypeAttr.wMajorVerNum) + '.' +
IntToStr(TypeAttr.wMinorVerNum));
  if ADocString  '' then
    treeTI.Items.AddChild(TypeLibNode, 'Help string: ' + ADocString);
  if AHelpContext  0 then
    treeTI.Items.AddChild(TypeLibNode, 'Help context: ' +
      IntToStr(AHelpContext));
  New(Names);
  try
    // cargar funciones
    FunctionsNode := treeTI.Items.AddChild(TypeLibNode, 'Functions');
    for FuncIndex := 0 to TypeAttr.cFuncs - 1 do begin
      TypeInfo.GetFuncDesc(FuncIndex, FuncDesc);
      try
        TypeInfo.GetDocumentation(FuncDesc.memid, @AName,
          @ADocString, @AHelpContext, nil);
        FunctionNode := treeTI.Items.AddChild(FunctionsNode, AName);
        if ADocString  '' then
          treeTI.Items.AddChild(FunctionNode, 'Help string: ' +
ADocString);
        if AHelpContext  0 then
          treeTI.Items.AddChild(FunctionNode, 'Help context: ' +
            IntToStr(AHelpContext));
        if FuncDesc.cParams > 0 then begin
          // cargar par□metros
          ParametersNode := treeTI.Items.AddChild(FunctionNode,
'Parameters');
          TypeInfo.GetNames(FuncDesc.memid, Names, sizeof(TBStrList),
cNames);
          // Saltamos Names[0] □ es el nombre de la funci□n
          for ParamIndex := 1 to FuncDesc.cParams do
            treeTI.Items.AddChild(ParametersNode, Names[ParamIndex]);
        end;
      finally
        TypeInfo.ReleaseFuncDesc(FuncDesc);
      end;
    end;
  finally
    Dispose(Names);
  end;
end;
procedure TForm1.LoadCoClass(TypeInfo: ITypeInfo;
  TypeAttr: PTypeAttr);
var
  TypeLibNode: TTreeNode;
  AName: WideString;
  ADocString: WideString;
  AHelpContext: LongInt;
  FunctionsNode: TTreeNode;
  FuncIndex: Integer;
  RefType: Cardinal;
```

```
  TypeInfo2: ITypeInfo;
begin
  TypeInfo.GetDocumentation(-1, @AName, @ADocString, @AHelpContext, nil);
  TypeLibNode := treeTI.Items.AddChild(FCoClassRoot, AName);
  treeTI.Items.AddChild(TypeLibNode, GUIDToString(TypeAttr.GUID));
  if ADocString  '' then
    treeTI.Items.AddChild(TypeLibNode, 'Help string: ' + ADocString);
  if AHelpContext  0 then
    treeTI.Items.AddChild(TypeLibNode, 'Help context: ' +
      IntToStr(AHelpContext));
  FunctionsNode := treeTI.Items.AddChild(TypeLibNode, 'Implements');
  for FuncIndex := 0 to TypeAttr.cImplTypes - 1 do begin
    TypeInfo.GetRefTypeOfImplType(FuncIndex, RefType);
    TypeInfo.GetRefTypeInfo(RefType, TypeInfo2);
    TypeInfo2.GetDocumentation(-1, @AName, nil, nil, nil);
    treeTI.Items.AddChild(FunctionsNode, AName);
  end;
end;
procedure TForm1.LoadTypeLibrary;
var
  LibAttr: PTLibAttr;
  TypeInfo: ITypeInfo;
  TypeLibIndex: Integer;
  AName: WideString;
  ADocString: WideString;
  AHelpContext: LongInt;
  AHelpFile: WideString;
  TypeAttr: PTypeAttr;
  RootNode: TTreeNode;
begin
  Screen.Cursor := crHourglass;
  try
    treeTI.Items.BeginUpdate;
    try
      treeTI.Items.Clear;
      FLib.GetDocumentation(-1, @AName, @ADocString, @AHelpContext,
        @AHelpFile);
      // Cargar informaci□n de ayuda
      RootNode := treeTI.Items.AddChild(nil, AName);
      // GUID, versi□n, LCID
      OleCheck(FLib.GetLibAttr(LibAttr));
      try
        treeTI.Items.AddChild(RootNode, GUIDToString(LibAttr.GUID));
        treeTI.Items.AddChild(RootNode, 'Version ' +
          IntToStr(LibAttr.wMajorVerNum) + '.' +
          IntToStr(LibAttr.wMinorVerNum));
        treeTI.Items.AddChild(RootNode, 'LCID: ' +
          IntToStr(LibAttr.lcid));
      finally
        FLib.ReleaseTLibAttr(LibAttr);
      end;
      // Informaci□n de ayuda
```

```
      if ADocString  '' then
        treeTI.Items.AddChild(RootNode, 'Help string: ' + ADocString);
      if AHelpContext  0 then
        treeTI.Items.AddChild(RootNode, 'Help context: ' +
          IntToStr(AHelpContext));
      if AHelpFile  '' then
        treeTI.Items.AddChild(RootNode, 'Help file: ' + AHelpFile);
      // Inicializar nodos para los diversos tipos
      FEnumRoot := treeTI.Items.AddChild(RootNode, 'Enum');
      FDispatchRoot := treeTI.Items.AddChild(RootNode, 'Dispatch');
      FInterfaceRoot := treeTI.Items.AddChild(RootNode, 'Interface');
      FCoClassRoot := treeTI.Items.AddChild(RootNode, 'CoClass');
      FUnknownRoot := treeTI.Items.AddChild(RootNode, 'Unknown');
      for TypeLibIndex := 0 to FLib.GetTypeInfoCount - 1 do begin
        FLib.GetTypeInfo(TypeLibIndex, TypeInfo);
        OleCheck(TypeInfo.GetTypeAttr(TypeAttr));
        try
          case TypeAttr.typeKind of
            TKIND_ENUM:
              LoadEnum(TypeInfo, TypeAttr);
            TKIND_DISPATCH,
            TKIND_INTERFACE:
              LoadInterface(TypeInfo, TypeAttr);
            TKIND_COCLASS:
              LoadCoClass(TypeInfo, TypeAttr);
            else
              treeTI.Items.AddChild(FUnknownRoot,
                'Unrecognized type: ' + IntToStr(TypeAttr.typeKind));
          end;
        finally
          TypeInfo.ReleaseTypeAttr(TypeAttr);
        end;
      end;
      RootNode.Expand(False);
    finally
      treeTI.Items.EndUpdate;
    end;
  finally
    Screen.Cursor := crDefault;
  end;
end;
procedure TForm1.FileOpen1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then begin
    OleCheck(LoadTypeLib(PWideChar(WideString(OpenDialog1.FileName)),
      FLib));
    LoadTypeLibrary;
  end;
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  if ParamCount = 1 then begin
```

```
    try
      OleCheck(LoadTypeLib(PWideChar(WideString(ParamStr(1))),
        FLib));
      LoadTypeLibrary;
    except
      on E: Exception do begin
        ShowMessage(E.Message);
        Application.Terminate;
      end;
    end;
  end;
end;
end.
```

Listado 3.5: Aplicación TIViewer - *TIViewer.dpr*

```
program TIViewer;
uses
 Forms,
 MainForm in 'MainForm.pas' {Form1};
{$R *.RES}
begin
 Application.Initialize;
 Application.Title := 'Type Information Viewer';
 Application.CreateForm(TForm1, Form1);
 Application.Run;
end;
```

El código mostrado en el Listado 3.4 hace un uso intensivo de la interfaz *ITypeInfo*, que está definida en *ActiveX.pas* como:

```
type
  ITypeInfo = interface (IUnknown)
    ['{00020401-0000-0000-C000-000000000046}']
    function GetTypeAttr(out ptypeattr: PTypeAttr): HResult; stdcall;
    function GetTypeComp(out tcomp: ITypeComp): HResult; stdcall;
    function GetFuncDesc(index: Integer; out pfuncdesc: PFuncDesc):
HResult;
      stdcall;
    function GetVarDesc(index: Integer; out pvardesc: PVarDesc): HResult;
      stdcall;
    function GetNames(memid: TMemberID; rgbstrNames: PBStrList;
      cMaxNames: Integer; out cNames: Integer): HResult; stdcall;
    function GetRefTypeOfImplType(index: Integer;
      out reftype: HRefType): HResult; stdcall;
    function GetImplTypeFlags(index: Integer;
      out impltypeflags: Integer): HResult; stdcall;
    function GetIDsOfNames(rgpszNames: POleStrList; cNames: Integer;
      rgmemid: PMemberIDList): HResult; stdcall;
    function Invoke(pvInstance: Pointer; memid: TMemberID; flags: Word;
      var dispParams: TDispParams; varResult: PVariant;
      excepInfo: PExcepInfo; argErr: PInteger): HResult; stdcall;
```

```
    function GetDocumentation(memid: TMemberID; pbstrName: PWideString;
      pbstrDocString: PWideString; pdwHelpContext: PLongint;
      pbstrHelpFile: PWideString): HResult; stdcall;
    function GetDllEntry(memid: TMemberID; invkind: TInvokeKind;
      bstrDllName, bstrName: PWideString; wOrdinal: PWord): HResult;
      stdcall;
    function GetRefTypeInfo(reftype: HRefType; out tinfo: ITypeInfo):
HResult;
      stdcall;
    function AddressOfMember(memid: TMemberID; invkind: TInvokeKind;
      out ppv: Pointer): HResult; stdcall;
    function CreateInstance(const unkOuter: IUnknown; const iid: TIID;
      out vObj): HResult; stdcall;
    function GetMops(memid: TMemberID; out bstrMops: WideString): HResult;
      stdcall;
    function GetContainingTypeLib(out tlib: ITypeLib;
      out pindex: Integer): HResult; stdcall;
    procedure ReleaseTypeAttr(ptypeattr: PTypeAttr); stdcall;
    procedure ReleaseFuncDesc(pfuncdesc: PFuncDesc); stdcall;
    procedure ReleaseVarDesc(pvardesc: PVarDesc); stdcall;
  end;
```

Revisando esta interfaz, encontraremos métodos tales como *GetFuncDesc*, *GetVarDesc* y otros que son utilizados en *TIViewer* para obtener los detalles de una biblioteca de tipos, tales como las interfaces definidas, sus métodos, y las declaraciones de éstos.

Como veremos en el Capítulo 4, algunos controladores de automatización (normalmente, aquellos que soportan enlace tardío) hacen llamadas automáticas a las funciones *GetIDsOfNames* e *Invoke* de la interfaz *ITypeInfo*. *FileOpen1Click* es el manejador de eventos responsable de iniciarlo todo. Este llama a la función *LoadTypeLib* del API de Windows para obtener una referencia a la biblioteca de tipos que el usuario ha seleccionado en el cuadro de diálogo correspondiente.

LoadTypeLib tiene dos parámetros: el nombre del fichero que contiene la biblioteca de tipos y una interfaz *ITypeLib* a través de la cual se devuelve la referencia a la biblioteca de tipos. El fichero original puede ser un fichero .TLB, o una DLL o fichero ejecutable que contenga una biblioteca de tipos.

Después que la biblioteca de tipos es cargada, *LoadTypeLibrary* descifra la información almacenada en la biblioteca de tipos y rellena la vista de árbol. *LoadTypeLibrary* realiza una llamada a *GetDocumentation* para obtener diversa información sobre la propia biblioteca, tal como el nombre, fichero de ayuda, y contexto de ayuda. *GetDocumentation* se define de la siguiente forma:

```
function GetDocumentation(memid: TMemberID; pbstrName: PWideString;
  pbstrDocString: PWideString; pdwHelpContext: PLongint;
  pbstrHelpFile: PWideString): HResult; stdcall;
```

El primer parámetro, *memid*, acepta el índice del elemento cuya documentación va a ser devuelta. Especifique -1 para obtener la documentación de la biblioteca en sí. Esta función se utiliza posteriormente para devolver la información sobre las interfaces de la biblioteca de tipos.

Los parámetros *pbstrName*, *pbstrDocString*, *pdwHelpContext*, y *pbstrHelpFile* devuelven el nombre, la cadena de documentación, el contexto de ayuda y el nombre del fichero de ayuda del elemento en cuestión. En el Editor de Bibliotecas de Tipos, esta información es introducida en el panel de información para la biblioteca.

Si alguno de estos datos no es importante para nosotros, podemos pasar el valor **nil** en el parámetro correspondiente. Por ejemplo, existe sólo un fichero de ayuda para una biblioteca de tipos, y por ello después que determinemos el nombre del fichero para una biblioteca dada, pasaremos **nil** a *GetDocumentation* en todas las llamadas subsiguientes.

A continuación, *LoadTypeLibrary* crea algunos nodos en el árbol:

```
for TypeLibIndex := 0 to FLib.GetTypeInfoCount  1 do begin
  FLib.GetTypeInfo(TypeLibIndex, TypeInfo);
  OleCheck(TypeInfo.GetTypeAttr(TypeAttr));
  try
    case TypeAttr.TypeKind of
    TKIND_ENUM:
       LoadEnum(TypeInfo, TypeAttr);
    TKIND_DISPATCH;
    TKIND_INTERFACE:
       LoadInterface(TypeInfo, TypeAttr);
    TKIND_COCLASS:
       LoadCoClass(TypeInfo, TypeAttr);
    else
       TreeTI.Items.AddChild(FUnknownRoot, 'Unrecognized type: '
         + IntToStr(TypeAttr, TypeKind));
    end;
  finally
    TypeInfo.ReleaseTypeAttr(TypeAttr);
  end;
end;
```

GetTypeInfoCount devuelve la cantidad de descripciones de tipo presentes en la biblioteca de tipos. El código posteriormente recorre en un bucle cada descripción de tipo, obteniendo una referencia a la información del tipo a través de una llamada a *GetTypeInfo*.

GetTypeInfo obtiene una referencia a la información de tipo para un elemento dado de la biblioteca de tipos. Simplemente se le debe pasar el índice del elemento en cuestión, y *GetTypeInfo* recupera una referencia a una interfaz *ITypeInfo*.

La interfaz *ITypeInfo* nos permite entonces obtener una referencia a una interfaz *ITypeAttr*, a través de una llamada a *GetTypeAttr*. *ITypeAttr* nos informa con qué tipo de elemento estamos tratando (interfaz, disp-interfaz, etc.).

Para hacer el código más comprensible, *LoadTypeLibrary* transfiere el control a uno de varios procedimientos diferentes, en dependencia del tipo de la información recuperada. Podemos ver en el código que *LoadTypeLibrary* puede tratar con enumeradores, interfaces, disp-interfaces, y co-clases. Cualquier otro tipo de información encontrada en la biblioteca es ignorada. La Figura 3.9 nos muestra la biblioteca de tipos de Microsoft Word 2000 (el fichero *MSWORD9.OLB*, normalmente situado en *C:\Archivos de programa\Microsoft Office\Office*).

Figura 3.9: Biblioteca de tipos de Microsoft Word 2000 en *TIViewer*.

Para facilitar su uso, en el manejador de eventos *FormCreate*, *TIViewer* comprueba si el usuario ha especificado un fichero en la línea de comandos. En ese caso, *TIViewer* trata de abrir el fichero inmediatamente. Si la operación falla por cualquier motivo, *TIViewer* muestra un mensaje de error y finaliza la ejecución.

Interpretando enumeraciones

Cuando *LoadTypeLibrary* detecta una enumeración, llama al procedimiento *LoadEnum* para decodificar la enumeración.

Inicialmente, *LoadEnum* hace una llamada a *GetDocumentation* para obtener el nombre, la cadena de documentación y el contexto de ayuda para la enumeración. Observe que hemos pasado el valor **nil** para el fichero de ayuda, pues ya hemos obtenido éste anteriormente.

ITypeAttr.cVar devuelve la cantidad de constantes definidas en la enumeración. Recorriendo los elementos en *cVars* y llamando a *GetDocumentation* para cada uno de ellos, podemos determinar los nombres de las constantes de la enumeración. *ITypeInfo.GetVarDesc* devuelve una estructura que describe el valor, incluyendo un identificador (*ID*) que podemos pasar a *GetDocumentation*.

Observe que la llamada a *GetVarDesc* reserva memoria internamente. Al finalizar, deberemos llamar a *ReleaseVarDesc* para liberar esa memoria. Hemos colocado este código en un bloque **try**-**finally** para garantizar que, no importa qué pase, la memoria será liberada.

Interpretando interfaces y disp-interfaces

Cuando *LoadTypeLibrary* detecta una interfaz o disp-interfaz, llama al procedimiento *LoadInterface* para decodificar la interfaz.

LoadInterface comienza de forma muy similar a *LoadEnum*. Llama a *GetDocumentation* para obtener el nombre, la cadena de documentación y el contexto de ayuda de la interfaz, y luego recorre en un bucle las funciones (métodos) definidas en la interfaz.

Del mismo modo que *ITypeAttr.cVar* devuelve el número de constantes en una enumeración, *ITypeAttr.cFuncs* devuelve la cantidad de métodos soportados por la interfaz. De forma análoga a como hicimos con *GetVarDesc*, debemos llamar a *GetFuncDesc* para obtener la información sobre cada función. Nuevamente, estamos interesados en obtener el campo *memid* del registro, así que podemos pasarlo a *GetDocumentation* para obtener el nombre de la función.

El registro *FuncDesc* nos comunica cuántos parámetros acepta la función en el campo *cParams*. Podemos entonces llamar al método *GetNames* para cargar una lista de nombres de parámetros.

Observe que al terminar de utilizar *FuncDesc* debemos llamar a *ReleaseFuncDesc* para liberar la memoria reservada por *GetFuncDesc*. Nuevamente hemos colocado este código en un bloque **try**-**finally** para garantizar que, no importa qué ocurra, esa memoria será liberada.

Interpretando co-clases

El último tipo de elemento de una biblioteca de tipos que nuestro programa reconoce es la co-clase. *LoadCoClass* es el procedimiento responsable de analizar la información asociada a una co-clase.

Como es usual, al inicio del procedimiento realizamos una llamada *GetDocumentation* para obtener el nombre de la co-clase, así como su cadena de documentación y contexto de ayuda.

ITypeAttr proporciona un campo llamado *cImplTypes*, que nos comunica cuántas interfaces implementa la co-clase. Si consulta el Listado 3.2, podrá observar que *LoadCoClass* realiza una llamada a *GetRefTypeOfImplType*. Esta devuelve un número de referencia a la interfaz implementada. *GetRefTypeInfo* devuelve la información de tipo para esa interfaz. La información de tipos devuelta es idéntica a la información acerca de la interfaz devuelta por el procedimiento *LoadInterface*. En este punto, todo lo que hace TIViewer es obtener el nombre de la interfaz (a través de GetDocumentation) para mostrarlo en el árbol.

Extendiendo TIViewer

Como habrá podido darse cuenta, el código del ejemplo no es particularmente difícil. La peor parte radica en buscar en la documentación de Microsoft para descifrar qué está sucediendo internamente.

Nota

Según me han explicado los arquitectos de Delphi, incluso después de revisar toda la documentación de Microsoft queda aún un largo trecho que recorrer para establecer cómo funciona todo. Por ejemplo, se supone que la primera interfaz asociada a una co-clase es la interfaz por defecto, y la segunda interfaz debe ser la interfaz de eventos. Sin embargo, algunas bibliotecas de tipos no siguen ese convenio, así que no es posible asumir siempre esa ordenación.

Si el ejemplo anterior ha despertado su interés, podría intentar extenderlo para que fuese capaz de mostrar los valores de las constantes en una enumeración, o los tipos de los parámetros de los métodos. La mejor fuente de información sobre las interfaces utilizadas en el ejemplo (*ITypeLib*, *ITypeInfo* e *ITypeAttr*) es Microsoft Developer's Network (MSDN).

Resumen

En este capítulo hemos mostrado qué es una biblioteca de tipos, y cómo añadir una a nuestros servidores COM. Hemos aprendido a editar una biblioteca de tipos con el objetivo de añadir métodos, propiedades y enumeraciones a nuestros servidores.

También hemos mostrado cómo nuestras aplicaciones pueden leer e interpretar la información almacenada en una biblioteca de tipos.

En el próximo capítulo utilizaremos intensamente las bibliotecas de tipos en nuestros servidores de automatización. Como veremos, no es necesario leer la biblioteca de tipos directamente en una aplicación cliente, ya que Delphi suministra las herramientas para generar automáticamente código Object Pascal a partir de una biblioteca de tipos.

ITypeAttr proporciona un campo llamado *cImplTypes*, que nos comunica cuántas interfaces implementa la co-clase. Si consulta el Listado 3.2, podrá observar que *LoadCoClass* realiza una llamada a *GetRefTypeOfImplType*. Esta devuelve un número de referencia a la interfaz implementada. *GetRefTypeInfo* devuelve la información de tipo para esa interfaz. La información de tipos devuelta es idéntica a la información acerca de la interfaz devuelta por el procedimiento *LoadInterface*. En este punto, todo lo que hace TIViewer es obtener el nombre de la interfaz (a través de GetDocumentation) para mostrarlo en el árbol.

Extendiendo TIViewer

Como habrá podido darse cuenta, el código del ejemplo no es particularmente difícil. La peor parte radica en buscar en la documentación de Microsoft para descifrar qué está sucediendo internamente.

Nota

Según me han explicado los arquitectos de Delphi, incluso después de revisar toda la documentación de Microsoft queda aún un largo trecho que recorrer para establecer cómo funciona todo. Por ejemplo, se supone que la primera interfaz asociada a una co-clase es la interfaz por defecto, y la segunda interfaz debe ser la interfaz de eventos. Sin embargo, algunas bibliotecas de tipos no siguen ese convenio, así que no es posible asumir siempre esa ordenación.

Si el ejemplo anterior ha despertado su interés, podría intentar extenderlo para que fuese capaz de mostrar los valores de las constantes en una enumeración, o los tipos de los parámetros de los métodos. La mejor fuente de información sobre las interfaces utilizadas en el ejemplo (*ITypeLib*, *ITypeInfo* e *ITypeAttr*) es Microsoft Developer's Network (MSDN).

Resumen

En este capítulo hemos mostrado qué es una biblioteca de tipos, y cómo añadir una a nuestros servidores COM. Hemos aprendido a editar una biblioteca de tipos con el objetivo de añadir métodos, propiedades y enumeraciones a nuestros servidores.

También hemos mostrado cómo nuestras aplicaciones pueden leer e interpretar la información almacenada en una biblioteca de tipos.

En el próximo capítulo utilizaremos intensamente las bibliotecas de tipos en nuestros servidores de automatización. Como veremos, no es necesario leer la biblioteca de tipos directamente en una aplicación cliente, ya que Delphi suministra las herramientas para generar automáticamente código Object Pascal a partir de una biblioteca de tipos.

ITypeAttr proporciona un campo llamado *cImplTypes*, que nos comunica cuántas interfaces implementa la co-clase. Si consulta el Listado 3.2, podrá observar que *LoadCoClass* realiza una llamada a *GetRefTypeOfImplType*. Esta devuelve un número de referencia a la interfaz implementada. *GetRefTypeInfo* devuelve la información de tipo para esa interfaz. La información de tipos devuelta es idéntica a la información acerca de la interfaz devuelta por el procedimiento *LoadInterface*. En este punto, todo lo que hace TIViewer es obtener el nombre de la interfaz (a través de GetDocumentation) para mostrarlo en el árbol.

Extendiendo TIViewer

Como habrá podido darse cuenta, el código del ejemplo no es particularmente difícil. La peor parte radica en buscar en la documentación de Microsoft para descifrar qué está sucediendo internamente.

Nota

Según me han explicado los arquitectos de Delphi, incluso después de revisar toda la documentación de Microsoft queda aún un largo trecho que recorrer para establecer cómo funciona todo. Por ejemplo, se supone que la primera interfaz asociada a una co-clase es la interfaz por defecto, y la segunda interfaz debe ser la interfaz de eventos. Sin embargo, algunas bibliotecas de tipos no siguen ese convenio, así que no es posible asumir siempre esa ordenación.

Si el ejemplo anterior ha despertado su interés, podría intentar extenderlo para que fuese capaz de mostrar los valores de las constantes en una enumeración, o los tipos de los parámetros de los métodos. La mejor fuente de información sobre las interfaces utilizadas en el ejemplo (*ITypeLib*, *ITypeInfo* e *ITypeAttr*) es Microsoft Developer's Network (MSDN).

Resumen

En este capítulo hemos mostrado qué es una biblioteca de tipos, y cómo añadir una a nuestros servidores COM. Hemos aprendido a editar una biblioteca de tipos con el objetivo de añadir métodos, propiedades y enumeraciones a nuestros servidores.

También hemos mostrado cómo nuestras aplicaciones pueden leer e interpretar la información almacenada en una biblioteca de tipos.

En el próximo capítulo utilizaremos intensamente las bibliotecas de tipos en nuestros servidores de automatización. Como veremos, no es necesario leer la biblioteca de tipos directamente en una aplicación cliente, ya que Delphi suministra las herramientas para generar automáticamente código Object Pascal a partir de una biblioteca de tipos.

4
Automatización

En los tres capítulos anteriores hemos invertido mucho tiempo examinando las interfaces y cómo crear objetos COM. También hemos introducido en el Capítulo 3 las bibliotecas de tipos. En este capítulo es donde todo lo visto con anterioridad se integra, y si se ha sentido decepcionado hasta el momento con COM, a partir de aquí cambiará su opinión. El capítulo es extenso, pero contiene muchas aplicaciones de ejemplo, y personalmente pienso que es el capítulo más importante del libro. En este capítulo examinaremos los siguientes puntos:

- Interfaces y automatización
- Variantes y automatización
- Disp-interfaces
- Interfaces duales
- Microsoft ADO

En el Capítulo 3 hemos estudiado qué es una biblioteca de tipos, cómo crearlas y extraer información de ellas. Como hemos visto, obtener la información manualmente es, cuanto menos, tedioso. La buena noticia es que normalmente no necesitaremos trabajar directamente con los datos de la biblioteca de tipos. En la mayoría de los casos trabajaremos con la biblioteca de tipos indirectamente a través de la *automatización*.

Definiendo la automatización

Como su nombre sugiere, la automatización es una forma de controlar automáticamente una aplicación desde otra. De hecho, podemos crear servidores de automatización internos al proceso y externos al proceso. Como veremos más adelante en el capítulo, cada tipo de controlador de automatización ofrece sus ventajas.

Delphi simplifica tanto la creación y uso de servidores de automatización, que prácticamente no es necesario siquiera dominar las bases de COM para ello. Crear un cliente de automatización en Delphi es trivial:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  V : Variant;
```

```
begin
  V := CreateOleObject('Word.Basic');
  V.AppShow;
  V.FileNew;
  V.Insert('La automatizaci□n es sencilla');
end;
```

Créalo o no, este pequeño fragmento de código arranca Microsoft Word, crea un nuevo documento, e inserta en él el texto "La automatización es sencilla". Más adelante en este capítulo, en la sección titulada "Variantes", examinaremos qué ocurre exactamente en este código, pero mi objetivo en este momento es mostrar cuán simple puede ser el uso de la automatización en Delphi.

Interfaces

Asumo que entiende qué son las interfaces y cómo utilizarlas. Las interfaces constituyen uno de los dos métodos que pueden ser usados para obtener acceso a los objetos COM de un servidor de automatización.

Cuando controlamos un servidor de automatización a través de una interfaz estamos usando *enlace temprano*. Enlace temprano simplemente significa que todas las llamadas a los métodos de la interfaz son comprobadas en tiempo de compilación. Como programadores de Delphi, Ud. debe estar familiarizado con el enlace temprano.

Hemos visto numerosos ejemplos de interfaces en los capítulos anteriores, así que no mostraré aquí un ejemplo de procedimiento para controlar un servidor de automatización a través de interfaces. A lo largo de este capítulo encontrará muchos ejemplos en los que se utilizan interfaces para controlar un servidor de automatización.

Variantes

En el Capítulo 2, "Las Interfaces y COM", hemos examinado brevemente las variantes, enfocando fundamentalmente la discusión hacia los *arrays* variantes. Aunque no sea evidente, las variantes también pueden ser utilizadas para controlar un servidor de automatización.

Cuando controlamos un servidor de automatización a través de variantes, estamos haciendo uso del *enlace tardío*. Enlace tardío significa que las llamadas a los métodos no serán resueltas hasta el momento de la ejecución. Esto tiene algunas implicaciones interesantes. Consideremos el siguiente código, que se conecta a Microsoft Word y le ordena que cree un nuevo fichero:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  V: Variant;
```

```
begin
  V := CreateOleObject('Word.Basic');
  V.AppShow;
  V.FileNew;
  V.Insert('La automatizaci□n es sencilla');
end;
```

Por el momento no entraremos en detalles con respecto a la función *CreateOleObject*. Baste por ahora decir que se utiliza para crear una instancia de un servidor de automatización. Posteriormente analizaremos esta función más detalladamente.

En este ejemplo, *CreateOleObject* arranca una copia de Microsoft Word. Luego obtiene una referencia a la interfaz *Word.Basic* y la almacena en la variante *V*.

Normalmente los servidores de automatización se inician ocultos. El método *AppShow* mostrará en pantalla la aplicación Word. Entonces se realiza una llamada a *V.FileNew*, que crea un nuevo documento Word. Delphi (y Windows) realizan una gran cantidad de trabajo entre bastidores para transformar la llamada *V.FileNew* en una llamada a función que crea un nuevo documento.

Es interesante observar que una variante no es un puntero a un objeto. ¿Cómo entonces es posible llamar a los métodos de ese "objeto"? La respuesta está en el enlace tardío. Delphi realmente aceptará cualquier cosa que escribamos, sin realizar ningún tipo de chequeo en tiempo de compilación; en otras palabras, Delphi no realiza enlace temprano sobre variantes. Por ejemplo podríamos escribir el siguiente código:

```
V.DelphiRules;
```

Delphi compilará este código correctamente, aunque podemos estar totalmente seguros de que Word no implementa ningún método llamado *DelphiRules*. Si tratamos de ejecutar esta instrucción, obtendremos el error de ejecución que se muestra en la Figura 4.1.

Figura 4.1: Llamada a un método inexistente de un servidor de automatización.

Para comprender cómo Delphi convierte este código en llamadas a métodos debemos echar un vistazo a la interfaz *IDispatch*. *IDispatch* está definida en el fichero *System.pas* de la siguiente forma:

```
type
  IDispatch = interface(IUnknown)
```

```
    ['{00020400-0000-0000-C000-000000000046}']
    function GetTypeInfoCount(out Count: Integer): HResult; stdcall;
    function GetTypeInfo(Index, LocaleID: Integer;
      out TypeInfo): HResult; stdcall;
    function GetIDsOfNames(const IID: TGUID; Names: Pointer;
      NameCount, LocaleID: Integer; DispIDs: Pointer): HResult; stdcall;
    function Invoke(DispID: Integer; const IID: TGUID; LocaleID: Integer;
      Flags: Word; var Params;
      VarResult, ExcepInfo, ArgErr: Pointer): HResult; stdcall;
  end;
```

Los métodos *GetTypeInfoCount* y *GetTypeInfo* deberían despertar nuestra curiosidad. *IDispatch* debe consultar la biblioteca de tipos asociada al controlador de automatización para resolver las llamadas a funciones en tiempo de ejecución. Eso es totalmente correcto. De hecho, lo que ocurre es lo siguiente:

1. Delphi pasa el nombre del método a *IDispatch.GetIDsOfNames*.
2. *GetIDsOfNames* devuelve un identificador entero (ID) que representa el nombre del método.
3. Delphi llama al método *Invoke*, usando el identificador obtenido en el paso 2.

Esta secuencia de pasos se ejecuta cada vez que llamamos a un método de un objeto a través de una variante. El tiempo requerido para ejecutar estos tres pasos es considerablemente mayor que el necesario para acceder directamente al mismo objeto a través de una interfaz.

Como programadores de Delphi, podríamos preguntarnos: ¿por qué razón nadie sobre la tierra desearía usar *IDispatch* para acceder a un servidor de automatización? Existen dos buenas razones para ello:

1. Estamos desarrollando un programa o macro en una aplicación que no soporta las interfaces, como Visual Basic o Microsoft Word.
2. Deseamos desarrollar rápidamente una aplicación cliente en Delphi, y no deseamos molestarnos en importar una biblioteca de tipos.

Disp-interfaces

Las *disp-interfaces* se sitúan en un punto intermedio entre las interfaces y las variantes. La declaración de una *disp-interfaz* es prácticamente idéntica a la de una interfaz. He aquí la declaración de una interfaz sencilla, y seguidamente la declaración de la *disp-interfaz* correspondiente.

```
// ******************************************************
// Interface: IUnitAuto
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {A1E420C1-F75F-11D2-B3B9-0040F67455FE}
// ******************************************************
```

```
  IUnitAuto = interface(IDispatch)
    ['{A1E42OC1-F75F-11D2-B3B9-OO4OF67455FE}']
    function Convert(Quantity: Double; InUnit: Integer;
      OutUnit: Integer): Double; safecall;
  end;
// *********************************************************
// DispIntf:  IUnitAutoDisp
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {A1E42OC1-F75F-11D2-B3B9-OO4OF67455FE}
// *********************************************************
  IUnitAutoDisp = dispinterface
    ['{A1E42OC1-F75F-11D2-B3B9-OO4OF67455FE}']
    function Convert(Quantity: Double; InUnit: Integer;
      OutUnit: Integer): Double; dispid 1;
  end;
```

Este código generado automáticamente por Delphi ha sido tomado directamente del primer ejemplo de este capítulo, así que por el momento no vamos a explicarlo en detalle. Sin embargo, a primera vista debemos observar algunos aspectos relacionados con el código:

- La interfaz *IUnitAuto* se deriva de *IDispatch*, así que si creamos un servidor de automatización que implemente *IUnitAuto*, los clientes podrán usar enlace tardío para comunicarse con éste.
- El método *Convert* de *IUnitAuto* está declarado como **safecall**.
- Tanto la interfaz *IUnitAuto* como la disp-interfaz *IUnitAutoDisp* usan el mismo GUID.
- La disp-interfaz *IUnitAutoDisp* define el mismo método (*Convert*) que la interfaz *IUnitAuto*. Sin embargo, la disp-interfaz incluye '**dispid** 1' a continuación del nombre del método.

Cuando se utiliza una disp-interfaz, se eliminan los dos primeros pasos asociados con *IDispatch*. El controlador de automatización no tiene que llamar a *GetIDsOfNames*, y el servidor no tiene que responder con el **dispid** del método en cuestión. En lugar de ello, el **dispid** es determinado en tiempo de compilación. En tiempo de ejecución el programa cliente simplemente llama a *Invoke* utilizando el **dispid** predeterminado.

Las disp-interfaces son simplemente una conveniencia para los clientes. Realmente no implementamos una disp-interfaz en el servidor. El servidor implementa interfaces. La aplicación cliente puede elegir conectarse al servidor a través de variantes o disp-interfaces, asumiendo que el objeto COM del servidor soporta también la interfaz *IDispatch*.

Interfaces Duales

Una *interfaz dual* se define simplemente como un servidor de automatización que soporta conexiones de clientes usando interfaces (enlace temprano) y también usando variantes (enlace tardío). Cualquier servidor de automatización que creemos con Delphi soportará automáticamente una interfaz dual. Esto es provechoso, ya que significa que cualquier servidor de automatización que escribamos podrá ser accedido prácticamente por cualquier componente de software del mercado actual.

Esta sección cubre a fondo la automatización y las formas en las que un cliente puede acceder a los servidores de automatización: interfaces, variantes, disp-interfaces e interfaces duales. La próxima sección aplica esta información para la creación y uso de un servidor de automatización interno al proceso.

Servidores de automatización internos al proceso

El primer servidor de automatización que vamos a desarrollar es un simple servidor interno al proceso. Debemos recordar del Capítulo 2 que un servidor interno al proceso se ejecuta en el mismo espacio de direcciones que la aplicación cliente.

Ejemplo: Servidor de conversión de unidades de medida

Este servidor de automatización será capaz de realizar conversiones entre diferentes unidades de medida. Podrá convertir entre metros cuadrados, centímetros cuadrados, yardas cuadradas, pies cuadrados, pulgadas cuadradas, kilómetros cuadrados, millas cuadradas y acres. Ya hemos visto antes el método responsable de la conversión:

```
function Convert(Quantity: Double; InUnit: Integer;
  OutUnit: Integer): Double; safecall;
```

En pocas palabras, *Convert* convierte una cantidad dada de la unidad de medida especificada en *InUnit* a la unidad de medida especificada en *OutUnit*. El código que ejecuta la conversión es muy simple, y se muestra en el siguiente ejemplo:

```
const
  auSquareMeters = $00000000;
  auSquareCentimeters = $00000001;
  auSquareYards = $00000002;
  auSquareFeet = $00000003;
  auSquareInches = $00000004;
  auSquareKilometers = $00000005;
  auSquareMiles = $00000006;
  auAcres = $00000007;
function TUnitAuto.Convert(Quantity: Double; InUnit: Integer;
  OutUnit: Integer): Double;
```

```
const
  AreaFactor: array[auSquareMeters..auAcres] of Double =
    (10000.0, 1.0, 8361.2736, 929.0304, 6.4516, 10000000000.0,
     25899881103.4, 40468726.0987);
begin
  Result := Quantity * AreaFactor[InUnit] /  AreaFactor[OutUnit];
end;
```

El *array AreaFactor* contiene los factores necesarios para convertir una unidad dada de medida a centímetros cuadrados. Por ejemplo, un metro cuadrado son 10.000 centímetros cuadrados, y un acre son 40.468.726,0987 centímetros cuadrados. La función *Convert* simplemente convierte de *InUnit* a centímetros cuadrados, y después de centímetros cuadrados a la unidad especificada en *OutUnit*.

Ahora que ya conocemos como funciona el proceso, creemos un servidor de automatización que ofrezca esta funcionalidad.

1. Seleccione *File | New* del menú principal de Delphi. En la página *ActiveX* del Depósito de Objetos, seleccione el icono *ActiveX Library*. Hasta ahora, los pasos son idénticos a los realizados cuando creamos un servidor COM interno al proceso.
2. Seleccionemos nuevamente *File | New*, y en la página *ActiveX* del Depósito de Objetos seleccionemos *Automation Object*. Delphi mostrará el Asistente para Objetos de Automatización (*Automation Object Wizard*), como muestra la Figura 4.2.

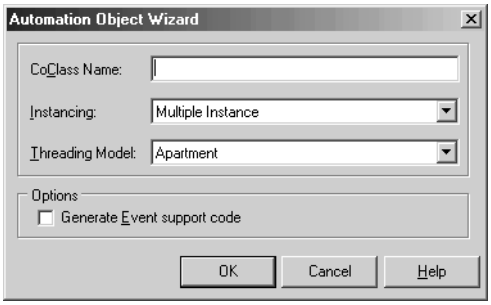


Figura 4.2: Asistente de Objetos de Automatización

3. Introduzca el nombre de la clase, *AreaUnitConverter*, y pulse OK. Delphi creará una unidad para la clase, y además una biblioteca de tipos. El Editor de Bibliotecas de Tipos aparecerá automáticamente.
4. Seleccione *IAreaUnitConverter* en la lista de objetos. Observe en el panel de información que la interfaz *IAreaUnitConverter* desciende de *IDispatch* (Figura 4.3).

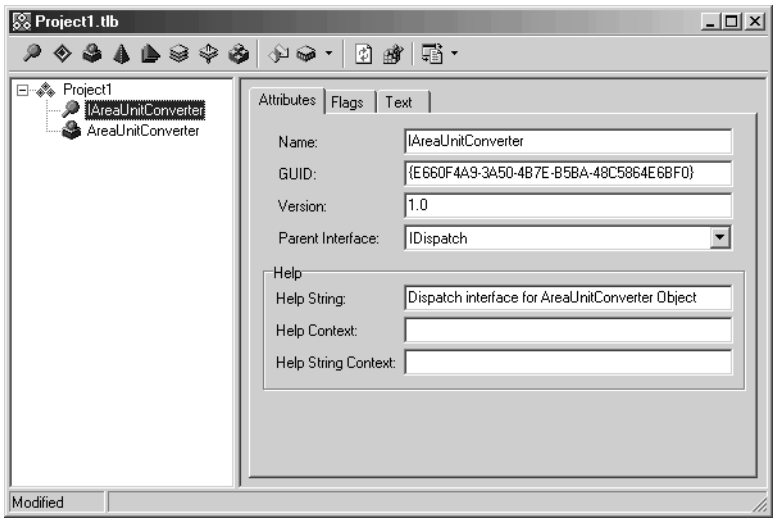


Figura 4.3: Editor de Bibliotecas de Tipos.

Ya conocemos qué hace el método *Convert*, añadámoslo a la interfaz:

5. Pulse sobre el icono '*New Method*' de la barra de herramientas, y asigne al nuevo método el nombre *Convert*. Observe en el panel de información que Delphi asigna a este método un dispid de 1.
6. Pulse sobre la página '*Parameters*', y asigne *double* como tipo de retorno. A continuación añada un parámetro llamado *Quantity*, de tipo *double*, y luego dos parámetros enteros, llamados *InUnit* y *OutUnit*. Al finalizar, la pantalla debe lucir de modo similar a la Figura 4.4.

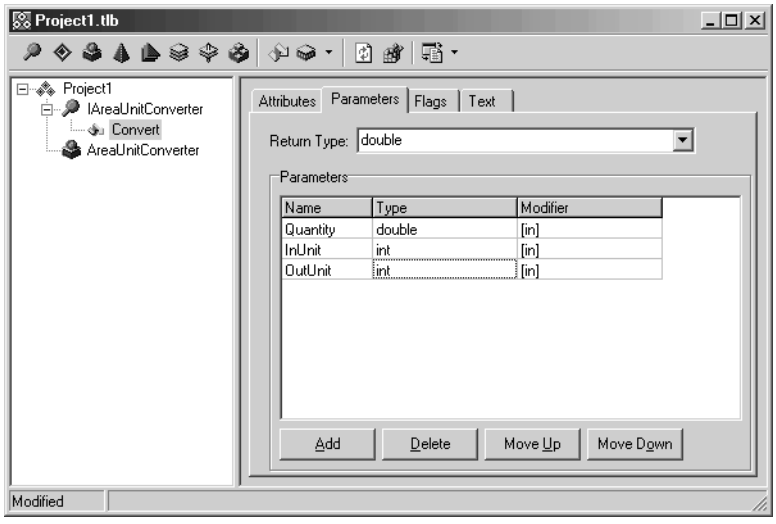


Figura 4.4: El Editor de Bibliotecas de Tipos después de añadir un método.

Ahora, en lugar de obligar a los usuarios del servidor de automatización a recordar que 0 significa metros cuadrados, etc., añadamos una enumeración a la biblioteca de tipos:

7. Pulse el botón '*New Enumeration*' en la barra de herramientas. Llame al enumerador *AreaUnit*. Luego pulse el botón '*New Const*' y añada una constante llamada *auSquareMeters*. Repita el proceso para *auSquareCentimeters*, *auSquareYards*, *auSquareFeet*, *auSquareInches*, *auSquareKilometers*, *auSquareMiles*, y *auAcres*. Delphi automáticamente asignará los valores del cero al siete a las constantes de la enumeración.
8. Pulse el botón '*Refresh Implementation*' en la barra de herramientas para asegurar que el código fuente muestra las adiciones hechas a la biblioteca de tipos. Y ya puede cerrar el Editor de Bibliotecas de Tipos.
9. Guarde la unidad como *AreaUnit.pas*, y el proyecto como *UnitSrv.dpr*. Ahora podemos completar el cuerpo del método *Convert*:

```
function TUnitAuto.Convert(Quantity: Double; InUnit: Integer;
  OutUnit: Integer): Double;
const
  AreaFactor: array [auSquareMeters..auAcres] of Double =
    (10000.0, 1.0, 8361.2736, 929.0304, 6.4516, 10000000000.0,
     25899881103.4, 40468726.0987);
begin
  Result := Quantity * AreaFactor[InUnit] /  AreaFactor[OutUnit];
end;
```

Eso es todo lo que necesitamos para completar este servidor. Compile para estar seguro que no ha cometido errores de sintaxis, y después seleccione la opción *Run | Register ActiveX Server* para registrar el servidor en el Registro de Windows.

No nos molestaremos en listar el código fuente del servidor, ya que hemos visto la mayoría de éste. Ahora podemos concentrarnos en crear una aplicación cliente para acceder al servidor.

Nota

Ud. podrá estar pensando que este proceso parece demasiado sencillo. Aún no hemos presionado Ctrl+Shift+G para generar el GUID de la interfaz IAreaUnitConverter. El hecho es que hemos realizado todo el trabajo requerido para escribir un servidor de automatización de interfaz dual. El asistente genera de forma automática el GUID.

Igualmente habrá observado que ha sido aún más sencillo crear un servidor de automatización COM como el que hemos creado aquí, que crear un servidor COM "plano" como el creado en el Capítulo 2. Cuando yo desarrollo código COM, casi siempre creo servidores de automatización. Son fáciles de crear y mucho más flexibles que los objetos COM estándar.

CreateOleObject y GetActiveOleObject

Con anterioridad hemos mostrado cómo *CreateOleObject* es usado para arrancar una nueva instancia de un servidor de automatización. *CreateOleObject* está declarado en *ComObj.pas* de la siguiente forma:

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

Como podemos ver, esta función primero obtiene el GUID de la clase que estamos creando. Seguidamente, pasa el GUID a *CoCreateInstance*, que hemos examinado en el Capítulo 2. *ProgIDToClassID* es básicamente un envoltorio alrededor de la API de Windows *CLSIDFromProgID*. Así que, en pocas palabras, *CreateOleObject* funciona de modo similar a *CreateComObject*, excepto que *CreateOleObject* recibe como parámetro un nombre de clase en lugar de un GUID.

Sin embargo, *CreateOleObject* siempre crea una nueva instancia de un servidor. ¿Y si deseáramos conectarnos a una instancia de un servidor que ya esté en ejecución? O podríamos desconocer si el servidor está ejecutándose o no. En ese caso tendríamos que iniciarlo si no está ejecutándose, y conectarnos en caso contrario.

GetActiveOleObject puede ser utilizado para obtener una referencia a un servidor que se está ejecutando en memoria. En el Capítulo 2 hemos mencionado la Tabla de Objetos en Ejecución de Windows, en la que se hace un seguimiento de todos los objetos COM activos. *GetActiveOleObject* accede a la Tabla de Objetos en Ejecución para verificar si un servidor dado está ejecutándose. Si es así, la función devuelve una referencia a la interfaz *IDispatch* del servidor. Si el servidor no está ejecutándose, *GetActiveOleObject* lanza una excepción.

Con esta información, el procedimiento siguiente iniciará una copia de Word si éste no está ejecutándose, o se conectará a una copia de Word si está ejecutándose.

```
procedure StartOrLinkToWord;
var
  V: Variant;
begin
  try
    V := GetActiveOleObject('Word.Basic');
  except
    V := CreateOleObject('Word.Basic');
  end;
  // Hacemos algo con V aqu□...
end;
```

Ejemplo: Cliente del servidor de conversión de unidades

Ahora podemos concentrarnos en escribir un programa cliente para acceder al servidor de automatización recién creado. La Figura 4.5 muestra el formulario principal de la aplicación *UnitCli* en tiempo de ejecución.

Como podemos ver, la interfaz de usuario de este programa está formada por seis botones. La columna de botones de la izquierda crea y accede a un servidor de automatización a través de las tres formas de acceso estudiadas: interfaz, variante y disp-interfaz. Los botones de la columna derecha operan del mismo modo que sus homólogos de la izquierda, con la excepción de que ejecutan 100.000 llamadas de una vez para propósitos de medición de tiempo.

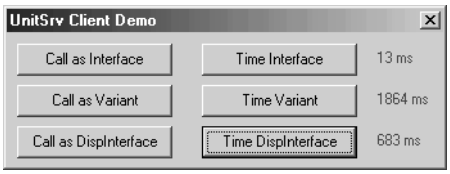


Figura 4.5: *UnitCli* ejecuta pruebas de tiempo para los diferentes métodos de acceso.

El código requerido para acceder al servidor es pequeño en los tres casos. Veamos primero un ejemplo de código para acceder al servidor a través de una interfaz.

```
procedure TForm1.btnInterfaceClick(Sender: TObject);
var
  I: IUnitAuto;
begin
  I := CoUnitAuto.Create;
  ShowMessage(FloatToStr(I.Convert(1.0, 0, 1)));
end;
```

Hemos visto un código similar en el Capítulo 2. La llamada a *CoUnitAuto* crea una instancia del servidor de automatización, y la llamada *I.Convert* convierte 1.0 metros cuadrados a centímetros cuadrados. No olvidemos que, debido al conteo de referencias, el servidor es automáticamente destruido al final del procedimiento.

Ahora echemos un vistazo a cómo acceder al mismo servidor a través de una variante:

```
procedure TForm1.btnVariantClick(Sender: TObject);
var
  V: Variant;
begin
  V := CreateOleObject('UnitSrv.UnitAuto');
  ShowMessage(FloatToStr(V.Convert(1.0, 0, 1)));
end;
```

Este código es muy similar al código requerido para acceder al servidor a través de una interfaz. La primera línea de código realiza una llamada a *CreateOleObject* para

crear una instancia del servidor. Probablemente se esté preguntando cómo supimos que *UnitSrv.UnitAuto* era el nombre de la clase a pasar a *CreateOleObject*. Delphi genera esta cadena concatenando el nombre del servidor, un punto, y el nombre de la co-clase.

La biblioteca de tipos del servidor *UnitSrv* se muestra en la Figura 4.6. Como podemos ver, el nombre de la biblioteca de tipos es *UnitSrv*. La co-clase está seleccionada en la lista de objetos, y su nombre es *UnitAuto*. Por lo tanto, el nombre de clase para el objeto es *UnitSrv.UnitAuto*.

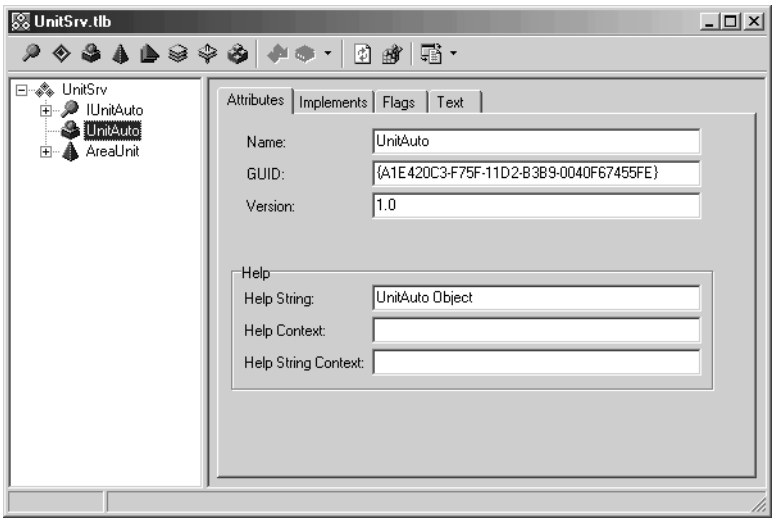


Figura 4.6: Biblioteca de tipos de *UnitSrv*.

El segundo método que podemos usar para determinar el nombre del servidor es buscar el GUID en el Registro de Windows. En el Registro encontraremos una entrada semejante a la de la Figura 4.7.

No podremos utilizar estos métodos si estamos usando un servidor de automatización creado por otros, que puede no haber sido creado con Delphi, o si no conocemos el GUID del objeto en cuestión. En estos casos, la documentación del servidor debe darnos el nombre de la clase disponible.

Una vez que *CreateOleObject* almacena una referencia al servidor de automatización en la variante *V*, podemos realizar llamadas al servidor como si estuviéramos usando una interfaz. Como hemos explicado con anterioridad, el mecanismo interno para realizar llamadas a variantes es muy diferente (y más lento) que el usado para interfaces.

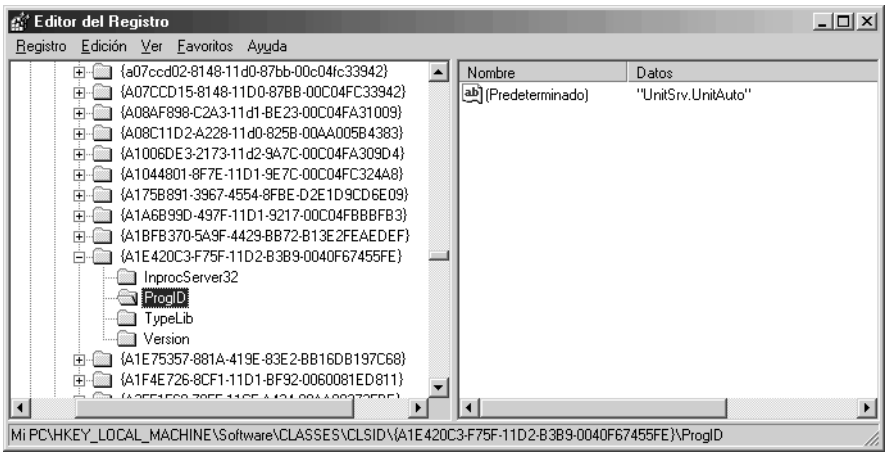


Figura 4.7: Entrada de *UnitAuto* en el Registro.

El último método para acceder al servidor de automatización es a través de una disp-interfaz.

```
procedure TForm1.btnDispInterfaceClick(Sender: TObject);
var
  DI: IUnitAutoDisp;
begin
  DI := CoUnitAuto.Create as IUnitAutoDisp;
  ShowMessage(FloatToStr(DI.Convert(1.0, 0, 1)));
end;
```

Como podemos ver, este código es casi idéntico al usado para las interfaces. La única diferencia es que la interfaz obtenida a través de *CoUnitAuto.Create* es convertida a una disp-interfaz.

El Listado 4.1 muestra el código para el formulario principal de la aplicación cliente.

Listado 4.1: Aplicación UnitCli - *MainForm.pas*

```
unit MainForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  StdCtrls;
type
  TForm1 = class(TForm)
    btnVariant: TButton;
    btnInterface: TButton;
    btnDispInterface: TButton;
    btnTimeInterface: TButton;
```

```
    btnTimeVariant: TButton;
    btnTimeDispInterface: TButton;
    lblInterface: TLabel;
    lblVariant: TLabel;
    lblDispInterface: TLabel;
    procedure btnVariantClick(Sender: TObject);
    procedure btnInterfaceClick(Sender: TObject);
    procedure btnDispInterfaceClick(Sender: TObject);
    procedure btnTimeInterfaceClick(Sender: TObject);
    procedure btnTimeVariantClick(Sender: TObject);
    procedure btnTimeDispInterfaceClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
uses
  ComObj, UnitSrv_TLB;
{$R *.DFM}
procedure TForm1.btnVariantClick(Sender: TObject);
var
  V: Variant;
begin
  V := CreateOleObject('UnitSrv.UnitAuto');
  ShowMessage(FloatToStr(V.Convert(1.0, 0, 1)));
end;
procedure TForm1.btnInterfaceClick(Sender: TObject);
var
  I: IUnitAuto;
begin
  I := CoUnitAuto.Create;
  ShowMessage(FloatToStr(I.Convert(1.0, 0, 1)));
end;
procedure TForm1.btnDispInterfaceClick(Sender: TObject);
var
  DI: IUnitAutoDisp;
begin
  DI := CoUnitAuto.Create as IUnitAutoDisp;
  ShowMessage(FloatToStr(DI.Convert(1.0, 0, 1)));
end;
procedure TForm1.btnTimeInterfaceClick(Sender: TObject);
var
  I: IUnitAuto;
  Count: Integer;
  Dbl: Double;
  T1, T2: DWord;
begin
  I := CoUnitAuto.Create;
  T1 := GetTickCount;
```

```
  for Count := 1 to 100000 do
    Dbl := I.Convert(1.0, 0, 1);
  T2 := GetTickCount;
  lblInterface.Caption := IntToStr(T2 - T1) + ' ms';
end;
procedure TForm1.btnTimeVariantClick(Sender: TObject);
var
  V: Variant;
  Count: Integer;
  Dbl: Double;
  T1, T2: DWord;
begin
  V := CreateOleObject('UnitSrv.UnitAuto');
  T1 := GetTickCount;
  for Count := 1 to 100000 do
    Dbl := V.Convert(1.0, 0, 1);
  T2 := GetTickCount;
  lblVariant.Caption := IntToStr(T2 - T1) + ' ms';
end;
procedure TForm1.btnTimeDispInterfaceClick(Sender: TObject);
var
  DI: IUnitAutoDisp;
  Count: Integer;
  Dbl: Double;
  T1, T2: DWord;
begin
  DI := CoUnitAuto.Create as IUnitAutoDisp;
  T1 := GetTickCount;
  for Count := 1 to 100000 do
    Dbl := DI.Convert(1.0, 0, 1);
  T2 := GetTickCount;
  lblDispInterface.Caption := IntToStr(T2 - T1) + ' ms';
end;
end.
```

Esta aplicación puede acceder a nuestro servidor a través de una interfaz, una disp-interfaz, o *IDispatch*. En aplicaciones reales, simplemente elegiremos uno de los tres métodos. Nuestro objetivo al mostrarles los tres métodos son**:**

1. Mostrar cómo usar cada método en nuestras aplicaciones.
2. Ejecutar algunas mediciones de tiempo para mostrar la eficiencia relativa de cada método.

Si consulta de nuevo la Figura 4.5, verá los resultados de la medición en mi ordenador. Estos valores representan el tiempo requerido al llamar al método *Convert* 100.000 veces.

Como puede observarse, las interfaces son (con ventaja) la forma más rápida de llamar a un método de un servidor de automatización. Las variantes son más lentas en más de un orden de magnitud, y las disp-interfaces están en un punto intermedio.

Ahora ya debemos estar familiarizados con los servidores de automatización internos al proceso. Sin embargo, éstos no son siempre apropiados. Si deseamos crear un servidor COM que se ejecute en un espacio de direcciones diferente al de la aplicación cliente, necesitaremos crear un servidor de automatización externo al proceso. La próxima sección nos guiará a través de la creación y uso de un servidor de automatización externo al proceso.

Servidores de automatización externos al proceso

En el Capítulo 2 examinamos la necesidad de utilizar la serialización (*marshaling*) para los servidores COM externos al proceso. Los tipos de datos que son compatibles con automatización son *Smallint, Integer, Single, Double, Currency, TDateTime, WideString, IDispatch, SCODE, WordBool, OleVariant, IUnknown, Shortint* y *Byte*. Como veremos en su momento, Delphi ofrece soporte para el *marshaling* de imágenes, listas de cadenas y fuentes a través de las interfaces *IPicture*, *IStrings* e *IFonts*.

Los servidores de automatización externos al proceso son deseables en aquellos casos donde el servidor de automatización debe actuar como aplicación independiente, y son imprescindibles en los casos en que deseemos acceder al servidor desde un ordenador remoto usando DCOM (DCOM se estudia en el Capítulo 6, "DCOM").

HResult y Safecall

Todos los métodos de un servidor de automatización deben devolver un *HResult*, que indica si el método ha tenido éxito o no. Cualquier otro valor que se devuelva deberá ser pasado a través de un parámetro **out**, según lo visto en el Capítulo 3.

Esto no suena lógico, dado que el método *Convert* que hemos usado en *UnitSrv* devuelve un *Double*. Una vez más, Delphi nos protege de alguna de las complejidades de COM. En un entorno tradicional de programación, necesitaríamos escribir el método *Convert* del siguiente modo:

```
function Convert(Quantity: Double; InUnit, OutUnit: Integer;
  out Result: Double): HResult; stdcall;
```

El convenio de llamada **safecall** permite escribir código como si el método tuviese la siguiente forma:

```
function Convert(Quantity: Double; InUnit, OutUnit: Integer): Double;
stdcall;
```

La directiva **safecall** también le indica a Delphi que ejecute otra importante función entre bastidores. La directiva **safecall** del lado del servidor COM le indica a Delphi que automáticamente encierre el cuerpo de todos los métodos dentro de

un bloque **try**/**except**. Por ejemplo, si escribimos el siguiente código en un servidor de automatización:

```
function TMyServer.DoSomething: Integer;
begin
 Result := someFunctionThatReturnsAnInteger;
end;
```

Delphi compilará el equivalente del siguiente código en nuestra aplicación:

```
function TMyServer.DoSomething(out Ret: Integer): HResult;
begin
  try
    Ret := someFunctionThatReturnsAnInteger;
    Result := S_OK;
  except
    Result := E_UNEXPECTED;
  end;
end;
```

En otras palabras, Delphi impide que alguna excepción se escape de un método de un servidor de automatización. En lugar de ello, Delphi transforma la excepción que se devuelve a la aplicación cliente como un *HResult*.

Del lado del cliente, **safecall** hace que el cliente verifique el *HResult* devuelto por la llamada al método y lanza una excepción de Delphi si la llamada al método devuelve un código de error.

Para entender por qué es deseable que esto ocurra, tenga en cuenta el hecho de que el servidor de automatización podría no tener interfaz de usuario. Podría, por ejemplo, tener una ventana principal oculta, o no tener ninguna. Si un mensaje resultante del lanzamiento de una excepción se lanzara de pronto, sería en el mejor de los casos desconcertante para el usuario final. Más aún, como aprenderemos en el Capítulo 6, los objetos de automatización no siempre tienen que residir en el mismo ordenador en el que residen los clientes que lo utilizan. Si el ordenador que aloja el objeto de automatización reside en un lugar distante, la aplicación cliente no tiene forma de conocer cuando una excepción se muestra en la pantalla del servidor.

Nota

Serializando cadenas, fuentes e imágenes

Como hemos mencionado brevemente antes, Delphi proporciona soporte para el marshaling de imágenes, listas de cadenas y fuentes. Se trata de una característica muy bienvenida, ya que Delphi hace un uso considerable de listas de cadenas y fuentes en la VCL. No mostraremos aquí las declaraciones de estas interfaces, pero si está interesado en echarles un vistazo, IStrings está declarada en StdVCL.pas, e IFont e IPicture están declaradas en ActiveX.pas. El programa de ejemplo que se presenta en la próxima sección muestra cómo usar IStrings e IFont en nuestras aplicaciones.

Automatizando una aplicación existente

En esta sección vamos a tomar una aplicación existente y añadirle capacidades de automatización. La aplicación es un programa muy sencillo, que presenta un editor multilíneas y dos botones sobre la ventana principal. Un botón cambia el tipo de letra del editor, y el otro su color. Nada excepcional, pero servirá para ilustrar los puntos que deseamos presentar.

Existen muchos casos en los que podríamos desear automatizar una aplicación existente. Por ejemplo, suponga que hemos escrito una aplicación de contabilidad (tipo ContaWin) que ofrece la posibilidad de buscar precios de productos *online*. Si se automatiza la funcionalidad de búsqueda, Usted (u otro programador) podrá posteriormente crear una aplicación cliente que ordene a la aplicación de contabilidad descargar los últimos precios a intervalos de tiempo regulares.

En este ejemplo aprenderemos no sólo cómo añadir capacidades de automatización a una aplicación existente, sino también cómo pasar listas de cadenas, colores y fuentes como parámetros a un método de automatización.

Nota

En la actualidad, muchos de los servidores de automatización externos al proceso, a diferencia del ejemplo que mostramos aquí, son programas completos en sí mismos. Por ejemplo, Microsoft Word es un servidor de automatización. Word es un programa muy completo, y proporciona la mayoría de su funcionalidad a otras aplicaciones a través de automatización.

El Listado 4.2 muestra el código fuente para el formulario principal de la aplicación MemoDemo.

Listado 4.2: Aplicación MemoDemo - *MainForm.pas*

```
unit MainForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  StdCtrls;
type
  TForm1 = class(TForm)
    Memo1: TMemo;
    btnFont: TButton;
    btnColor: TButton;
    FontDialog1: TFontDialog;
    ColorDialog1: TColorDialog;
    procedure btnFontClick(Sender: TObject);
    procedure btnColorClick(Sender: TObject);
  private
```

```
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.btnFontClick(Sender: TObject);
begin
  if FontDialog1.Execute then
    Memo1.Font.Assign(FontDialog1.Font);
end;
procedure TForm1.btnColorClick(Sender: TObject);
begin
   if ColorDialog1.Execute then
     Memo1.Color := ColorDialog1.Color;
end;
end.
```

Como podemos ver, el código fuente de la aplicación es casi insignificante. La Figura 4.8 muestra el formulario principal de *MemoDemo* en tiempo de ejecución.

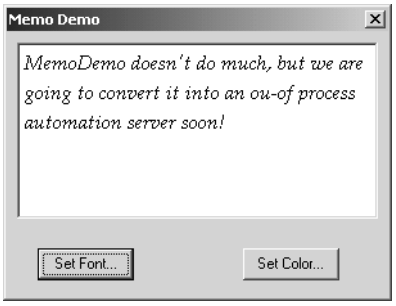


Figura 4.8: *MemoDemo* no hace mucho, pero es un buen punto de partida.

Escriba algún texto dentro del editor. Pulse el botón '*Set Font*' para cambiar el tipo de letra que utiliza el editor, o el botón '*SetColor*' para cambiar su color de fondo.

Añadiendo un objeto de automatización

Ahora que hemos visto *MemoDemo* – que, mucho me temo, no ganará ningún premio de la industria, veamos qué es necesario para añadir capacidades de automatización a éste.

Una de las cosas más importantes que puede hacer cuando esté añadiendo capacidades de automatización a una aplicación existente es hacer uso del código que ya forma parte de la aplicación. Idealmente, los cuerpos de nuestros métodos de automatización deben contener una sola línea de código, que contiene una llamada a una función ya existente en la aplicación principal.

Si estamos desarrollando una aplicación teniendo en mente la automatización, no es difícil tener esto en cuenta desde el principio. Sin embargo, supongamos que la automatización era algo que estaba muy lejos de nuestras mentes cuando escribimos *MemoDemo*. La forma más rápida y sencilla de preparar una aplicación para la automatización es añadirle las funciones de soporte necesarias. Por ejemplo, en el caso de *MemoDemo*, deseamos ser capaces de leer y controlar el tipo de letra, el color y el texto del editor desde otra aplicación. Como debe ser obvio, debemos añadir los métodos *GetColor*, *SetColor*, *GetFont*, *SetFont*, *GetText* y *SetText* al formulario principal. El Listado 4.3 muestra el resultado de esta modularización.

Listado 4.3: Aplicación MemoSrv - *MainForm.pas*

```
unit MainForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  StdCtrls;
type
  TForm1 = class(TForm)
    Memo1: TMemo;
    btnFont: TButton;
    btnColor: TButton;
    FontDialog1: TFontDialog;
    ColorDialog1: TColorDialog;
    procedure btnFontClick(Sender: TObject);
    procedure btnColorClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
    function GetColor: TColor;
    procedure SetColor(AColor: TColor);
    function GetFont: TFont;
    procedure SetFont(AFont: TFont);
    function GetText: TStrings;
    procedure SetText(AText: TStrings);
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.btnFontClick(Sender: TObject);
begin
  if FontDialog1.Execute then
    Memo1.Font.Assign(FontDialog1.Font);
end;
procedure TForm1.btnColorClick(Sender: TObject);
begin
```

```
  if ColorDialog1.Execute then
    Memo1.Color := ColorDialog1.Color;
end;
procedure TForm1.SetColor(AColor: TColor);
begin
  Memo1.Color := AColor;
end;
procedure TForm1.SetFont(AFont: TFont);
begin
  Memo1.Font.Assign(AFont);
end;
procedure TForm1.SetText(AText: TStrings);
begin
  Memo1.Lines.Assign(AText);
end;
function TForm1.GetColor: TColor;
begin
  Result := Memo1.Color;
end;
function TForm1.GetFont: TFont;
begin
  Result := Memo1.Font;
end;
function TForm1.GetText: TStrings;
begin
  Result := Memo1.Lines;
end;
end.
```

Ahora que el formulario principal soporta las funciones de acceso necesarias, es hora de añadir el objeto de automatización a *MemoSrv*. Seleccione *FileNew* en el menú principal de Delphi. Pulse sobre la página *ActiveX* del Depósito de Objetos, y seleccione *Automation Object*. Llame al objeto *MemoIntf*, y pulse OK. Delphi crea un nuevo fichero fuente para *MemoIntf* y presenta el Editor de Bibliotecas de Tipos.

Vamos a añadir tres propiedades a la biblioteca de tipos: *Color*, *Font* y *Text*.

1. Pulse el botón *New Property*. Si se despliega un menú contextual, seleccione *Read/Write*. Llame a la propiedad *Color*, y asígnele el tipo *OLE_COLOR*.
2. Añada una nueva propiedad de lectura y escritura de nombre *Font*, y asígnele el tipo *IFontDisp*. No encontrará *IFontDisp* en la lista de tipos, pero el Editor de Bibliotecas de tipos aceptará la declaración si Ud. la escribe.
3. Añada una nueva propiedad de lectura y escritura, llámela *Text*, y asígnele el tipo *IStrings*.
4. Pulse el botón *Refresh Implementation* y cierre el Editor de Bibliotecas de Tipos.
5. Guarde el fichero fuente como *MemoIntf.pas*.

6. Añada *ActiveX*, *StdVCL* y *AxCtrls* a la cláusula **uses** de la sección **interface** de la unidad, ya que en éstas están definidas *IFont* e *IStrings*, así como los procedimientos *GetOLExx*.
7. Añada *MainForm* a la cláusula **uses** de la sección **implementation**.
8. Reprograme los métodos de *MemoIntf* de forma tal que llamen a sus métodos correspondientes en *TForm1* (ver Listado 4.4).

Ahora ya puede compilar el programa. Ejecútelo para registrar el servidor de automatización en el registro de Windows, y salga de la aplicación.

Nota

Recuerde que los servidores COM externos al proceso se registran en el Registro de Windows cada vez que son ejecutados. Para registrar el servidor externo al proceso sin ejecutar la aplicación, puede ejecutar MemoSrv.exe /regserver. Para desregistrarlo, se debe ejecutar MemoSrv.exe /unregserver.

El código fuente para *MemoIntf* se muestra en el Listado 4.4.

Listado 4.4: Aplicación MemoSrv – *MemoIntf.pas*

```
unit MemoIntf;
interface
uses
  ComObj, ActiveX, AXCtrls, StdVCL, MemoSrv_TLB;
type
  TMemoIntf = class(TAutoObject, IMemoIntf)
  protected
    function Get_Color: OLE_COLOR; safecall;
    procedure Set_Color(Value: OLE_COLOR); safecall;
    function Get_Font: IFontDisp; safecall;
    function Get_Text: IStrings; safecall;
    procedure Set_Font(const Value: IFontDisp); safecall;
    procedure Set_Text(const Value: IStrings); safecall;
    { Protected declarations }
  end;
implementation
uses ComServ, MainForm;
function TMemoIntf.Get_Color: OLE_COLOR;
begin
  Result := Form1.GetColor;
end;
procedure TMemoIntf.Set_Color(Value: OLE_COLOR);
begin
  Form1.SetColor(Value);
end;
function TMemoIntf.Get_Font: IFontDisp;
begin
  GetOleFont(Form1.GetFont, Result);
```

```
end;
function TMemoIntf.Get_Text: IStrings;
begin
  GetOleStrings(Form1.GetText, Result);
end;
procedure TMemoIntf.Set_Font(const Value: IFontDisp);
begin
  SetOleFont(Form1.GetFont, Value);
end;
procedure TMemoIntf.Set_Text(const Value: IStrings);
begin
  SetOleStrings(Form1.GetText, Value);
end;
initialization
  TAutoObjectFactory.Create(ComServer, TMemoIntf, Class_MemoIntf,
    ciMultiInstance, tmApartment);
end.
```

Creando un cliente de automatización

Ahora que tenemos una aplicación a automatizar, vamos a concentrarnos en desarrollar una aplicación cliente que la automatice.

En la mayoría de los casos en que se desea utilizar un servidor de automatización, no se dispone del código fuente de las interfaces que proporciona el servidor. En este libro hemos desarrollado tanto el servidor como los clientes, así que el código fuente estaba disponible. Pero, ¿qué pasaría si *MemoSrv* hubiera sido creado por otro programador?

Vamos a mostrar cómo podemos importar bibliotecas de tipos en Delphi y hacer que éste reconstruya automáticamente el código fuente necesario.

Cree una aplicación nueva en Delphi, y seguidamente seleccione *Project Import Type Library* desde el menú principal de Delphi. Aparecerá la pantalla que se muestra en la Figura 4.9.

Desplácese a través de la lista hasta encontrar *MemoSrv (Version 1.0)*. Si no aparece en la lista, pulse el botón *Add...*, y seleccione el fichero *MemoSrv.exe* en el directorio donde éste se encuentre.

Seleccione *MemoSrv (Version 1.0)* y pulse OK. Delphi generará una biblioteca de importación llamada *MemoSrv_TLB.pas* para el servidor de automatización *MemoSrv*. Por defecto, el fichero será colocado en el directorio *Imports* del disco duro. Si ha instalado Delphi con la configuración por defecto, este directorio será *C:\Archivos de programa\Borland\DelphiX\Imports*, donde *DelphiX* representa la versión de Delphi que esté utilizando. El Listado 4.5 muestra la biblioteca de tipos importada para *MemoSrv*.

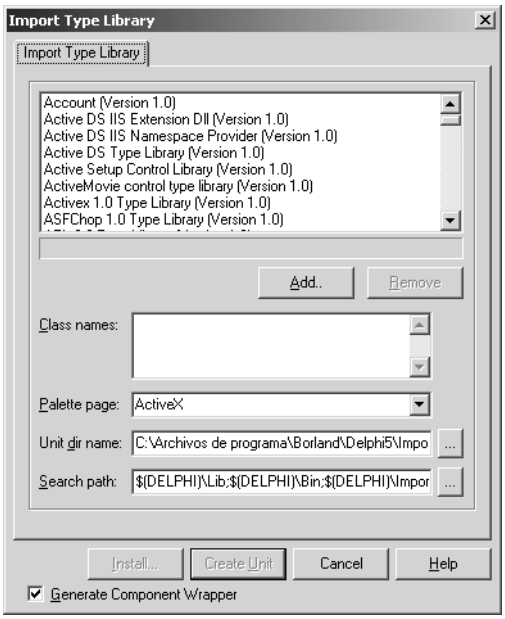


Figura 4.9: Importando una biblioteca de tipos.

Listado 4.5: Aplicación MemoCli - *MemoSrv_TLB.pas*

```
unit MemoSrv_TLB;
// ************************************************************************ //
// WARNING                                                                  //
// □□□-                                                                     //
// The types declared in this file were generated from data read from       //
// a Type Library. If this type library is explicitly or indirectly         //
// (via another type library referring to this type library)                //
// re-imported, or the 'Refresh' command of the Type Library Editor         //
// activated while editing the Type Library, the contents of this file      //
// will be regenerated and all manual modifications will be lost.           //
// ************************************************************************ //
// PASTLWTR : $Revision:   1.11.1.75  $
// File generated on 5/15/01 10:46:43 AM from Type Library described
below.
// ************************************************************************ //
// Type Lib: C:\Book\samples\Chap04\MemoSrv\MemoSrv.tlb
// IID\LCID: {A1E420C7-F75F-11D2-B3B9-0040F67455FE}\0
// Helpfile:
// HelpString: MemoSrv Library
// Version:    1.0
// ************************************************************************ //
interface
uses Windows, ActiveX, Classes, Graphics, OleCtrls, StdVCL;
// ************************************************************************ //
// GUIDS declared in the TypeLibrary. Following prefixes are used:          //
```

```
//   Type Libraries     : LIBID_xxxx                                    //
//   CoClasses          : CLASS_xxxx                                    //
//   DISPInterfaces     : DIID_xxxx                                     //
//   Non-DISP interfaces: IID_xxxx                                      //
// ********************************************************************** //
const
  LIBID_MemoSrv: TGUID = '{A1E420C7-F75F-11D2-B3B9-0040F67455FE}';
  IID_IMemoIntf: TGUID = '{A1E420C8-F75F-11D2-B3B9-0040F67455FE}';
  CLASS_MemoIntf: TGUID = '{A1E420CA-F75F-11D2-B3B9-0040F67455FE}';
type
// ********************************************************************** //
// Forward declaration of interfaces defined in Type Library            //
// ********************************************************************** //
  IMemoIntf = interface;
  IMemoIntfDisp = dispinterface;
// ********************************************************************** //
// Declaration of CoClasses defined in Type Library                     //
// (NOTE: Here we map each CoClass to its Default Interface)            //
// ********************************************************************** //
  MemoIntf = IMemoIntf;
// ********************************************************************** //
// Interface: IMemoIntf
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {A1E420C8-F75F-11D2-B3B9-0040F67455FE}
// ********************************************************************** //
  IMemoIntf = interface (IDispatch)
    ['{A1E420C8-F75F-11D2-B3B9-0040F67455FE}']
    function Get_Color: OLE_COLOR; safecall;
    procedure Set_Color(Value: OLE_COLOR); safecall;
    function Get_Font: IFontDisp; safecall;
    procedure Set_Font(const Value: IFontDisp); safecall;
    function Get_Text: IStrings; safecall;
    procedure Set_Text(const Value: IStrings); safecall;
    property Color: OLE_COLOR read Get_Color write Set_Color;
    property Font: IFontDisp read Get_Font write Set_Font;
    property Text: IStrings read Get_Text write Set_Text;
  end;
// ********************************************************************** //
// DispIntf:  IMemoIntfDisp
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {A1E420C8-F75F-11D2-B3B9-0040F67455FE}
// ********************************************************************** //
  IMemoIntfDisp = dispinterface
    ['{A1E420C8-F75F-11D2-B3B9-0040F67455FE}']
    property Color: OLE_COLOR dispid 2;
    property Font: IFontDisp dispid 1;
    property Text: IStrings dispid 3;
  end;
  CoMemoIntf = class
    class function Create: IMemoIntf;
    class function CreateRemote(const MachineName: string): IMemoIntf;
  end;
```

```
implementation
uses ComObj;
class function CoMemoIntf.Create: IMemoIntf;
begin
  Result := CreateComObject(CLASS_MemoIntf) as IMemoIntf;
end;
class function CoMemoIntf.CreateRemote(const MachineName: string):
IMemoIntf;
begin
  Result := CreateRemoteComObject(MachineName, CLASS_MemoIntf) as
IMemoIntf;
end;
end.
```

Añada *MemoSrv_TLB* a la cláusula **uses** del formulario principal, y compile el proyecto para asegurar que Delphi encuentra la unidad importada. Si se produce un error de compilación al intentar compilar *MemoSrv_TLB*, deberá comprobar sus opciones del entorno (menú *Tools | Environment Options*) para asegurarse de que *$(Delphi)\Imports* forma parte de las rutas de bibliotecas (ver Figura 4.10).

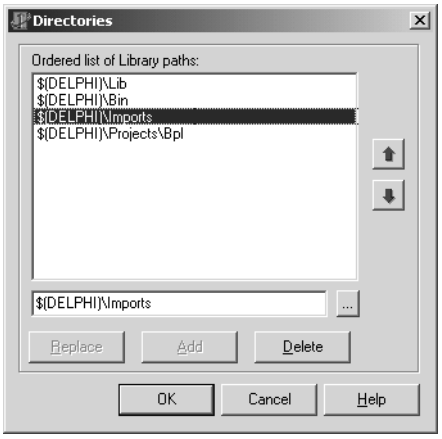


Figura 4.10: Verificando que $(DELPHI)\Imports se encuentra en la lista de rutas de bibliotecas.

Ahora podemos escribir el código para conectarse y desconectarse del servidor. Como el código utiliza interfaces, no hay nada nuevo aquí:

```
procedure TForm1.btnConnectClick(Sender: TObject);
begin
  FMemo := CoMemoIntf.Create;
end;
procedure Tform1.btnDisconnectClick(Sender: TObject);
begin
  FMemo := nil;
end;
```

Añadiremos tres botones al formulario para configurar el tipo de letra, el color y el contenido del control de edición remoto. Como hemos señalado anteriormente, Delphi proporciona el *marshaling* de tipos de letra, listas de cadenas e imágenes. Esto hace que el código que tenemos que escribir resulte trivial.

```
procedure TForm1.btnSetFontClick(Sender: TObject);
var
  FontDisp: IfontDisp;
begin
  if Fontdialog1.Execute then begin
    Memo1.Font.Assign(FontDialog1.Font);
    GetOleFont(FontDialog1.Font, FontDisp);
    FMemo.Set_Font(FontDisp);
  end;
end;
```

GetOleFont en un procedimiento de Delphi que "convierte" un objeto *TFont* en una interfaz *IFontDisp* apropiada para atravesar las fronteras de un proceso. De forma similar, *GetOleStrings* produce una interfaz *IStrings* a partir de un objeto *TStrings*. El Listado 4.6 muestra el código de la aplicación cliente.

Listado 4.6: Aplicación MemoCli - *MainForm.pas*

```
unit MainForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  StdCtrls, MemoSrv_TLB;
type
  TForm1 = class(TForm)
    btnSetColor: TButton;
    btnConnect: TButton;
    btnSetFont: TButton;
    btnSetText: TButton;
    btnDisconnect: TButton;
    btnGetColor: TButton;
    btnGetFont: TButton;
    btnGetText: TButton;
    Memo1: TMemo;
    FontDialog1: TFontDialog;
    ColorDialog1: TColorDialog;
    procedure btnConnectClick(Sender: TObject);
    procedure btnDisconnectClick(Sender: TObject);
    procedure btnSetColorClick(Sender: TObject);
    procedure btnSetFontClick(Sender: TObject);
    procedure btnSetTextClick(Sender: TObject);
    procedure btnGetColorClick(Sender: TObject);
    procedure btnGetFontClick(Sender: TObject);
    procedure btnGetTextClick(Sender: TObject);
  private
```

```
    { Private declarations }
    FMemo: IMemoIntf;
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
uses
  ActiveX, AXCtrls, StdVCL;
{$R *.DFM}
procedure TForm1.btnConnectClick(Sender: TObject);
begin
  FMemo := CoMemoIntf.Create;
end;
procedure TForm1.btnDisconnectClick(Sender: TObject);
begin
  FMemo := nil;
end;
procedure TForm1.btnSetColorClick(Sender: TObject);
begin
  if ColorDialog1.Execute then begin
    Memo1.Color := ColorDialog1.Color;
    FMemo.Set_Color(ColorToRGB(ColorDialog1.Color));
  end;
end;
procedure TForm1.btnSetFontClick(Sender: TObject);
var
  FontDisp: IFontDisp;
begin
  if FontDialog1.Execute then begin
    Memo1.Font.Assign(FontDialog1.Font);
    GetOleFont(FontDialog1.Font, FontDisp);
    FMemo.Set_Font(FontDisp);
  end;
end;
procedure TForm1.btnSetTextClick(Sender: TObject);
var
  Strings: IStrings;
begin
  GetOleStrings(Memo1.Lines, Strings);
  FMemo.Set_Text(Strings);
end;
procedure TForm1.btnGetColorClick(Sender: TObject);
begin
  Memo1.Color := FMemo.Get_Color;
end;
procedure TForm1.btnGetFontClick(Sender: TObject);
var
  FontDisp: IFontDisp;
begin
  FontDisp := FMemo.Get_Font;
```

```
  SetOleFont(Memo1.Font, FontDisp);
end;
procedure TForm1.btnGetTextClick(Sender: TObject);
var
  Strings: IStrings;
begin
  Strings := FMemo.Get_Text;
  SetOleStrings(Memo1.Lines, Strings);
end;
end.
```

La Figura 4.11 muestra la aplicación *MemoCli* conectada a *MemoSrv*.

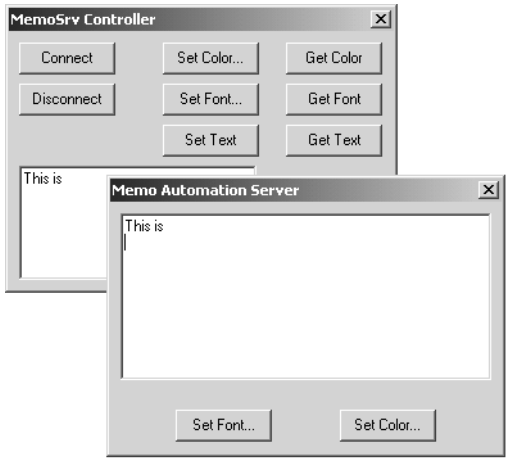


Figura 4.11: *MemoCli* controla al servidor de automatización *MemoSrv*.

Recapitulando, hemos ejecutado los siguientes pasos para crear un cliente de automatización:

- Crear una nueva aplicación.
- Importar la biblioteca de tipos del servidor.
- Añadir en la cláusula **uses** de la aplicación cliente la unidad de importación.
- Realizar llamadas a los métodos del servidor de automatización.

Hasta aquí, hemos cubierto las bases de la automatización. Ahora sabe cómo crear un servidor de automatización, y cómo puede controlar el servidor utilizando interfaces, disp-interfaces y variantes. En la próxima sección examinaremos las características más avanzadas de la automatización COM: los eventos y funciones de respuesta (*callbacks*).

Eventos y funciones de respuesta

Los programadores de Delphi utilizan los eventos constantemente para las más diversas tareas de programación. Todavía no hemos mostrado controladores de automatización que disparen eventos. Aún cuando ésta no es realmente una característica avanzada, hay que trabajar un poco en Delphi para dar soporte a los eventos. Ciertamente es un trabajo menor que el que tendríamos que realizar en otros lenguajes, sin embargo no es posible afirmar que Delphi hace transparente el trabajo con los eventos COM (por supuesto, es de esperar que futuras versiones de Delphi incluyan un mayor soporte).

Podemos usar disp-interfaces o interfaces para implementar un mecanismo mediante el cual el servidor haga llamadas a la aplicación cliente. Ambos métodos tienen sus ventajas y desventajas. Delphi ofrece un mejor soporte para los eventos en caso de que utilicemos disp-interfaces, y debemos usar éstas si pretendemos que el código resultante sea compatible con Visual Basic. Mediante el uso de interfaces se puede obtener un código ligeramente más rápido, pero éstas no son compatibles con Visual Basic y hay que escribir más código. Las dos secciones siguientes examinan ambos métodos en detalle.

Independientemente del método utilizado, el resultado final es el mismo. La aplicación cliente le proporciona al servidor una interfaz que el servidor utiliza para devolver llamadas al cliente.

Disp-interfaces

Delphi proporciona cierto soporte de automatización para eventos mediante disp-interfaces, así que comenzaremos creando un servidor y cliente que hagan uso de los eventos a través de disp-interfaces.

Creando el Servidor de Automatización

Con propósitos ilustrativos, creemos un servidor de automatización que permita que múltiples clientes conectados simultáneamente envíen y reciban texto. En otras palabras, un servidor *chat* sencillo.

Delphi puede manejar automáticamente la creación de servidores de automatización que soporten eventos basados en disp-interfaces. Cree una nueva aplicación, y seguidamente ejecute el Asistente de Objetos de Automatización seleccionando *Automation Object* del Depósito de Objetos.

La Figura 4.12 muestra el Asistente de Objetos de Automatización rellenado para soportar el manejo de eventos. Pulse OK para generar el código fuente del objeto y mostrar el Editor de Bibliotecas de Tipos.

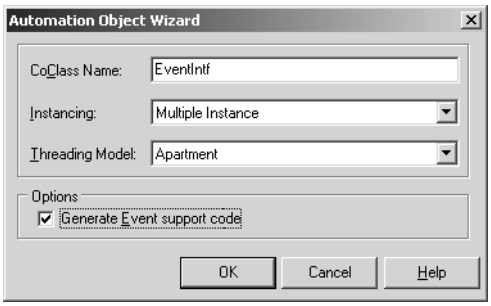


Figura 4.12: Añadiendo manejadores de evento a una disp-interfaz.

Observe que Delphi crea dos interfaces, una para el objeto COM y otra para los eventos que serán lanzados por el objeto.

Añadamos un método llamado *SendText* a la interfaz *IEventIntf*. Proporciónele un parámetro llamado *Text*, de tipo *WideString*. A continuación, asocie a la disp-interfaz *IEventIntfEvents* un evento llamado *OnText*, y añádale un parámetro *Text* de tipo *WideString*.

Pulse el botón '*Refresh Implementation*' en el Editor de Bibliotecas de Tipos y ciérrelo. Guarde el fichero fuente como *EventIntf.pas*. El código fuente del servidor de automatización se asemejará al código del Listado 4.7.

Listado 4.7: Servidor EventSrv - *EventIntf.pas*

```
unit EventIntf;
interface
uses
  ComObj, ActiveX, AxCtrls, EventSrv_TLB;
type
  TEventIntf = class(TAutoObject, IConnectionPointContainer, IEventIntf)
  private
    { Private declarations }
    FConnectionPoints: TConnectionPoints;
    FEvents: IEventIntfEvents;
  public
    procedure Initialize; override;
  protected
    { Protected declarations }
    property ConnectionPoints: TConnectionPoints read FConnectionPoints
      implements IConnectionPointContainer;
    procedure EventSinkChanged(const EventSink: IUnknown); override;
    procedure SendText(const Text: WideString); safecall;
  end;
implementation
uses ComServ;
procedure TEventIntf.EventSinkChanged(const EventSink: IUnknown);
begin
```

```
  FEvents := EventSink as IEventIntfEvents;
end;
procedure TEventIntf.Initialize;
begin
  inherited Initialize;
  FConnectionPoints := TConnectionPoints.Create(Self);
  if AutoFactory.EventTypeInfo  nil then
    FConnectionPoints.CreateConnectionPoint(AutoFactory.EventIID,
      ckSingle, EventConnect);
end;
procedure TEventIntf.SendText(const Text: WideString);
begin
  FEvents.OnText(Text);
end;
initialization
  TAutoObjectFactory.Create(ComServer, TEventIntf, Class_EventIntf,
    ciMultiInstance, tmApartment);
end.
```

Todo lo que necesitamos hacer es añadir la siguiente línea de código al método *SendText*:

```
procedure TEventIntf.SendText(const Text: WideString);
begin
  FEvents.OnText(Text);
end;
```

Compile el servidor, y regístrelo ejecutándolo una vez.

El código fuente del fichero *EventSrv_TLB.pas* generado automáticamente se muestra en el Listado 4.8.

Listado 4.8: Servidor EventSrv - *EventSrv_TLB.pas*

```
unit EventSrv_TLB;
// ************************************************************************ //
// WARNING                                                                  //
// □□□-                                                                     //
// The types declared in this file were generated from data read from       //
// a Type Library. If this type library is explicitly or indirectly         //
// (via another type library referring to this type library)                //
// re-imported, or the 'Refresh' command of the Type Library Editor         //
// activated while editing the Type Library, the contents of this file      //
// will be regenerated and all manual modifications will be lost.           //
// ************************************************************************ //
// PASTLWTR : $Revision:   1.11.1.75  $
// File generated on 5/15/01 2:27:31 PM from Type Library described
below.
// ************************************************************************ //
// Type Lib: C:\Book\samples\Chap04\EventSrv\EventSrv.tlb
// IID\LCID: {34FB8111-476E-11D3-B83E-0040F67455FE}\0
// Helpfile:
```

```
// HelpString: Project1 Library
// Version:    1.0
// ************************************************************************ //
interface
uses Windows, ActiveX, Classes, Graphics, OleCtrls, StdVCL;
/ ************************************************************************ //
// GUIDS declared in the TypeLibrary. Following prefixes are used:        //
//   Type Libraries     : LIBID_xxxx                                      //
//   CoClasses          : CLASS_xxxx                                      //
//   DISPInterfaces     : DIID_xxxx                                       //
//   Non-DISP interfaces: IID_xxxx                                        //
// ************************************************************************ //
const
  LIBID_EventSrv: TGUID = '{34FB8111-476E-11D3-B83E-0040F67455FE}';
  IID_IEventIntf: TGUID = '{34FB8112-476E-11D3-B83E-0040F67455FE}';
  DIID_IEventIntfEvents: TGUID = '{34FB8114-476E-11D3-B83E-0040F67455FE}';
  CLASS_EventIntf: TGUID = '{34FB8116-476E-11D3-B83E-0040F67455FE}';
type
// ************************************************************************ //
// Forward declaration of interfaces defined in Type Library              //
// ************************************************************************ //
  IEventIntf = interface;
  IEventIntfDisp = dispinterface;
  IEventIntfEvents = dispinterface;
// ************************************************************************ //
// Declaration of CoClasses defined in Type Library                       //
// (NOTE: Here we map each CoClass to its Default Interface)              //
// ************************************************************************ //
  EventIntf = IEventIntf;
// ************************************************************************ //
// Interface: IEventIntf
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {34FB8112-476E-11D3-B83E-0040F67455FE}
// ************************************************************************ //
  IEventIntf = interface (IDispatch)
    ['{34FB8112-476E-11D3-B83E-0040F67455FE}']
    procedure SendText(const Text: WideString); safecall;
  end;
// ************************************************************************ //
// DispIntf:  IEventIntfDisp
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {34FB8112-476E-11D3-B83E-0040F67455FE}
// ************************************************************************ //
  IEventIntfDisp = dispinterface
    ['{34FB8112-476E-11D3-B83E-0040F67455FE}']
    procedure SendText(const Text: WideString); dispid 1;
  end;
// ************************************************************************ //
// DispIntf:  IEventIntfEvents
// Flags:     (4096) Dispatchable
// GUID:      {34FB8114-476E-11D3-B83E-0040F67455FE}
// ************************************************************************ //
```

```
  IEventIntfEvents = dispinterface
    ['{34FB8114-476E-11D3-B83E-0040F67455FE}']
    procedure OnText(const Text: WideString); dispid 1;
  end;
  CoEventIntf = class
    class function Create: IEventIntf;
    class function CreateRemote(const MachineName: string): IEventIntf;
  end;
implementation
uses ComObj;
class function CoEventIntf.Create: IEventIntf;
begin
  Result := CreateComObject(CLASS_EventIntf) as IEventIntf;
end;
class function CoEventIntf.CreateRemote(const Machine: string):
IEventIntf;
begin
  Result := CreateRemoteComObject(Machine, CLASS_EventIntf) as IEventIntf;
end;
end.
```

Creando una aplicación cliente con Delphi 3 o Delphi 4

Mientras que Delphi se ocupa por nosotros de todo el trabajo sucio del lado del servidor, necesitamos realizar un pequeño esfuerzo del lado del cliente.

Nota

Delphi5 introdujo nuevo soporte para el manejo de eventos COM que hace esta sección obsoleta. Incluimos esta sección para aquellos lectores que usan las versiones 3 y 4 de Delphi. Si usa Delphi 5 ó 6, puede ir directamente a la sección titulada “Creando una aplicación cliente con Delphi 5 ó 6”.

El Componente TEventSink

Nota

Con el objetivo de encapsular el código necesario para manejar eventos recibidos de un servidor COM he escrito el componente TEventSink. Este no es un libro sobre cómo crear componentes con Delphi, así que no entraremos en detalles sobre cómo hacerlo. En cualquier caso, no es necesario entender cómo crear componentes para utilizar un componente existente en nuestras aplicaciones.

Una pequeña discusión sobre terminología nos podría ser de utilidad en estos momentos. Un sumidero de eventos (event sink) implementa los eventos de una interfaz. Una fuente de eventos (event source) es responsable de llamar a los eventos definidos por la interfaz. En la aplicación que estamos desarrollando, el servidor es la fuente de eventos, y el cliente es el sumidero. El servidor llamará a los eventos definidos por la interfaz, y los eventos serán ejecutados en el contexto del cliente.

El código fuente del componente TEventSink parece muy extenso, pero la mayor parte de los métodos son rutinas vacías para funcionalidad que no necesitamos implementar. Los métodos importantes son QueryInterface, Invoke, Connect y Disconnect.

QueryInterface inicialmente comprueba si quien hace la llamada está solicitando una de las interfaces implementadas, entre las que se incluyen (según podemos ver de la propia declaración de *TAbstractEventSink*) *IUnknown* e *IDispatch*. Si la interfaz solicitada no es una de estas dos interfaces, entonces el código verifica si se ha solicitado la interfaz de eventos (*FDispIntfIID*). Si este es el caso, la interfaz *IDispatch* es devuelta.

```
function TAbstractEventSink.QueryInterface(const IID: TGUID;
  out Obj): HRESULT;
begin
  Result := E_NOINTERFACE;
  if GetInterface(IID, Obj) then
    Result := S_OK;
  if IsEqualGUID(IID, FDispIntfIID) and (GetInterface(IDispatch, Obj) then
    Result := S_OK;
end;
```

El método *Invoke* simplemente pasa sus parámetros al componente *TEventSink*. El propietario del componente *TEventSink* puede responder al evento como desee.

```
function  TAbstractEventSink.Invoque(DispID: Integer; const IID: TGUID;
  LocaleID: Integer; Flags: Word; var Params;
  VarResult, ExcepInfo, ArgErr: Pointer): HRESULT;
begin
  (FOwner as TEventSink).DoInvoke(DispID, IID, LocaleID, Flags,
     Params, VarResult, ExcepInfo, Argerr);
  Result := S_OK;
end;
```

Los métodos *Connect* y *Disconnect* simplemente se encargan de conectarse y desconectarse del servidor. Para ello llaman respectivamente a los métodos predefinidos de Delphi *InterfaceConnect* e *InterfaceDisconnect*.

El código fuente del componente *TEventSink* se muestra en el Listado 4.9.

Listado 4.9: El componente *EventSink*

```
unit EventSink;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  ActiveX;
type
  TInvokeEvent = procedure(Sender: TObject; DispID: Integer;
    const IID: TGUID; LocaleID: Integer; Flags: Word;
```

```
    Params: TDispParams; VarResult, ExcepInfo, ArgErr: Pointer) of object;
  TAbstractEventSink  = class(TInterfacedObject, IUnknown, IDispatch)
  private
    FDispatch: IDispatch;
    FDispIntfIID: TGUID;
    FConnection: Integer;
    FOwner: TComponent;
  protected
    { IUnknown }
    function QueryInterface(const IID: TGUID; out Obj): HRESULT; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
    { IDispatch }
    function GetTypeInfoCount(out Count: Integer): HRESULT; stdcall;
    function GetTypeInfo(Index, LocaleID: Integer;
      out TypeInfo): HRESULT; stdcall;
    function GetIDsOfNames(const IID: TGUID; Names: Pointer;
      NameCount, LocaleID: Integer; DispIDs: Pointer): HRESULT; stdcall;
    function Invoke(DispID: Integer; const IID: TGUID; LocaleID: Integer;
      Flags: Word; var Params;
      VarResult, ExcepInfo, ArgErr: Pointer): HRESULT; stdcall;
  public
    constructor Create(AOwner: TComponent);
    destructor Destroy; override;
    procedure Connect(AnAppDispatch: IDispatch;
      const AnAppDispIntfIID: TGUID);
    procedure Disconnect;
  end;
  TEventSink = class(TComponent)
  private
    { Private declarations }
    FSink: TAbstractEventSink;
    FOnInvoke: TInvokeEvent;
  protected
    { Protected declarations }
    procedure DoInvoke(DispID: Integer; const IID: TGUID;
      LocaleID: Integer; Flags: Word; var Params;
      VarResult, ExcepInfo, ArgErr: Pointer); virtual;
  public
    { Public declarations }
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    procedure Connect(AnAppDispatch: IDispatch;
      const AnAppDispIntfIID: TGUID);
  published
    { Published declarations }
    property OnInvoke: TInvokeEvent read FOnInvoke write FOnInvoke;
  end;
procedure Register;
implementation
uses
  ComObj;
```

```
procedure Register;
begin
  RegisterComponents('DCP', [TEventSink]);
end;
{$IFDEF VER100}
procedure InterfaceConnect(const Source: IUnknown; const IID: TIID;
  const Sink: IUnknown; var Connection: Longint);
var
  CPC: IConnectionPointContainer;
  CP: IConnectionPoint;
begin
  Connection := 0;
  if Succeeded(Source.QueryInterface(IConnectionPointContainer, CPC)) then
    if Succeeded(CPC.FindConnectionPoint(IID, CP)) then
      CP.Advise(Sink, Connection);
end;
procedure InterfaceDisconnect(const Source: IUnknown; const IID: TIID;
  var Connection: Longint);
var
  CPC: IConnectionPointContainer;
  CP: IConnectionPoint;
begin
  if Connection  0 then
    if Succeeded(Source.QueryInterface(IConnectionPointContainer, CPC))
then
      if Succeeded(CPC.FindConnectionPoint(IID, CP)) then
        if Succeeded(CP.Unadvise(Connection)) then Connection := 0;
end;
{$ENDIF}
{ TAbstractEventSink }
function TAbstractEventSink._AddRef: Integer;
begin
  Result := -1;
end;
function TAbstractEventSink._Release: Integer;
begin
  Result := -1;
end;
constructor TAbstractEventSink.Create(AOwner: TComponent);
begin
  inherited Create;
  FOwner := AOwner;
end;
destructor TAbstractEventSink.Destroy;
begin
  Disconnect;
  inherited Destroy;
end;
function TAbstractEventSink.GetIDsOfNames(const IID: TGUID; Names:
Pointer;
  NameCount, LocaleID: Integer; DispIDs: Pointer): HRESULT;
begin
```

```
  Result := E_NOTIMPL;
end;
function TAbstractEventSink.GetTypeInfo(Index, LocaleID: Integer;
  out TypeInfo): HRESULT;
begin
  Result := E_NOTIMPL;
end;
function TAbstractEventSink.GetTypeInfoCount(out Count: Integer): HRESULT;
begin
  Count := 0;
  Result := S_OK;
end;
function TAbstractEventSink.Invoke(DispID: Integer; const IID: TGUID;
  LocaleID: Integer; Flags: Word; var Params; VarResult, ExcepInfo,
  ArgErr: Pointer): HRESULT;
begin
  (FOwner as TEventSink).DoInvoke(DispID, IID, LocaleID, Flags,
    Params, VarResult, ExcepInfo, ArgErr);
  Result := S_OK;
end;
function TAbstractEventSink.QueryInterface(const IID: TGUID;
  out Obj): HRESULT;
begin
  // Devolver la interfaz que nos piden
  Result := E_NOINTERFACE;
  if GetInterface(IID,Obj) then
    Result := S_OK;
  if IsEqualGUID(IID, FDispIntfIID) and GetInterface(IDispatch,Obj) then
    Result := S_OK;
end;
procedure TAbstractEventSink.Connect(AnAppDispatch: IDispatch;
  const AnAppDispIntfIID: TGUID);
begin
  FDispIntfIID := AnAppDispIntfIID;
  FDispatch := AnAppDispatch;
  // Conectar sumidero al servidor de automatizaci□n
  InterfaceConnect(FDispatch, FDispIntfIID, Self, FConnection);
end;
procedure TAbstractEventSink.Disconnect;
begin
  if Assigned(FDispatch) then begin
    // Desconectar sumidero de eventos del servidor de automatizaci□n
    InterfaceDisconnect(FDispatch, FDispIntfIID, FConnection);
    FDispatch := nil;
    FConnection := 0;
  end;
end;
{ TEventSink }
procedure TEventSink.Connect(AnAppDispatch: IDispatch;
  const AnAppDispIntfIID: TGUID);
begin
  FSink.Connect(AnAppDispatch, AnAppDispIntfIID);
```

```
end;
constructor TEventSink.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FSink := TAbstractEventSink.Create(self);
end;
destructor TEventSink.Destroy;
begin
  FSink.Free;
  inherited Destroy;
end;
procedure TEventSink.DoInvoke(DispID: Integer; const IID: TGUID;
  LocaleID: Integer; Flags: Word;
  var Params; VarResult, ExcepInfo, ArgErr: Pointer);
begin
  if Assigned(FOnInvoke) then
    FOnInvoke(self, DispID, IID, LocaleID, Flags, TDispParams(Params),
      VarResult, ExcepInfo, ArgErr);
end;
end.
```

La aplicación cliente

Gracias al componente *TEventSink*, el programa cliente es bastante sencillo de escribir. Este se muestra en el Listado 4.10.

Listado 4.10: EventCli - *Mainform.pas*

```
unit MainForm;
interface
uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs,
    StdCtrls, EventSrv_TLB, ActiveX, ComObj, EventSink, ExtCtrls;
  type
    TForm1 = class(TForm)
      EventSink1: TEventSink;
      Panel1: TPanel;
      Panel2: TPanel;
      btnSend: TButton;
      Memo1: TMemo;
      Edit1: TEdit;
      procedure FormCreate(Sender: TObject);
      procedure btnSendClick(Sender: TObject);
      procedure EventSink1Invoke(Sender: TObject; DispID: Integer;
        const IID: TGUID; LocaleID: Integer; Flags: Word;
        Params: tagDISPPARAMS; VarResult, ExcepInfo, ArgErr: Pointer);
    private
      { Private declarations }
      F: IEventIntf;
    public
      { Public declarations }
```

```
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.FormCreate(Sender: TObject);
begin
  F := CoEventIntf.Create;
  EventSink1.Connect(F, IEventIntfEvents);
end;
procedure TForm1.btnSendClick(Sender: TObject);
begin
  F.SendText(Edit1.Text);
end;
procedure TForm1.EventSink1Invoke(Sender: TObject; DispID: Integer;
  const IID: TGUID; LocaleID: Integer; Flags: Word; Params: tagDISPPARAMS;
  VarResult, ExcepInfo, ArgErr: Pointer);
var
  vText: OleVariant;
begin
  case DispID of
    1: begin
         vText := OleVariant(Params.rgvarg^[0]);
         Memo1.Lines.Add(vText);
       end;
  end;
end;
end.
```

El mayor dolor de cabeza con este método es que hay que descifrar manualmente los parámetros en *EventSink1Invoke*. El método *EventSink1Invoke* del Listado 4.10 muestra cómo realizar esta tarea.

```
procedure TForm1.EventSink1Invoke(Sender: TObject; DispID: Integer;
  const IID: TGUID; LocaleID: Integer; Flags: Word; Params: tagDISPPARAMS;
  VarResult, ExcepInfo, ArgErr: Pointer);
var
  vText: OleVariant;
begin
  case DispID of
    1: begin
         vText := OleVariant(Params.rgvarg^[0]);
         Memo1.Lines.Add(vText);
       end;
  end;
end;
```

Observe que en este código se conoce la cantidad y los tipos de argumentos asociados a cada *DispID*. La cantidad de argumentos comienza en cero, y el evento con **dispid** 1 (*OnText*) recibe un único argumento de tipo *WideString*. Es posible

escribir código que detecte la cantidad y los tipos de los argumentos en tiempo de ejecución, pero ello está fuera del alcance de este libro.

Creando una Aplicación cliente con Delphi 5 ó 6

Con la aparición de Delphi 5, crear una aplicación cliente es aún más sencillo que crear el servidor. Cree una nueva aplicación, y seguidamente seleccione *Project Import Type Library* del menú principal. Aparecerá el cuadro de diálogo que se muestra en la Figura 4.13.

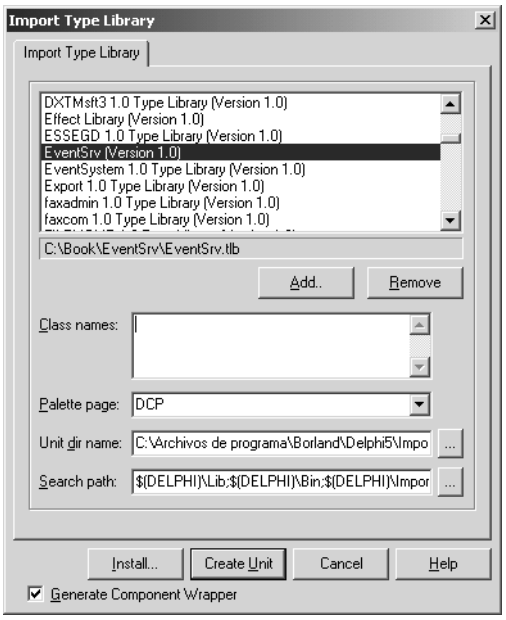


Figura 4.13: Delphi 5 puede generar automáticamente un componente "envoltorio" para el servidor COM.

Observe entre las características del diálogo *Import Type Library* la presencia de una casilla de verificación adicional llamada '*Generate Component Wrapper*' *(Generar componente "envoltorio"). Seleccione EventSrv (Version 1.0)* en el cuadro de lista, y '*DCP*' en el cuadro destinado a la página de la Paleta de Componentes en la que se desea colocar el componente. Si lo prefiere, puede instalar el componente en otra página, como por ejemplo *ActiveX*. Después de asegurarse de haber marcado la casilla '*Generate Component Wrapper*', pulse el botón '*Install...*'.

Delphi preguntará por el paquete en el que desea instalar el componente. Puede aceptar el que se propone por defecto y pulsar OK. Construya el paquete cuando se le solicite. Si todo transcurre normalmente, Delphi informará que ha instalado el componente exitosamente en la Paleta de Componentes.

Cree ahora una aplicación y arrastre sobre el formulario principal un componente *TEventIntf* desde la página '*DCP*' (o la página en la que ha instalado el componente). El componente *TEventIntf* publica tres propiedades que son de interés para nosotros.

AutoConnect determina si la aplicación debe tratar de conectarse a un servidor automáticamente al ser arrancada. Si *AutoConnect* es *False*, debemos llamar a *EventIntf1.Connect* para conectarnos al servidor. Asigne a esta propiedad el valor *True* para el ejemplo.

ConnectKind le indica al componente cómo conectarse al servidor. Los valores válidos para la propiedad *ConnectKind* se muestran en la Tabla 4.1.

Tabla 4.1: Valores válidos para *ConnectKind*

Opción de Conexión	Descripción
ckAttachToInterface	Esta es una opción avanzada, que no será examinada en este libro.
ckNewInstance	El cliente siempre crea y se conecta a una nueva instancia del servidor.
ckRemote	El servidor se ejecuta en una máquina remota. Esta opción se examina en Capítulo 6.
ckRunningInstance	El cliente solamente se conecta a una instancia en ejecución del servidor
ckRunningOrNew	El cliente trata de conectarse a una instancia en ejecución del servidor. Si el servidor no está ejecutándose, el cliente inicia una nueva instancia del servidor.

Para el ejemplo, asignemos a la propiedad el valor *ckRunningOrNew*.

La propiedad *RemoteMachineName* sólo juega un papel cuando nos conectamos a un servidor remoto. Analizaremos esta propiedad en el Capítulo 6.

El componente *TEventIntf* publica un solo evento llamado *OnText*. Este corresponde al evento del mismo nombre que hemos añadido a la disp-interfaz *IEventIntfEvents*.

El código fuente de la aplicación cliente se muestra en el Listado 4.11.

Listado 4.11: EventCli5 - *MainForm.pas*

```
unit MainForm;
interface
uses
```

```
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  OleServer, EventSrv_TLB, StdCtrls, ExtCtrls;
type
  TForm1 = class(TForm)
    EventIntf1: TEventIntf;
    Panel1: TPanel;
    Panel2: TPanel;
    Memo1: TMemo;
    Edit1: TEdit;
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
    procedure EventIntf1Text(Sender: TObject; var Text: OleVariant);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.Button1Click(Sender: TObject);
begin
  EventIntf1.SendText(Edit1.Text);
end;
procedure TForm1.EventIntf1Text(Sender: TObject; var Text: OleVariant);
begin
  Memo1.Lines.Add(Text);
end;
end.
```

Ejecutando la aplicación cliente

Independientemente de la versión de Delphi que hemos utilizado para crear la aplicación cliente, cuando la ejecutemos su apariencia será similar a la de la Figura 4.14.

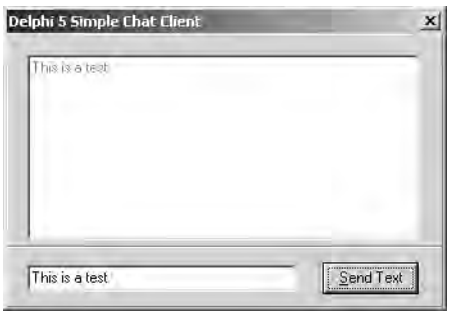


Figura 4.14: Las aplicaciones *EventCli/EventCli5* permiten enviar líneas de texto a otros clientes conectados.

Conectando Múltiples Clientes a un Servidor

Existe un pequeño problema con el código del servidor tal cual lo tenemos ahora. Aunque un solo servidor será creado para dar servicio a todos los clientes, el servidor sólo enviará eventos al primer cliente conectado. Esto no es obviamente lo que deseamos.

Necesitaremos realizar algunas modificaciones al servidor y a los clientes para resolver este problema. Afortunadamente, los cambios en ambos lados son mínimos.

Nota

Si estamos usando Delphi 5 o superior, podemos saltar los cambios del lado del cliente. El componente TEventIntf generado automáticamente por Delphi se encarga de ellos.

Primero, modifiquemos el método *Initialize* del servidor:

```
procedure TEventIntf.Initialize;
begin
  inherited Initialize;
  FConnectionPoints := TConnectionsPoints.Create(Self);
  if AutoFactory.EventTypeInfo  nil then
    FConnectionPoints.CreateConnectionPoint(AutoFactory.EventIID,
      ckMulti, EventConnect);
end;
```

Observe que he cambiado el valor del penúltimo parámetro de *CreateConnectionPoint* de *ckSingle* a *ckMulti*. Esto es todo lo que debemos realizar para que el servidor "recuerde" múltiples conexiones de clientes.

Ahora que el servidor puede mantener múltiples conexiones simultáneas, necesitamos alguna forma de iterar a través de todas las conexiones activas del servidor. La interfaz *IConnectionPointContainer* nos proporciona una enumeración que puede ser utilizada para iterar a través de las conexiones. El siguiente método puede usarse para obtener un enumerador para las conexiones:

```
function TEventIntf.GetEnumerator: IEnumConnections;
var
  Container: IConnectionPointContainer;
  ConnectionPoint: IConnectionPoint;
begin
  OleCheck(QueryInterface(IConnectionPointContainer, Container));
  OleCheck(Container.FindConnectionPoint(AutoFactory.EventIID,
    ConnectionPoint));
  ConnectionPoint.EnumConnections(Result);
end;
```

Una vez que tenemos el enumerador, debemos simplemente iterar por todas las conexiones, disparando el evento para cada una de ellas. El siguiente código muestra cómo modificar el evento *Trigger* para soportar múltiples conexiones:

```
procedure TEventIntf.SendText(const Text: WideString);
var
  Enum: IEnumConnections;
  ConnectData: TConnectData;
  Fetched: Cardinal;
begin
  Enum := GetEnumerator;
  if Enum  nil then
    while Enum.Next(1, ConnectData, @Fetched) = S_OK do
      if ConnectData.pUnk  nil then
        (ConnectData.pUnk as IEventIntfEvents).OnText(Text);
end;
```

El cambio final que debemos realizar en el código del servidor es registrar el servidor en la tabla de objetos en ejecución de Windows. Para esto, simplemente añada la siguiente línea de código al final de la función *Initialize*:

```
RegisterActiveObject(Self as IUnknown, CLASS_EventIntf,
  ACTIVEOBJECT_WEAK, FObjectID);
```

Para eliminar el servidor de la tabla de objetos en ejecución, cree un método *Destroy* y asóciele la siguiente línea de código:

```
RevokeActiveObject(FobjectID, nil);
```

El Listado 4.12 muestra el código para el servidor modificado.

Listado 4.12: EventMultSrv - *EventIntf.pas*

```
unit EventIntf;
interface
uses
    ComObj, ActiveX, AxCtrls, EventSrv_TLB;
  type
    TEventIntf = class(TAutoObject, IConnectionPointContainer, IEventIntf)
    private
      { Private declarations }
      FConnectionPoints: TConnectionPoints;
      FEvents: IEventIntfEvents;
      FObjectID: Integer;
    public
      procedure Initialize; override;
      destructor Destroy; override;
    protected
      { Protected declarations }
      property ConnectionPoints: TConnectionPoints read FConnectionPoints
        implements IConnectionPointContainer;
      procedure EventSinkChanged(const EventSink: IUnknown); override;
      procedure SendText(const Text: WideString); safecall;
      function GetEnumerator: IEnumConnections;
    end;
  implementation
  uses Windows, ComServ;
  procedure TEventIntf.EventSinkChanged(const EventSink: IUnknown);
```

```
begin
  FEvents := EventSink as IEventIntfEvents;
end;
procedure TEventIntf.Initialize;
begin
  inherited Initialize;
  FConnectionPoints := TConnectionPoints.Create(Self);
  if AutoFactory.EventTypeInfo  nil then
    FConnectionPoints.CreateConnectionPoint(AutoFactory.EventIID,
      ckMulti, EventConnect);
  RegisterActiveObject(self as IUnknown, CLASS_EventIntf,
    ACTIVEOBJECT_WEAK, FObjectID);
end;
procedure TEventIntf.SendText(const Text: WideString);
var
  Enum: IEnumConnections;
  ConnectData: TConnectData;
  Fetched: Cardinal;
begin
  Enum := GetEnumerator;
  if Enum  nil then begin
    while Enum.Next(1, ConnectData, @Fetched) = S_OK do
      if ConnectData.pUnk  nil then
        (ConnectData.pUnk as IEventIntfEvents).OnText(Text);
  end;
end;
function TEventIntf.GetEnumerator: IEnumConnections;
var
  Container: IConnectionPointContainer;
  ConnectionPoint: IConnectionPoint;
begin
  OleCheck(QueryInterface(IConnectionPointContainer, Container));
  OleCheck(Container.FindConnectionPoint(AutoFactory.EventIID,
    ConnectionPoint));
  ConnectionPoint.EnumConnections(Result);
end;
destructor TEventIntf.Destroy;
begin
  RevokeActiveObject(FObjectID, nil);
  inherited Destroy;
end;
initialization
  TAutoObjectFactory.Create(ComServer, TEventIntf, Class_EventIntf,
    ciMultiInstance, tmApartment);
end.
```

Del lado del cliente, necesitamos modificar el método *FormCreate* de la siguiente forma:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  Obj: IUnknown;
begin
```

```
    GetActiveObject(CLASS_EventIntf, nil, Obj);
    if Obj  nil then
      F := Obj as IEventIntf
    else
      F := CoEventIntf.Create;
    EventSink1.Connect(F, IEventInfEvents);
  end;
```

GetActiveObject busca una instancia en ejecución del servidor de automatización *EventMultSrv*. Si encuentra una, esa instancia es utilizada. Si no existe ninguna instancia activa del servidor, la llamada a *CoEventIntf.Create* inicia una nueva.

La Figura 4.15 muestra dos clientes accediendo simultáneamente al servidor.

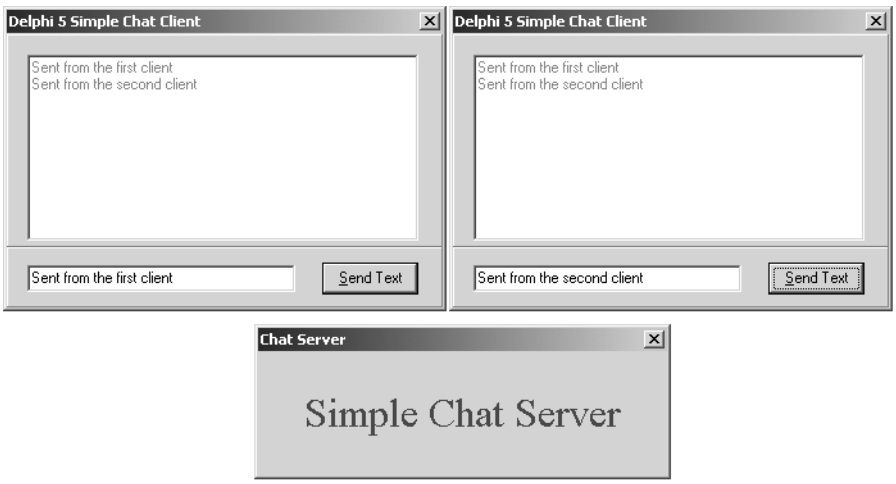


Figura 4.15: El servidor puede enviar eventos a múltiples clientes conectados.

Interfaces de respuesta

El siguiente método que presentaremos requiere una considerable cantidad de programación del lado del servidor, pero bastante poca del lado del cliente. En lugar de usar disp-interfaces para enviar eventos a la aplicación cliente, podemos crear una interfaz en que se definan métodos de respuesta. La interfaz de respuesta se define en el servidor, pero es implementada en el cliente.

En esta sección crearemos un cliente y un servidor de pruebas, que ilustran el proceso de utilización de una interfaz personalizada para realizar llamadas del servidor al cliente.

Creando el servidor

Para crear la aplicación servidora, inicie un nuevo proyecto y añada un objeto de automatización, como hemos hecho con anterioridad. Seleccione *File | New*

Application del menú principal de Delphi. Seguidamente seleccione *File|New...* para mostrar el Depósito de Objetos. En la página *ActiveX*, seleccione *Automation Object* y pulse OK.

Rellene los campos del Asistente de Objetos de Automatización según se muestra en la Figura 4.16.

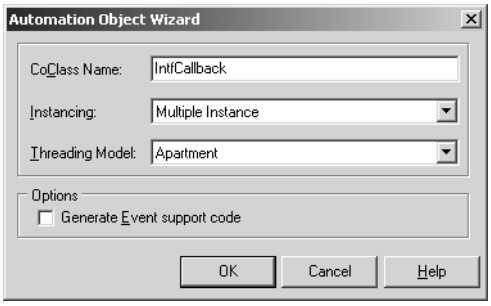


Figura 4.16: Asistente para Objetos de Automatización.

Aparecerá el Editor de Bibliotecas de Tipos. Lo primero que debe hacer es crear una interfaz que utilizaremos para devolver llamadas al cliente. Utilice el Editor de Bibliotecas de Tipos para añadir una nueva interfaz llamada *IIntfCallbackEvents*. A continuación, añada a la interfaz un método llamado *OnText*. *OnText* se define mediante el siguiente fragmento de código:

```
procedure OnText(Text: WideString); safecall;
```

Ud. debe estar a estas alturas lo suficientemente familiarizado con el Editor de Bibliotecas de Tipos como para añadir un método por su cuenta.

Ahora añada tres métodos a la interfaz *IIntfCallback*, llamados *Connect*, *Disconnect* y *SendText*. Sus declaraciones se muestran en el siguiente código:

```
function Connect(const Callback: ITestEvents): Integer;
function Disconnect(UserID: Integer): Boolean;
procedure SendText(Text: WideString);
```

Estos son los únicos métodos que definiremos en este ejemplo. Pulse el botón '*Refresh Implementation*' en el Editor de Bibliotecas de Tipos, y luego ciérrelo.

Nota:

Debe ser evidente que estamos creando básicamente el mismo servidor de *chat* de la sección anterior, aunque hemos utilizado una interfaz de respuesta en lugar de eventos COM.

La interfaz *IIntfCallbackEvents* será implementada en la aplicación cliente, así que no examinaremos cómo implementar *OnText*. En lugar de ello, nos concentraremos en los tres métodos de *IIntfCallback*: *Connect*, *Disconnect* y *SendText*.

Instanciando el servidor

Cuando un servidor es solicitado por un cliente, el método *Initialize* del servidor es llamado. *TIntfCallback.Initialize* se codifica de la siguiente forma:

```
procedure TIntfCallback.Initialize;
begin
  inherited Initialize;
  // Incrementar el n□mero global de conexiones
  Inc(NumConnections);
  // Mostrar n□mero de conexiones
  if NumConnections = 1 then
    Form1.Label2.Caption := '(1 active connection)'
  else
    Form1.Label2.Caption := '(' + IntToStr(NumConnections)
      + ' active connections)';
end;
```

Por supuesto, lo primero es llamar al procedimiento heredado *Initialize*. Después de eso, se incrementa el número de conexiones al servidor. *NumConnections* es una variable global; debe ser global ya que *Initialize* será llamada cuando un cliente se conecte al servidor. Se desea mantener el seguimiento, en una localización centralizada, de la cantidad de conexiones al servidor.

Después que el número de conexiones ha sido actualizado, el formulario principal es actualizado para reflejar esta cantidad.

Manejando las Conexiones de Cliente

La aplicación cliente llamará al método *Connect* para establecer una conexión entre el cliente y el servidor.

Nota

No hay nada mágico en relación con el nombre Connect. Podríamos haber llamado a este método *RegisterClient* o con cualquier otro nombre.

Normalmente se desea permitir la conexión de múltiples clientes al servidor, por lo que el método *Connect* añade el cliente que se conecta a la lista interna de clientes. Para lograr esto, hemos creado dos clases auxiliares llamadas *TConn* y *TConns*. Consulte en el Listado 4.13 la implementación de estas clases.

TConn representa una conexión de un cliente. Cada cliente debe proporcionar una implementación de la interfaz *IIntfCallbackEvents* para que el servidor la utilice cuando llame al cliente. Del mismo modo, a cada cliente se le asigna un identificador que lo distingue de manera única.

TConns contiene una lista de objetos *TConn*, y además almacena el identificador único asignado al último cliente conectado. Cuando el siguiente cliente se conecta al servidor, el identificador único es incrementado.

Podrá darse cuenta de que *TConns* contiene tres métodos llamados *Connect*, *Disconnect* y *SendText*. *TIntfCallback* (que implementa la interfaz *IIntfCallback*) simplemente transfiere el control al método de *TConns* con el mismo nombre.

Llamando a los clientes desde el servidor

Es un asunto sencillo realizar llamadas desde el servidor a todos los clientes conectados. La lista almacenada por *TConns* contiene una lista de todos los clientes conectados, así como una referencia a sus interfaces *IIntfCallbackEvents*, así que podemos simplemente recorrer la lista de clientes, llamando a un método de la interfaz *IIntfCallbackEvents* para cada cliente. El siguiente código muestra cómo hacerlo:

```
procedure TConns.SendText(text: WideString);
var
  Index: Integer;
  C: Tconn;
begin
  for index := 0 to FConns.Count  1 do begin
    C := TConns[Index];
    C.FCallback.OnText(Text);
  end;
end;
```

Este sencillo procedimiento llama al método *OnText* de la interfaz *IIntfCallbackEvents* de cada uno de los clientes conectados.

El servidor completo

Hemos discutido las secciones más importantes de la aplicación servidora. El Listado 4.13 muestra el código fuente para *IntfUnit.pas*.

Listado 4.13: Servidor de automatización IntfSrv - *IntfUnit.pas*

```
unit IntfUnit;
interface
uses
     Windows, Classes, SysUtils, ComObj, ActiveX, IntfSrv_TLB;
   type
     // Clase para representar una nica conexin
     TConn = class
     public
       FUserID: Integer;
       FCallback: IIntfCallbackEvents;
       destructor Destroy; override;
     end;
     // Clase para representar la lista de conexiones activas
     TConns = class
     private
       FConns: TList;
       FLastUserID: Integer;
```

```
  public
    constructor Create;
    destructor Destroy; override;
    function Connect(const Callback: IIntfCallbackEvents): Integer;
    function Disconnect(UserID: Integer): Boolean;
    procedure SendText(Text: WideString);
  end;
  // Objeto COM con el que el cliente "habla"
  TIntfCallback = class(TAutoObject, IIntfCallback)
  protected
    function Connect(const Callback: IIntfCallbackEvents): Integer;
safecall;
    function Disconnect(UserID: Integer): WordBool; safecall;
    procedure SendText(const Text: WideString); safecall;
  public
    procedure Initialize; override;
    destructor Destroy; override;
    function Connections: TConns;
  end;
// Estas dos variables son globales, así que solo habrá una única
// instancia de la aplicación servidora.
const
  NumConnections: Integer = 0;
var
  GlobalConnections: TConns;
implementation
uses ComServ, MainForm;
{ TIntfCallback }
procedure TIntfCallback.SendText(const Text: WideString);
begin
  // Pasar el texto a la lista de conexiones
  Connections.SendText(Text);
end;
function TIntfCallback.Connect(const Callback: IIntfCallbackEvents):
Integer;
// Decir a la lista de conexiones que añada una nueva conexión
begin
  Result := Connections.Connect(Callback);
end;
destructor TIntfCallback.Destroy;
begin
  // Decrementar la cantidad de conexiones
  Dec(NumConnections);
  // Si no quedan conexiones activas, liberar la lista.
  // Esto no es imprescindible — podríamos simplemente vaciarla.
  if NumConnections = 0 then begin
    GlobalConnections.Free;
    GlobalConnections := nil;
  end;
  // Actualizar formulario principal
  if NumConnections = 1 then
    Form1.Label2.Caption := '(1 active connection)'
```

```
  else
    Form1.Label2.Caption := '(' + IntToStr(NumConnections)
      + ' active connections)';
  inherited Destroy;
end;
// Redefinición del método de TAutoObject.
// Funciona de modo similar a un constructor.
procedure TIntfCallback.Initialize;
begin
  inherited Initialize;
  // Incrementar la cantidad de conexiones
  Inc(NumConnections);
  // Actualizar formulario principal
  if NumConnections = 1 then
    Form1.Label2.Caption := '(1 active connection)'
  else
    Form1.Label2.Caption := '(' + IntToStr(NumConnections)
      + ' active connections)';
end;
function TIntfCallback.Disconnect(UserID: Integer): WordBool;
begin
  // Gestionar petición de desconexión
  Result := Connections.Disconnect(UserID);
end;
function TIntfCallback.Connections: TConns;
begin
  // Devuelve una lista de conexiones global
  if GlobalConnections = nil then
    GlobalConnections := TConns.Create;
  Result := GlobalConnections;
end;
{ TConn }
destructor TConn.Destroy;
begin
  // Liberar explícitamente la interfaz de eventos
  FCallback := nil;
  inherited Destroy;
end;
{ TConns }
function TConns.Connect(const Callback: IIntfCallbackEvents): Integer;
var
  C: TConn;
begin
  // Asignar a cada conexión un ID único
  Inc(FLastUserID);
  // Crear una nueva conexión
  C := TConn.Create;
  // Almacenar la interfaz de eventos
  C.FCallback := Callback;
  // Asignar el ID de usuario
  C.FUserID := FLastUserID;
  // Añadir el usuario a la lista
```

```
  FConns.Add(C);
  // Devolver el ID asignado
  Result := FLastUserID;
end;
constructor TConns.Create;
begin
  FLastUserID := 0;
  FConns := TList.Create;
end;
destructor TConns.Destroy;
var
  Index: Integer;
  C: TConn;
begin
  for Index := 0 to FConns.Count - 1 do begin
    C := TConn(FConns[Index]);
    C.Free;
  end;
  FConns.Free;
end;
function TConns.Disconnect(UserID: Integer): Boolean;
var
  Index: Integer;
  C: TConn;
begin
  Result := False;
  for Index := 0 to FConns.Count - 1 do begin
    C := TConn(FConns[Index]);
    if C.FUserID = UserID then begin
      C.Free;
      FConns.Delete(Index);
      Result := True;
      exit;
    end;
  end;
end;
procedure TConns.SendText(Text: WideString);
var
  Index: Integer;
  C: TConn;
begin
  for Index := 0 to FConns.Count - 1 do begin
    C := TConn(FConns[Index]);
    C.FCallback.OnText(Text);
  end;
end;
initialization
  // Cambie aquí a ciSingleInstance, y cada copia del cliente
  // lanzará su propia copia del servidor.
  TAutoObjectFactory.Create(ComServer, TIntfCallback, Class_IntfCallback,
    ciMultiInstance, tmApartment);
end.
```

El Listado 4.14 muestra el código fuente del formulario principal del servidor. La Figura 4.17 muestra al servidor en tiempo de ejecución.

Figura 4.17: Aplicación *IntfSrv* en tiempo de ejecución.

Listado 4.14: Servidor de automatización IntfSrv - *Mainform.pas*

```
unit MainForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  StdCtrls, IntfUnit;
type
  TForm1 = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
end.
```

El Listado 4.15 muestra la biblioteca de tipos generada por Delphi para el servidor.

Listado 4.15: Servidor de automatización IntfSrv – *IntfSrv_TLB.pas*

```
unit IntfSrv_TLB;
// ************************************************************************ //
// WARNING                                                                  //
// □□□-                                                                     //
// The types declared in this file were generated from data read from       //
// a Type Library. If this type library is explicitly or indirectly         //
// (via another type library referring to this type library)                //
// re-imported, or the 'Refresh' command of the Type Library Editor         //
// activated while editing the Type Library, the contents of this file      //
// will be regenerated and all manual modifications will be lost.           //
// ************************************************************************ //
// PASTLWTR : $Revision:   1.11.1.75  $
```

```
// File generated on 5/15/01 2:59:14 PM from Type Library described
below.
// ************************************************************************ //
// Type Lib: C:\Book\samples\Chap04\IntfSrv\IntfSrv.tlb
// IID\LCID: {E9D7678E-F7E3-11D2-909B-0040F6741DE2}\0
// Helpfile:
// HelpString: IntfSrv Library
// Version:    1.0
// ************************************************************************ //
interface
uses Windows, ActiveX, Classes, Graphics, OleCtrls, StdVCL;
// ************************************************************************ //
// GUIDS declared in the TypeLibrary. Following prefixes are used:     //
//   Type Libraries     : LIBID_xxxx                                   //
//   CoClasses          : CLASS_xxxx                                   //
//   DISPInterfaces     : DIID_xxxx                                    //
//   Non-DISP interfaces: IID_xxxx                                     //
// ************************************************************************ //
const
  LIBID_IntfSrv: TGUID ='{E9D7678E-F7E3-11D2-909B-0040F6741DE2}';
  IID_IIntfCallback: TGUID ='{E9D7678F-F7E3-11D2-909B-0040F6741DE2}';
  CLASS_IntfCallback: TGUID ='{E9D76791-F7E3-11D2-909B-0040F6741DE2}';
  IID_IIntfCallbackEvents: TGUID='{E9D76793-F7E3-11D2-909B-0040F6741DE2}';
type
// ************************************************************************ //
// Forward declaration of interfaces defined in Type Library           //
// ************************************************************************ //
  IIntfCallback = interface;
  IIntfCallbackDisp = dispinterface;
  IIntfCallbackEvents = interface;
  IIntfCallbackEventsDisp = dispinterface;
// ************************************************************************ //
// Declaration of CoClasses defined in Type Library                    //
// (NOTE: Here we map each CoClass to its Default Interface)           //
// ************************************************************************ //
  IntfCallback = IIntfCallback;
// ************************************************************************ //
// Interface: IIntfCallback
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {E9D7678F-F7E3-11D2-909B-0040F6741DE2}
// ************************************************************************ //
  IIntfCallback = interface(IDispatch)
    ['{E9D7678F-F7E3-11D2-909B-0040F6741DE2}']
    procedure SendText(const Text: WideString); safecall;
    function Connect(const Callback:IIntfCallbackEvents):Integer;
safecall;
    function Disconnect(UserID: Integer): WordBool; safecall;
  end;
// ************************************************************************ //
// DispIntf:  IIntfCallbackDisp
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {E9D7678F-F7E3-11D2-909B-0040F6741DE2}
```

```
// ********************************************************************* //
  IIntfCallbackDisp = dispinterface
    ['{E9D7678F-F7E3-11D2-9909-0040F6741DE2}']
    procedure SendText(const Text: WideString); dispid 1;
    function Connect(const Callback:IIntfCallbackEvents):Integer;dispid 2;
    function Disconnect(UserID: Integer): WordBool; dispid 3;
  end;
// ********************************************************************* //
// Interface: IIntfCallbackEvents
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {E9D76793-F7E3-11D2-9909-0040F6741DE2}
// ********************************************************************* //
  IIntfCallbackEvents = interface(IDispatch)
    ['{E9D76793-F7E3-11D2-9909-0040F6741DE2}']
    procedure OnText(const Text: WideString); safecall;
  end;
// ********************************************************************* //
// DispIntf:  IIntfCallbackEventsDisp
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {E9D76793-F7E3-11D2-9909-0040F6741DE2}
// ********************************************************************* //
  IIntfCallbackEventsDisp = dispinterface
    ['{E9D76793-F7E3-11D2-9909-0040F6741DE2}']
    procedure OnText(const Text: WideString); dispid 1;
  end;
  CoIntfCallback = class
    class function Create: IIntfCallback;
    class function CreateRemote(const MachineName: string): IIntfCallback;
  end;
implementation
uses ComObj;
class function CoIntfCallback.Create: IIntfCallback;
begin
  Result := CreateComObject(CLASS_IntfCallback) as IIntfCallback;
end;
class function CoIntfCallback.CreateRemote(const
  MachineName: string): IIntfCallback;
begin
  Result := CreateRemoteComObject(MachineName, CLASS_IntfCallback)
    as IIntfCallback;
end;
end.
```

Creando el Cliente

Una vez completado el servidor, es una tarea muy simple crear la aplicación cliente que se conectará al servidor.

Inicie una nueva aplicación en Delphi. Añada a la cláusula **uses** de la aplicación la copia del servidor de *IntfSrv_TLB.pas*, o importe la bibliotecas de tipos del servidor en Delphi usando el procedimiento descrito con anterioridad en este capítulo.

Implementando la interfaz de respuesta

En la aplicación del cliente es donde implementaremos la interfaz *ITestEvents* definida en la biblioteca de tipos del servidor. La declaración de la interfaz es simple, dado que cuenta con un solo método.

```
type
  TEventHandler = class(TAutoIntfObject, IIntfCallbackEvents)
    procedure OnText(const Text: WideString); safecall;
end;
```

En esta declaración hay un punto de interés. *TEventHandler* es una clase derivada de *TAutoIntfObject*. *TAutoIntfObject* es una clase "ligera" compatible con la automatización que puede utilizar cuando desee implementar una interfaz que no debe ser publicada por Windows. En otras palabras, *IIntfCallbackEvents* es una interfaz privada que sólo conocen este servidor y este cliente. Ud. no desea que ninguna otra aplicación sea capaz de crear una instancia de *IIntfCallbackEvents*, que no tiene sentido fuera de este contexto específico.

Nota

Debe recordar que *TInterfacedObject* es también una clase COM ligera. *TAutoIntfObject* difiere de *TInterfacedObject* en que *TAutoIntfObject* requiere de una biblioteca de tipos, soporta el enlace tardío vía *IDispatch*, y puede ser llamada por clientes COM externos al proceso. *TInterface* no soporta ninguna de estas posibilidades, y ha sido diseñada para ser utilizada dentro de una única aplicación.

IIntfCallbackEvents sólo contiene un método, *DoIt*, y hemos programado una implementación muy sencilla que muestra la cadena de caracteres recibida como parámetro en el formulario principal.

Construyendo el objeto de respuesta

Cuando se inicia la aplicación cliente, ésta crea el objeto *TEventHandler*. Este proceso, aunque sencillo, es diferente a todo lo que hemos visto con anterioridad. Para crear la clase, el código llama a *LoadRegTypeLib*, pasando el GUID y número de versión de la biblioteca de tipos. *LoadRegTypeLib* devuelve una referencia a la interfaz *ITypeLib*. El constructor *TEventHandler.Create* utiliza esta referencia para crear una instancia de *TEventHandler*.

```
procedure TForm1.FormCreate(Sender: TObject);
var
  TypeLib: ITypeLib;
begin
  OleCheck(LoadRegTypeLib(LIBID_IntfSrv, 1, 0, 0, TypeLib));
  FCallback := TEventHandler.Create(TypeLib, IIntfCallbackEvents);
end;
```

Conectándose al servidor

El resto del código es muy similar al código que ya hemos visto antes. Cuando el usuario pulsa el botón '*Connect*', el código llama a *CoTest.Create* para crear una instancia del servidor. Seguidamente, se hace una llamada a *Connect* en el servidor, pasando una referencia a la interfaz *IIntfCallbackEvents* implementada por *TEventHandler*. En realidad se pasa el propio *TEventHandler*, que Delphi convierte automáticamente a una interfaz *IIntfCallbackEvents*.

Cuando el usuario pulsa el botón '*Trigger*', el código simplemente llama al método *Trigger* del servidor, que a su vez llama al método *DoIt* de todos los clientes conectados.

El Listado 4.16 muestra el código fuente de la aplicación cliente.

Listado 4.16: IntfCli - *MainForm.pas*

```
unit MainForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, ComObj, ActiveX, IntfSrv_TLB, StdCtrls, ExtCtrls;
type
  TEventHandler = class(TAutoIntfObject, IIntfCallbackEvents)
    procedure OnText(const Text: WideString); safecall;
  end;
  TForm1 = class(TForm)
    Panel1: TPanel;
    Panel2: TPanel;
    btnSend: TButton;
    Memo1: TMemo;
    btnConnect: TButton;
    btnDisconnect: TButton;
    Edit1: TEdit;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure btnConnectClick(Sender: TObject);
    procedure btnDisconnectClick(Sender: TObject);
    procedure btnSendClick(Sender: TObject);
  private
    { Private declarations }
    FCallback: TEventHandler;
    FServer: IIntfCallback;
    FID: Integer;
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
```

```
{ TEventHandler }
procedure TEventHandler.OnText(const Text: WideString);
begin
  Form1.Memo1.Lines.Add(Text);
end;
procedure TForm1.FormCreate(Sender: TObject);
var
  TypeLib : ITypeLib;
begin
  // Hay que llamar a LoadRegTypeLib para poder acceder
  // a la interfaz de eventos
  OleCheck(LoadRegTypeLib(LIBID_IntfSrv, 1, 0, 0, TypeLib));
  FCallback := TEventHandler.Create(TypeLib, IIntfCallbackEvents);
end;
procedure TForm1.FormDestroy(Sender: TObject);
begin
  // No liberar FCallback aqu  el servidor lo har  por nosotros
end;
procedure TForm1.btnConnectClick(Sender: TObject);
begin
  // Conectarse al servidor
  FServer := CoIntfCallback.Create;
  FID := FServer.Connect(FCallback);
  btnConnect.Enabled := False;
  btnDisconnect.Enabled := True;
  btnSend.Enabled := True;
  Edit1.Enabled := True;
end;
procedure TForm1.btnDisconnectClick(Sender: TObject);
begin
  FServer.Disconnect(FID);
  btnConnect.Enabled := True;
  btnDisconnect.Enabled := False;
  btnSend.Enabled := False;
  Edit1.Enabled := False;
end;
procedure TForm1.btnSendClick(Sender: TObject);
begin
  FServer.SendText(Edit1.Text);
end;
end.
```

Como ha podido ver, podemos manejar la devolución de llamadas a servidores a través de disp-interfaces o de interfaces. Ambos métodos tienen sus ventajas y desventajas. Las disp-interfaces son obligatorias si deseamos que nuestro código funcione, por ejemplo, con Visual Basic. Además, Delphi ofrece un mejor soporte para las disp-interfaces a través del Asistente de Objetos de Automatización. Sin embargo, tendremos que escribir algo de código si queremos que múltiples clientes se conecten a una única instancia de un servidor.

Las interfaces proporcionan algunos beneficios en términos de velocidad, y también nos protegen de las molestias de tener que implementar el método *Invoke* de *IDispatch* y descifrar manualmente los parámetros que se pasan a los manejadores de eventos.

Mi objetivo aquí era simplemente familiarizar al lector con los diferentes mecanismos que pueden ser utilizados para manejar funciones de respuesta en COM, para que éste pueda decidir en cada momento cuál es el que mejor se ajusta a sus necesidades. Dicho esto, debo añadir que si está usando Delphi 5 ó 6, a menos que tenga una buena razón para lo contrario, debe siempre apoyarse en las disp-interfaces y los eventos COM.

En el Capítulo 6 veremos situaciones reales en las cuales implementar eventos COM puede ser extremadamente beneficioso. En la siguiente sección examinaremos un servidor de automatización muy común (Microsoft ADO), y mostraremos cómo escribir un cliente COM en Delphi para acceder a éste.

Microsoft ADO

Casi todas las nuevas tecnologías que provienen de Redmond se basan en COM, y la más reciente tecnología de acceso a bases de datos de Microsoft no es ninguna excepción. *ActiveX Data Objects* (ADO) es la última tecnología de bases de datos de Microsoft, diseñada para acceder a cualquier tipo de datos. Esta sección no pretende ser un tutorial detallado sobre ADO. En lugar de ello, mi deseo es mostrar cómo podemos usar Delphi para de una forma rápida y sencilla obtener las ventajas de las nuevas tecnologías de Windows basadas en COM.

Nota

En la versión 5 de Delphi y posteriores, Borland ofrece componentes para el desarrollo con ADO. Por ello esta sección será de más interés para los usuarios de Delphi 3 y Delphi 4, aunque los conceptos generales relacionados con la implementación de clientes para tecnologías basadas en COM serán útiles a cualquiera.

ADO es uno de los conjuntos de componentes que conforman la estrategia *Universal Data Access* de Microsoft. ADO es el API que los programadores pueden (y deben) utilizar con vistas a acceder a cualquier tipo de datos, desde bases de datos relacionales SQL hasta formatos de ficheros ISAM para acceso a datos no relacionales, como por ejemplo mensajes de correo electrónico. Microsoft ADO es esencialmente un envoltorio construido alrededor de OLE-DB. OLE-DB es un API muy potente, pero difícil de programar. ADO nos libera en gran medida de la complejidad de OLE-DB.

Observe que, dado que ADO es una tecnología de Microsoft, puede estar seguro de que la mayoría de sus esfuerzos han sido concentrados en lograr la ejecución más eficiente posible con Microsoft Access y Microsoft SQL Server. Si el rendimiento es su preocupación esencial, tal vez debería evaluar uno de estos dos sistemas de bases de datos. Sin embargo, según avance el tiempo, podemos esperar que aparecerán proveedores de OLE-DB de otros fabricantes de bases de datos que proporcionen una óptima ejecución sobre sus propias bases de datos.

ADO es un servidor de automatización de interfaz dual. Al momento de escribirse este libro, la última versión de ADO disponible es la 2.6. Sin embargo, ya que ADO 1.1 se instala con el Internet Explorer 4.0 y Windows 98, la mayoría de los lectores tendrán instalado ADO 1.1 en sus ordenadores. Por esta razón, esta sección no examina las nuevas características de ADO 2.6.

La filosofía de ADO

La Figura 4.18 muestra un diagrama conceptual de cómo ADO se relaciona con otras tecnologías como OLE-DB u ODBC.

Figura 4.18: La arquitectura ADO.

Como puede verse, ADO puede obtener datos directamente a través de un proveedor OLE-DB, o indirectamente a través de ODBC. Podríamos alegar que ADO añade otra capa encima de la arquitectura de bases de datos ya existente. Le puedo asegurar que la capa ADO es muy fina y su rendimiento excelente.

ADO introduce el concepto de *proveedor* (*provider*) para brindar el acceso a los datos. Si estamos familiarizados con ODBC, podemos pensar en un proveedor como el equivalente a un controlador ODBC. El proveedor debe exponer al menos un conjunto mínimo de interfaces para que ADO pueda acceder a diferentes tipos de datos de un modo consistente. Un proveedor para un almacén de datos avanzado deberá necesariamente ofrecer más posibilidades que un proveedor para un sistema de ficheros planos, por ejemplo.

Actualmente ADO proporciona acceso para la mayoría de los formatos comunes de bases de datos, pero... ¿qué haríamos si estuviésemos tratando con un fichero con un formato específico? Tal vez seamos los creadores de un nuevo sistema de bases de datos tecnológicamente avanzado. Si disponemos de un controlador ODBC para nuestros datos, podemos usarlo. Sin embargo, podemos también escribir un proveedor OLE-DB para los mismos. Desarrollar un proveedor OLE-DB es un tema avanzado, así que no entraremos en detalles. Sin embargo, la documentación de Microsoft cubre estos temas, y entre los ficheros que ofrece como parte de ADO se incluye un ejemplo de proveedor OLE-DB que podemos estudiar.

Los componentes principales de ADO son *Connection*, *Recordset* y *Command*. El objeto *Connection* es responsable de la conexión a una base de datos local o remota. El objeto *Recordset* proporciona una conexión a un conjunto de registros. El objeto *Command* es útil para enviar a una base de datos comandos que no devuelven un conjunto de resultados, como la instrucción UPDATE de SQL. Estos objetos serán examinados con detalle posteriormente.

Obteniendo ADO

ADO puede descargarse gratuitamente. Visite y seleccione el enlace a la última descarga disponible. Podemos redistribuir este fichero (*MDAC_TYP.EXE*) con nuestro programa de instalación, o los usuarios de la aplicación pueden descargarlo directamente del sitio web de Microsoft. Tanto Windows 98 como Internet Explorer 4.0 incluyen ADO 1.1, que funcionará sin problemas con sus aplicaciones.

Después de descargar los ficheros apropiados del sitio web de Microsoft, simplemente ejecútelo para instalar ADO, conjuntamente con un controlador OLE-DB para el motor Jet de Microsoft, y controladores ODBC para SQL Server, Oracle y Paradox.

Instalando ADO en Delphi

Una vez que tenga ADO instalado en su ordenador, deseará instalarlo en Delphi para poder comenzar a desarrollar aplicaciones ADO. Del menú principal de Delphi seleccione *Project | Import Type Library*. Aparecerá el cuadro de diálogo que se muestra en la Figura 4.19.

Seleccione *Microsoft ActiveX Data Objects 2.1 Library (version 2.1)* y pulse OK. Delphi generará para nosotros la unidad de importación *ADODB_TLB.PAS*. Abra de nuevo el cuadro de diálogo '*Import Type Library*' e importe la librería *Microsoft ActiveX Data Objects Recordset 2.1 Library (Version 2.1)*. Delphi generará otra unidad de importación, en este caso *ADOR_TLB.PAS*. Deberá incluir estas unidades en cualquier aplicación ADO que escriba en Delphi.

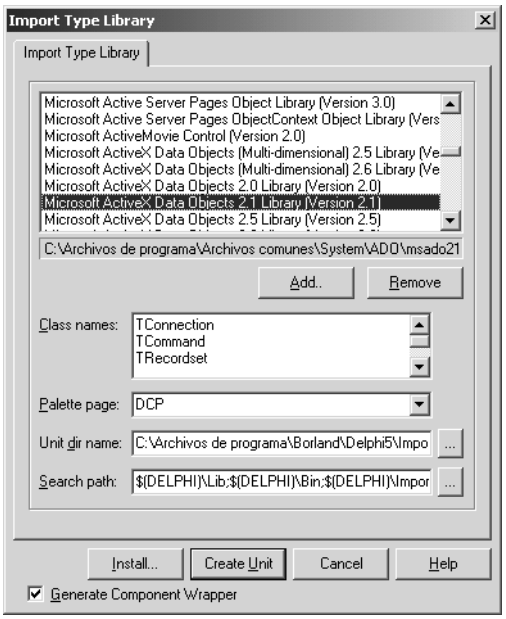


Figura 4.19: Instalando ADO en Delphi.

Conectando a una base de datos

En la terminología ADO, una conexión es aproximadamente el equivalente de un *TDatabase* en Delphi. Al igual que ocurre con los componentes *TDatabase* y *TTable* en Delphi, ADO es lo suficientemente flexible como para permitir establecer una conexión a una base de datos de forma explícita, o obtenerla implícitamente a través de un *recordset*. En la mayoría de las aplicaciones serias desearemos tener un control total sobre las conexiones a las bases de datos, así que vamos a establecerlas explícitamente.

Dependiendo del proveedor a utilizar, podemos conectarnos a una base de datos de varias formas diferentes. Mostraré dos escenarios típicos. En primer lugar, podríamos tener configurada una fuente de datos ODBC para la base de datos. En este caso, podremos conectarnos a la base de datos a través del código siguiente:

```
Connection.Open('Data Source=MyDataSourceName','UserID','Password',-1);
```

Los primeros tres parámetros mostrados en el fragmento de código deberían ser obvios. Pasamos el DSN, el ID de usuario, y la contraseña de acceso. Para un acceso no seguro a una base de datos, el ID de usuario y la contraseña podrían estar en blanco. El último parámetro representa las opciones a utilizar cuando se abre la conexión. Normalmente se asigna a este parámetro el valor *–1* para utilizar las opciones por defecto.

Tabla 4.2: Identificadores válidos para *Connection.Open*

Opción de Conexión	Descripción
CONNECTUNSPECIFIED	Tipo de conexión no especificado.
ASYNCCONNECT	Abre la conexión en modo asíncrono. Cuando la conexión es establecida, se dispara el evento *ConnectComplete* (sólo a partir de ADO 2.0).

El siguiente código es un ejemplo de cómo conectarse a una base de datos Microsoft Jet. Observe que en lugar de conectar a través de un controlador ODBC, la conexión se establece directamente a través del proveedor OLE-DB para Microsoft Jet.

```
Connection.Open(
  'Data Source=C:\MyData.mdb; Provider=Microsoft.Jet.OLEDB.3.51',
  'admin', '', -1);
```

Abriendo un conjunto de datos

Después de haber establecido una conexión a una base de datos, podemos abrir un *recordset*. Un recordset es conceptualmente equivalente a un *TQuery* o un *TTable* de Delphi. Por ejemplo, el siguiente código abre una tabla llamada *Customer* en la base de datos referida por *Connection*.

```
Recordset.Open('Customer', Connection,
  adOpenForwardOnly, adLockReadOnly, adCmdTable);
```

Alternativamente, podemos especificar una instrucción SQL para seleccionar un subconjunto de las filas de la tabla *Customer*, como:

```
Recordset.Open('SELECT * FROM Customer WHERE Balance > 1000',
  Connection, adOpenForwardOnly, adLockReadOnly, adCmdText);
```

Recordset.Open tiene cinco parámetros. El primero se utiliza para determinar qué información contiene el conjunto de datos. En el primer ejemplo, simplemente hemos suministrado un nombre de una tabla de la base de datos. En el segundo ejemplo hemos especificado el conjunto de datos a través de una instrucción SQL.

El segundo parámetro especifica la conexión a utilizar. Si se ha creado explícitamente una conexión, podemos simplemente pasar la conexión como parámetro. Si no hemos creado la conexión, podríamos pasar la información sobre la conexión, como:

```
Recordset.Open('Customer',
  'Data Source=c:\MyData.mdb;Provider=Microsoft.Jet.OLEDB.3.51',
  adOpenForwardOnly, adLockReadOnly, adCmdTable);
```

El tercer parámetro especifica el tipo de cursor a utilizar con la conexión. Los valores posibles se listan en la Tabla 4.3.

Tabla 4.3: Tipos de cursor válidos para *Recordset.Open*

Tipo de cursor	Descripción
adOpenUnspecified	Tipo de cursor no especificado.
adOpenForwardOnly	Lo mismo que un cursor estático, excepto que sólo permite moverse hacia delante en el conjunto de datos. Esto puede mejorar el rendimiento.
adOpenKeySet	Las modificaciones y borrados realizados por otros usuarios son visibles a este conjunto de datos. Las inserciones realizadas por otros usuarios no son visibles.
adOpenDynamic	Las inserciones, modificaciones y borrados realizados por otros usuarios son visibles a este conjunto de datos.
adOpenStatic	Las inserciones, modificaciones y borrados realizados por otros usuarios son visibles a este conjunto de datos.

El cuarto parámetro es el tipo de bloqueo. Los valores posibles se listan en la Tabla 4.4.

Tabla 4.4: Tipos de bloqueo válidos para *Recordset.Open*

Tipo de bloqueo	Descripción
adLockUnspecified	Tipo de bloqueo no especificado.
adLockReadOnly	Sólo lectura – los datos no pueden ser modificados.
adLockPessimistic	El proveedor bloquea el registro tan pronto como comienza la edición.
adLockOptimistic	El proveedor bloquea los datos sólo cuando el método Update es llamado.
adLockBatchOptimistic	Lo mismo que adLockOptimistic, pero requerido para las actualizaciones por lotes.

El quinto y último parámetro especifica el tipo de comando pasado en el primer parámetro. La Tabla 4.5 lista los valores posibles para este parámetro.

Tabla 4.5: Tipos de comando válidos para *Recordset.Open.*

Tipo de Comando	Descripción
adCmdUnspecified	Tipo de comando no especificado.
adCmdUnknown	Tipo de comando desconocido.
adCmdText	El comando se refiere a una sentencia SQL.
adCmdTable	El comando se refiere a un nombre de tabla, donde todas las columnas son devueltas por una sentencia SQL creada internamente.
adCmdStoredProc	El comando se refiere al nombre de un procedimiento almacenado.

Tipo de Comando	Descripción
adCmdFile	El comando se refiere al nombre de fichero de un conjunto de datos persistente.
adCmdTableDirect	El comando se refiere al nombre de una tabla, donde todas las columnas son devueltas.
adExecuteNoRecords	El comando se refiere a un comando o procedimiento almacenado que no devuelve un conjunto de datos. Este valor siempre se utiliza conjuntamente con adCmdText o adCmdStoredProc, combinados mediante una operación lógica OR.

Ejecutando un comando

Si necesitamos ejecutar una acción sobre la base de datos que no devuelva un conjunto de datos, debemos usar un objeto *Command* en lugar de un objeto *Recordset*. El objeto *Command* trabaja mano a mano con el objeto *Parameter*, así que los examinaremos conjuntamente. El siguiente fragmento de código muestra una forma de aumentar el sueldo un 10% a cada miembro de la tabla *Employee*:

```
Parameter := Command.CreateParameter('Raise',
  adDouble, adParamInput, sizeof(Double),  1.10);
Command.CommandText := 'UPDATE Employee SET Salary = Salary * ?';
Command.Execute(RecsAffected, Parameter, -1);
```

El primer parámetro es el nombre del objeto *Parameter*. Le podemos dar cualquier nombre que nos resulte razonable. El segundo parámetro es el tipo de datos del parámetro. Hay demasiados tipos de datos posibles para listarlos todos aquí. Consulte la documentación de Microsoft o los ficheros de importación generados por Delphi para ver un listado de los posibles valores.

El tercer parámetro indica el tipo de parámetro. Los valores posibles se muestran en la Tabla 4.6.

Tabla 4.6: Tipos de parámetros válidos para *Command.CreateParameter*

Tipo de parámetro	Descripción
adParamUnknown	La dirección del parámetro es desconocida.
adParamInput	El parámetro es un parámetro de entrada.
adParamOutput	El parámetro es un parámetro de salida.
adParamInputOutput	El parámetro es de entrada y salida.
adParamReturnValue	El parámetro es un valor de retorno.

El cuarto parámetro especifica la longitud máxima del parámetro, en caracteres o bytes, dependiendo del tipo de parámetro. El quinto y último parámetro es una variante que especifica el valor del parámetro.

Accediendo a los valores de los campos

ADO crea un objeto *Field* para cada columna del conjunto de datos. Para obtener datos de un conjunto de datos o modificar el valor de un campo, debemos acceder al valor de los objetos *Field* correspondientes, como en:

```
Recordset.Fields.Item(FieldNo).Value
```

Podemos obtener el nombre del campo mediante la propiedad *Name*:

```
Recordset.Fields.Item(FieldNo).Name
```

El objeto *Field* ofrece varias propiedades útiles. Alguna de las más frecuentemente utilizadas se listan en la Tabla 4.7.

Tabla 4.7: Propiedades comunes del objeto *Field*

Propiedad	Descripción
Name	El nombre del campo de la base de datos.
Precision	El número máximo de dígitos utilizados para representar valores numéricos.
Type	El tipo de datos del campo. Consulte la documentación de Microsoft para un listado de los tipos de datos soportados.
Value	El valor del campo.

Manejando errores de bases de datos

En toda programación de base de datos debe estar preparado para manejar los errores que puedan ocurrir, y ADO no es la excepción. ADO ofrece el objeto *Errors*, que proporciona información detallada de todos y cada uno de los posibles errores. Gracias al mecanismo de excepciones de Delphi, manejar los errores de ADO es muy sencillo. El siguiente fragmento de código muestra cómo interceptar y responder a un error de ADO:

```
try
  FRecordset.MoveNext;
except
  E := FConnection.Errors[0];
  // hacer algo con el error aqu□ □ por ejemplo,
  // registrarlo en un fichero
  raise; // volver a elevar la excepci□n
end;
```

Como probablemente habrá supuesto del fragmento de código anterior, es posible que ADO genere múltiples errores simultáneamente. Para poder comunicar al usuario toda la información sobre los errores, se debe examinar la propiedad *FConnection.Errors.Count*, que determina la cantidad de errores generados. Posteriormente debemos implementar un mecanismo que permita al usuario desplazarse a través de la lista de errores, que estarán almacenados en *FConnection.Errors[0* hasta *FConnection.Errors[Count – 1]*. El

Listado 4.17 contiene un procedimiento llamado *ShowADOErrors* que muestra una manera muy simple de lograr esto.

Ejemplo: aplicación Microsoft ADO

Esta aplicación de ejemplo no hace nada realmente útil; es únicamente una colección de procedimientos que ejecutan las funciones típicas de ADO.

Listado 4.17: ADOTest - *MainForm.pas*

```
unit MainForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ADODB_TLB, Grids;
const
  ConnectionString =
    'Data Source=' +
    'C:\Archivos de programa\Common Files\Borland
Shared\Data\DBDemos.mdb;' +
    'Provider=Microsoft.Jet.OLEDB.4.0';
type
  TForm1 = class(TForm)
    StringGrid1: TStringGrid;
    btnLoadGrid: TButton;
    btnAdd: TButton;
    btnAddDuplicate: TButton;
    btnDelete: TButton;
    btnSearch: TButton;
    btnSort: TButton;
    procedure FormCreate(Sender: TObject);
    procedure btnLoadGridClick(Sender: TObject);
    procedure btnAddClick(Sender: TObject);
    procedure btnAddDuplicateClick(Sender: TObject);
    procedure btnDeleteClick(Sender: TObject);
    procedure btnSearchClick(Sender: TObject);
    procedure btnSortClick(Sender: TObject);
  private
    { Private declarations }
    FConnection: _Connection;
    FRecordset: _Recordset;
    FByOrder: Boolean;
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure ShowADOErrors(ErrorList: Errors);
var
  I: Integer;
```

```
  E: Error;
  S: string;
begin
  for I := 0 to ErrorList.Count - 1 do begin
    E := ErrorList[I];
    S := Format('ADO Error %d of %d:'#13#13'%s',
      [I + 1, ErrorList.Count, E.Description]);
    ShowMessage(S);
  end;
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  FConnection := CoConnection.Create;
  FConnection.Open(ConnectionString, 'admin', '', -1);
  FRecordset := CoRecordset.Create;
  FRecordset.Open('SELECT * FROM Orders ORDER BY OrderNo',
    FConnection, adOpenKeyset, adLockOptimistic, adCmdText);
  FByOrder := True;
  btnLoadGridClick(Sender);
end;
procedure TForm1.btnLoadGridClick(Sender: TObject);
var
  V: OleVariant;
  S: string;
  Row, Col: Integer;
begin
  try
    FRecordset.MoveFirst;
    StringGrid1.ColCount := FRecordset.Fields.Count;
    StringGrid1.RowCount := FRecordset.RecordCount + 1;
    for Col := 0 to FRecordset.Fields.Count - 1 do
      StringGrid1.Cells[Col, 0] := FRecordset.Fields.Item[Col].Name;
    Row := 1;
    while not FRecordset.EOF do begin
      for Col := 0 to FRecordset.Fields.Count - 1 do begin
        V := FRecordset.Fields.Item[Col].Value;
        if VarType(V)  varNull then
          S := V
        else
          S := '';
        StringGrid1.Cells[Col, Row] := S;
      end;
      Inc(Row);
      FRecordset.MoveNext;
    end;
  except
    ShowADOErrors(FConnection.Errors);
  end;
end;
procedure TForm1.btnAddClick(Sender: TObject);
var
  Fields: OleVariant;
```

```
  Values: OleVariant;
begin
  Fields := VarArrayCreate([1, 3], varVariant);
  Values := VarArrayCreate([1, 3], varVariant);
  try
    Fields[1] := 'OrderNo';
    Fields[2] := 'CustNo';
    Fields[3] := 'EmpNo';
    Values[1] := 99999;
    Values[2] := 3615;
    Values[3] := 2;
    FRecordset.AddNew(Fields, Values);
    btnLoadGridClick(Sender);
    ShowMessage('OrderNo 99999 was added');
  except
    ShowADOErrors(FConnection.Errors);
  end;
end;
procedure TForm1.btnAddDuplicateClick(Sender: TObject);
var
  Fields: OleVariant;
  Values: OleVariant;
begin
  Fields := VarArrayCreate([1, 3], varVariant);
  Values := VarArrayCreate([1, 3], varVariant);
  try
    Fields[1] := 'OrderNo';
    Fields[2] := 'CustNo';
    Fields[3] := 'EmpNo';
    Values[1] := 1035;
    Values[2] := 1645;
    Values[3] := 2;
    FRecordset.AddNew(Fields, Values);
  except
    ShowADOErrors(FConnection.Errors);
  end;
end;
procedure TForm1.btnDeleteClick(Sender: TObject);
begin
  FRecordset.MoveFirst;
  FRecordset.Find('OrderNo=99999', 0, adSearchForward, 0);
  if not FRecordset.EOF then begin
    FRecordset.Delete(adAffectCurrent);
    btnLoadGridClick(Sender);
    ShowMessage('OrderNo 99999 was deleted.');
  end else
    ShowMessage('OrderNo 99999 was not found.');
end;
procedure TForm1.btnSearchClick(Sender: TObject);
begin
  FRecordset.MoveFirst;
  FRecordset.Find('OrderNo=1070', 0, adSearchForward, 0);
```

```
  if not FRecordset.EOF then
    ShowMessage('OrderNo 1070 was found.')
  else
    ShowMessage('OrderNo 1070 was not found.')
end;
procedure TForm1.btnSortClick(Sender: TObject);
begin
  FByOrder := not FByOrder;
  FRecordset.Close;
  if FByOrder then
    FRecordset.Open('SELECT * FROM Orders ORDER BY OrderNo',
      FConnection, adOpenKeyset, adLockOptimistic, adCmdText)
  else
    FRecordset.Open('SELECT * FROM Orders ORDER BY CustNo, OrderNo',
      FConnection, adOpenKeyset, adLockOptimistic, adCmdText);
  btnLoadGridClick(Sender);
end;
end.
```

Cuando ejecutamos *ADOTest*, veremos la pantalla que se muestra en la Figura 4.20.

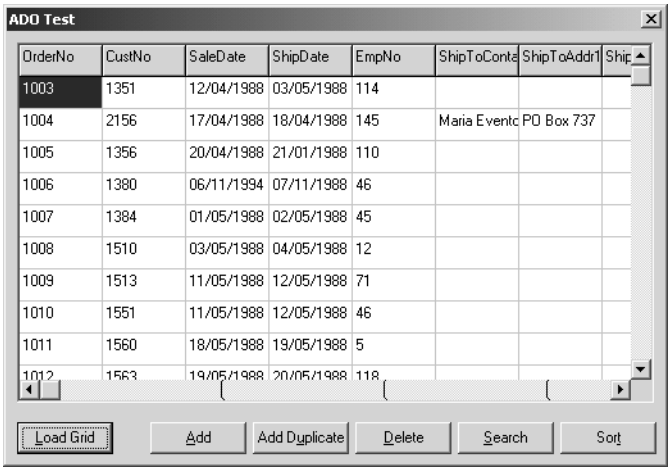


Figura 4.20: ADOTest muestra cómo ejecutar las tareas comunes de acceso a datos.

El formulario de la aplicación tiene una rejilla de cadenas de caracteres y seis botones. De izquierda a derecha, los botones ejecutan las siguientes operaciones:

- **Load Grid**: Fuerza a la aplicación a recargar todos los datos del *recordset*.
- **Add**: Añade un nuevo pedido a la tabla de pedidos. Como código del nuevo pedido se asigna 99999, el código de cliente a 3615, y el de empleado a 2. Este registro no existe inicialmente en la base de datos, así que esta rutina demuestra una adición exitosa a la base de datos.

- **Add Duplicate**: Intenta añadir un nuevo registro con un código de pedido ya existente (1035). Esta rutina demuestra un error de clave duplicada.
- **Delete**: Trata de localizar y borrar el pedido con código 99999. Si ya se ha añadido el pedido 99999, esta rutina demuestra cómo localizar y borrar un registro.
- **Search:** Utiliza el método *Find* para localizar el pedido con código 1070.
- **Sort**: Cambia entre la ordenación por código de pedido o por código de cliente.

Resumen

En este capítulo hemos mostrado cómo crear servidores de automatización COM internos al proceso y externos al proceso. Hemos visto cómo las bibliotecas de tipos juegan un papel esencial en la creación de servidores de automatización. Este capítulo reúne todos los temas estudiados en capítulos anteriores.

Hemos examinado también el rol que juegan las variantes en la automatización COM, y cómo utilizarlas para permitir el enlace tardío a un objeto de automatización.

Después de presentar las bases de los servidores de automatización, hemos mostrado cómo implementar los eventos COM y las funciones de respuesta de dos formas diferentes. Ha aprendido que implementar los eventos a través de disp-interfaces es el método más compatible a utilizar, pero crear interfaces *callback* puede ser más eficiente si conocemos que el servidor y el cliente proporcionan soporte de interfaz. Delphi 5 y 6 mejoran el manejo de eventos COM, haciendo obsoleta parte de esta sección**.**

Finalmente, hemos concluido el capítulo con una sección acerca de Microsoft ADO. El objetivo de esta sección no fue hacer de Usted un programador experto en ADO, sino mostrarle cómo es posible utilizar Delphi para de una forma relativamente sencilla acceder a las ventajas de la tecnología COM. Delphi 5 y 6 proporcionan componentes ADO que deben usarse para desarrollar aplicaciones que hagan uso de ADO, pero el concepto general de la importación de bibliotecas de tipos dentro de Delphi se aplica a los servidores de automatización en general.

Después de leer este capítulo deberá conocer las diferencias entre interfaces, disp-interfaces y variantes, y deberá estar armado con los conocimientos necesarios para determinar cuál de estos tres métodos de acceso es el mejor para una situación particular.

En el próximo capítulo examinaremos los controles ActiveX y los formularios activos (*Active Forms*). Aprenderá a convertir un control VCL en un control ActiveX. También le mostraremos cómo crear una aplicación sencilla y distribuirla como un *ActiveForm*.

- **Add Duplicate**: Intenta añadir un nuevo registro con un código de pedido ya existente (1035). Esta rutina demuestra un error de clave duplicada.
- **Delete**: Trata de localizar y borrar el pedido con código 99999. Si ya se ha añadido el pedido 99999, esta rutina demuestra cómo localizar y borrar un registro.
- **Search:** Utiliza el método *Find* para localizar el pedido con código 1070.
- **Sort**: Cambia entre la ordenación por código de pedido o por código de cliente.

Resumen

En este capítulo hemos mostrado cómo crear servidores de automatización COM internos al proceso y externos al proceso. Hemos visto cómo las bibliotecas de tipos juegan un papel esencial en la creación de servidores de automatización. Este capítulo reúne todos los temas estudiados en capítulos anteriores.

Hemos examinado también el rol que juegan las variantes en la automatización COM, y cómo utilizarlas para permitir el enlace tardío a un objeto de automatización.

Después de presentar las bases de los servidores de automatización, hemos mostrado cómo implementar los eventos COM y las funciones de respuesta de dos formas diferentes. Ha aprendido que implementar los eventos a través de disp-interfaces es el método más compatible a utilizar, pero crear interfaces *callback* puede ser más eficiente si conocemos que el servidor y el cliente proporcionan soporte de interfaz. Delphi 5 y 6 mejoran el manejo de eventos COM, haciendo obsoleta parte de esta sección**.**

Finalmente, hemos concluido el capítulo con una sección acerca de Microsoft ADO. El objetivo de esta sección no fue hacer de Usted un programador experto en ADO, sino mostrarle cómo es posible utilizar Delphi para de una forma relativamente sencilla acceder a las ventajas de la tecnología COM. Delphi 5 y 6 proporcionan componentes ADO que deben usarse para desarrollar aplicaciones que hagan uso de ADO, pero el concepto general de la importación de bibliotecas de tipos dentro de Delphi se aplica a los servidores de automatización en general.

Después de leer este capítulo deberá conocer las diferencias entre interfaces, disp-interfaces y variantes, y deberá estar armado con los conocimientos necesarios para determinar cuál de estos tres métodos de acceso es el mejor para una situación particular.

En el próximo capítulo examinaremos los controles ActiveX y los formularios activos (*Active Forms*). Aprenderá a convertir un control VCL en un control ActiveX. También le mostraremos cómo crear una aplicación sencilla y distribuirla como un *ActiveForm*.

- **Add Duplicate**: Intenta añadir un nuevo registro con un código de pedido ya existente (1035). Esta rutina demuestra un error de clave duplicada.
- **Delete**: Trata de localizar y borrar el pedido con código 99999. Si ya se ha añadido el pedido 99999, esta rutina demuestra cómo localizar y borrar un registro.
- **Search:** Utiliza el método *Find* para localizar el pedido con código 1070.
- **Sort**: Cambia entre la ordenación por código de pedido o por código de cliente.

Resumen

En este capítulo hemos mostrado cómo crear servidores de automatización COM internos al proceso y externos al proceso. Hemos visto cómo las bibliotecas de tipos juegan un papel esencial en la creación de servidores de automatización. Este capítulo reúne todos los temas estudiados en capítulos anteriores.

Hemos examinado también el rol que juegan las variantes en la automatización COM, y cómo utilizarlas para permitir el enlace tardío a un objeto de automatización.

Después de presentar las bases de los servidores de automatización, hemos mostrado cómo implementar los eventos COM y las funciones de respuesta de dos formas diferentes. Ha aprendido que implementar los eventos a través de disp-interfaces es el método más compatible a utilizar, pero crear interfaces *callback* puede ser más eficiente si conocemos que el servidor y el cliente proporcionan soporte de interfaz. Delphi 5 y 6 mejoran el manejo de eventos COM, haciendo obsoleta parte de esta sección**.**

Finalmente, hemos concluido el capítulo con una sección acerca de Microsoft ADO. El objetivo de esta sección no fue hacer de Usted un programador experto en ADO, sino mostrarle cómo es posible utilizar Delphi para de una forma relativamente sencilla acceder a las ventajas de la tecnología COM. Delphi 5 y 6 proporcionan componentes ADO que deben usarse para desarrollar aplicaciones que hagan uso de ADO, pero el concepto general de la importación de bibliotecas de tipos dentro de Delphi se aplica a los servidores de automatización en general.

Después de leer este capítulo deberá conocer las diferencias entre interfaces, disp-interfaces y variantes, y deberá estar armado con los conocimientos necesarios para determinar cuál de estos tres métodos de acceso es el mejor para una situación particular.

En el próximo capítulo examinaremos los controles ActiveX y los formularios activos (*Active Forms*). Aprenderá a convertir un control VCL en un control ActiveX. También le mostraremos cómo crear una aplicación sencilla y distribuirla como un *ActiveForm*.

- **Add Duplicate**: Intenta añadir un nuevo registro con un código de pedido ya existente (1035). Esta rutina demuestra un error de clave duplicada.
- **Delete**: Trata de localizar y borrar el pedido con código 99999. Si ya se ha añadido el pedido 99999, esta rutina demuestra cómo localizar y borrar un registro.
- **Search:** Utiliza el método *Find* para localizar el pedido con código 1070.
- **Sort**: Cambia entre la ordenación por código de pedido o por código de cliente.

Resumen

En este capítulo hemos mostrado cómo crear servidores de automatización COM internos al proceso y externos al proceso. Hemos visto cómo las bibliotecas de tipos juegan un papel esencial en la creación de servidores de automatización. Este capítulo reúne todos los temas estudiados en capítulos anteriores.

Hemos examinado también el rol que juegan las variantes en la automatización COM, y cómo utilizarlas para permitir el enlace tardío a un objeto de automatización.

Después de presentar las bases de los servidores de automatización, hemos mostrado cómo implementar los eventos COM y las funciones de respuesta de dos formas diferentes. Ha aprendido que implementar los eventos a través de disp-interfaces es el método más compatible a utilizar, pero crear interfaces *callback* puede ser más eficiente si conocemos que el servidor y el cliente proporcionan soporte de interfaz. Delphi 5 y 6 mejoran el manejo de eventos COM, haciendo obsoleta parte de esta sección**.**

Finalmente, hemos concluido el capítulo con una sección acerca de Microsoft ADO. El objetivo de esta sección no fue hacer de Usted un programador experto en ADO, sino mostrarle cómo es posible utilizar Delphi para de una forma relativamente sencilla acceder a las ventajas de la tecnología COM. Delphi 5 y 6 proporcionan componentes ADO que deben usarse para desarrollar aplicaciones que hagan uso de ADO, pero el concepto general de la importación de bibliotecas de tipos dentro de Delphi se aplica a los servidores de automatización en general.

Después de leer este capítulo deberá conocer las diferencias entre interfaces, disp-interfaces y variantes, y deberá estar armado con los conocimientos necesarios para determinar cuál de estos tres métodos de acceso es el mejor para una situación particular.

En el próximo capítulo examinaremos los controles ActiveX y los formularios activos (*Active Forms*). Aprenderá a convertir un control VCL en un control ActiveX. También le mostraremos cómo crear una aplicación sencilla y distribuirla como un *ActiveForm*.

5

Controles ActiveX y ActiveForms

En este capítulo examinamos los controles ActiveX y los formularios activos (ActiveForms), y le enseñaremos cómo utilizar controles ActiveX existentes en sus aplicaciones Delphi, y cómo crear sus propios controles ActiveX y ActiveForms.

Usando controles ActiveX existentes en Delphi

Actualmente existe una enorme cantidad de controles ActiveX disponibles en el mercado. Existen controles ActiveX para la gran mayoría de los propósitos que podamos imaginar y también para algunos en lo que jamás hemos imaginado.

En esta sección presentaremos un control ActiveX de distribución libre – el Microsoft Agent, del cual probablemente nunca haya oído antes.

Microsoft Agent

Microsoft Agent es una *suite* de DLLs que pueden instalarse en cualquier versión de Windows. Estas DLLs implementan un conjunto de controles ActiveX que permiten controlar a personajes similares a los de dibujos animados, llamados *agentes*. Los agentes tienen capacidades de movimiento, habla y audición. Pueden ser utilizados para casi cualquier cosa que pueda imaginar, aunque probablemente la principal aplicación de Microsoft Agent es en tutoriales o contexto de ayuda dentro de las aplicaciones.

Obteniendo Microsoft Agent

Obtener una copia de Microsoft Agent es fácil. Simplemente visite la página Web *http://msdn.microsoft.com/msagent*.

Pulse en el enlace **Download Microsoft Agent**. Microsoft Agent se divide en varios "componentes". Primero necesitará descargar **Microsoft Agent Core Components.** Seguidamente, descargue uno o más ficheros de personajes en **Microsoft Agent character files.** Finalmente, deberá descargar el Panel de Control de Agentes, mediante el cual se controlan los parámetros de configuración

del habla**,** tales como velocidad, volumen y otros, utilizando el enlace **Download the Speech Control Panel.**

Microsoft proporciona cuatro agentes:

- Genie (un genio)
- Merlin (un mago)
- Robbie (un robot)
- Peedy (un loro)

Espero que con el tiempo Microsoft vaya creando agentes adicionales, y podrán encontrarse también ofertas de terceras partes. En la aplicación ejemplo de este capítulo utilizaremos al genio, así que asegúrese de descargar al menos el genio.

Si desea tener la posibilidad de oír la voz del agente a través de su tarjeta de sonido, descargue el **Lernout & Hauspie TruVoice Text-To-Speech Engine.**

Le recomiendo que siga el enlace '**Go to the Microsoft Agent Downloads for Developers page**', y descargue la documentación completa de Microsoft Agent. La documentación contiene una referencia completa de todos los aspectos de la programación con Microsoft Agent.

Después de haber descargado los ficheros adecuados, ejecute MSAGENT.EXE para instalar Microsoft Agent. Este programa creará el núcleo de los ficheros dentro de *C:\Archivos de programa\Microsoft Agent*. A continuación ejecute TV_ENUA.EXE, que instalará el motor del habla. Ejecute GENIE.EXE para instalar el agente *Genie*, y para finalizar, ejecute SPCHCPL.EXE para instalar el *applet* de control del habla en el Panel de Control**.**

Nota

Con Microsoft Agent también es posible crear nuestros propios agentes. Para ello deberá descargar el Editor de Personajes (Microsoft Agent Character Editor) y el Editor de Información Lingüística (Microsoft Agent Linguistic Information Sound Editing Tool). Esta tarea requiere un tiempo y esfuerzo significativos, así como talento artístico.

Instalando Microsoft Agent en Delphi.

Para instalar Microsoft Agent en Delphi, siga los siguientes pasos:

1. Seleccione del menú principal de Delphi *Component | Import ActiveX Control.*
2. Seleccione Microsoft Agent Control 2.0 (Version 2.0), y pulse el botón *Install...* (ver Figura 5.1).
3. Seleccione *Into new package*, e introduzca un nombre de fichero para el nuevo paquete (por ejemplo, *MSAgent.DPK*), escriba una descripción, y pulse OK.

El control Microsoft Agent ActiveX está ahora instalado en Delphi.

Figura 5.1: Instalando el control Microsoft Agent en Delphi.

Programación con Microsoft Agent

Los Listados 5.1 y 5.2 contienen el código fuente de un programa muy básico que usa Microsoft Agent. Al ejecutar la aplicación, si todo ha sido instalado correctamente, verá que un genio envuelto en una nube de humo aparece en la esquina superior izquierda de su pantalla y le oirá decir 'Hello'. También verá que la palabra 'Hello' aparece en un globo de texto sobre la cabeza del genio. Cuando se cierra la aplicación, el genio se marcha. No es muy excitante, pero es un comienzo.

La Figura 5.2 muestra la aplicación *AgentTest* en tiempo de ejecución.

Listado 5.1: AgentTest - *AgentTest.dpr*

```
program AgentTest;
uses
  Forms,
  MainForm in 'MainForm.pas' {Form1};
{$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Figura 5.2: Probando Microsoft Agent.

Listado 5.2: AgentTest - *MainForm.pas*

```
unit MainForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  OleCtrls, AgenTObjects_TLB;
type
  TForm1 = class(TForm)
    Agent1: TAgent;
    procedure FormCreate(Sender: TObject);
  private
    procedure WaitFor(Request: IAgentCtlRequest);
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
const
  AGENT = 'genie';
  AGENTPATH = 'C:\WinNT\MSAgent\Chars\genie.acs';
procedure TForm1.WaitFor(Request: IAgentCtlRequest);
var
  Status: LongInt;
begin
  repeat
    Application.ProcessMessages;
    Status := Request.Get_Status;
  until (Status  2) and (Status  4);
```

```
end;
procedure TForm1.FormCreate(Sender: TObject);
var
  Character: IAgentCtlCharacter;
  Request: IAgentCtlRequest;
begin
  Agent1.Connected := True;
  Request := Agent1.Characters.Load(AGENT, AGENTPATH);
  Character := Agent1.Characters.Character(AGENT);
  Request := Character.Show(False);
  WaitFor(Request);
  Request := Character.Speak('Hello', EmptyParam);
end;
end.
```

Observe que, debido a consideraciones de espacio, no presentamos el código de *AgentObjects_TLB.pas*. Este fichero es generado automáticamente por Delphi cuando se instala el control ActiveX.

Controles de Usuario

Además de lo que como programador pueda hacer, existen otras interacciones posibles de un usuario final con Microsoft Agent. Si se ha instalado el *applet* del Panel de Control del Habla, estará disponible un nuevo icono '*Speech*' en el Panel de Control de Windows. Haciendo doble clic sobre el icono aparecerá el cuadro de diálogo de propiedades del habla (*Speech Properties*).

Figura 5.3: Cuadro de diálogo '*Speech Properties*'.

Mediante este cuadro de diálogo el usuario puede controlar la velocidad, el tono y el volumen del habla del agente.

También el usuario puede mover el agente por la pantalla simplemente pulsándolo y arrastrándolo sobre ésta. Esto es importante, porque los agentes aparecen siempre por encima de otras aplicaciones que se estén ejecutando. Podría ser necesario mover el agente para ver qué hay detrás de él.

Este control por parte del usuario significa que Ud. debe ser conciente no sólo de lo que está haciendo su aplicación, sino que también deberá de algún modo seguir las interacciones del usuario con el agente. Afortunadamente, Microsoft Agent proporciona un número de eventos que se disparan cuando el usuario interactúa con el agente. Más adelante en este capítulo examinaremos estos eventos, en la sección titulada "Notificación de Eventos".

Peticiones del Agente

Microsoft Agent hace uso extensivo de una característica conocida como *peticiones* (*requests*). Una petición no es más que una técnica de sincronización que permite a las aplicaciones determinar cuándo un comando dado ha finalizado. La mayoría de los comandos de un agente son asíncronos (o sea, que devuelven el control al programa antes de que el comando haya sido completado), así que la aplicación debe tener la manera de determinar cuándo el comando ha sido ejecutado. Suponga que deseamos que el agente se mueva a un punto determinado de la pantalla, apunte a un botón e informe al usuario que debe pulsar éste para llevar a cabo una acción específica. Podríamos desear que el agente no hable hasta después que haya arribado a su punto de destino en la pantalla. Utilizando una petición, podemos esperar a que el agente finalice la ejecución de la orden de movimiento antes de emitir el comando de hablar.

Hay dos maneras de utilizar las peticiones. La primera y más simple desde el punto de vista de la codificación, es esperar hasta que la operación haya finalizado antes de continuar con la ejecución del programa. A este método yo lo denomino operación *WaitFor* (*Espera*), y podemos implementarlo de la siguiente forma:

```
procedure TForm.WaitFor(Request: IAgentCtlRequest)
var
 Status: LongInt;
begin
  repeat
    Application.ProcessMessages;
    Request.Get_Status(Status);
  until (Status  2) and (Status  4);
end;
```

El siguiente seudocódigo ilustra cómo utilizar esta técnica:

```
Moverse a posici□n en pantalla
Esperar finalizaci□n de la petici□n
Hablar
```

Aunque es sencillo de implementar, este método no hace el uso más efectivo del sistema. El programa simplemente se detiene mientras espera a que se complete la petición. Sin embargo, podría ser adecuado en ciertas aplicaciones, así que mostraré su utilización en el Listado 5.2.

El segundo y más robusto método de seguimiento de peticiones consiste en manejar un evento que es lanzado por Microsoft Agent cuando las peticiones son completadas. En este caso, Microsoft Agent desencadenará el evento *OnRequestComplete*, indicando que el agente ha finalizado su movimiento. Cuando nuestro programa reciba el evento, indicará al agente que comience a hablar. Discutiremos este evento, así como algunos otros, más adelante en este capítulo, en la sección "Notificaciones de Eventos".

Si se va a utilizar este último método, no se deben declarar las variables de peticiones como locales. Se debe asegurar que el identificador de la petición esté disponible al gestor del evento cuando éste se produzca. Prefiero declarar las variables de peticiones en la declaración del formulario, de tal forma que sean accesibles durante todo el tiempo de vida del formulario.

Las peticiones son también críticas cuando se trata de sincronizar dos agentes, por ejemplo cuando se esté produciendo una conversación entre ellos. Si no se desea que un agente interrumpa constantemente al otro, se deberá esperar a que el primer agente termine de hablar antes indicar al segundo agente que comience.

Volvamos al primer ejemplo y examinemos el código escrito en el manejador del evento *FormCreate*. Primero se realiza la asignación `Agent1.Connected := True`. Esto inicializa el control ActiveX Microsoft Agent para nuestra aplicación. Seguidamente, se inicializa una variante para que almacene la ruta del directorio donde están situados "los personajes" en el ordenador. En una aplicación real, probablemente se permitirá al usuario indicar la ruta de los personajes, o se podrían instalar personajes en el directorio la aplicación.

A continuación, se carga al genio, y se obtiene una referencia a éste, que se almacena en la variable *Character*. Esta es toda la inicialización que se necesita para comenzar a usar el agente. En este punto, la aplicación presenta al genio en pantalla, espera a que haya sido completamente dibujado y le indica que hable. El comando *Show* tiene dos parámetros: el primero indica si el agente debe ser mostrado inmediatamente, o si éste debe mostrarse en modo animado y el segundo es opcional. En el ejemplo, hemos asignado *False* al parámetro, por lo que el genio aparecerá "lentamente". Si se utiliza

```
Request := Character.Show(True);
```

entonces el genio se mostrará inmediatamente, sin ninguna fanfarria. Es más rápido, pero menos interesante.

Comandos comunes del agente

Microsoft Agent entiende un número de comandos diferentes. La Tabla 5.1 lista algunos de los comandos que usaremos frecuentemente:

Tabla 5.1: Comandos de Microsoft Agent utilizados frecuentemente

Comando	Descripción
GestureAt	Apunta a una localización de la pantalla
Hide	Oculta el agente
MoveTo	Mueve el agente a una localización de la pantalla
Play	Ejecuta una animación
Show	Muestra el agente
Speak	Habla
Stop	Detiene una animación en curso

Show y *Hide* se utilizan para mostrar y ocultar el agente. Recuerde que el usuario también puede mostrar y ocultar el agente directamente a través del menú contextual asociado con el icono, así que se debe seguir de cerca el estado de visibilidad del agente.

El comando *Show* ya ha sido presentado. *Hide* simplemente oculta (esconde) al agente. Esta acción no destruye al agente ni cierra Microsoft Agent. Este método tiene dos parámetros, una opción de acción inmediata y un identificador de petición, y su empleo se corresponde con el comando *Show*.

El comando *MoveTo* es utilizado (obviamente) para mover al agente alrededor de la pantalla. *MoveTo* tiene tres argumentos. Su sintaxis es:

```
function MoveTo(x: Smallint; y: Smallint;
  Speed: OleVariant): IAgentCtlRequest; safecall;
```

Los argumentos *x* e *y* especifican las coordenadas de pantalla a donde se moverá el agente. *Speed*, expresada en milisegundos, determina cuán rápido el agente se moverá a su nueva posición. Mientras se mueve, la animación de movimiento se ejecuta automáticamente. Un valor cero para *Speed* hace que el agente arribe a su destino inmediatamente, sin que se ejecute ninguna animación. Valores grandes para *Speed* provocan que el agente se desplace más lentamente. Intentar mover el agente o ejecutar otro tipo de animación cuando el agente no está mostrado en pantalla origina un error. Las animaciones se discuten en detalle posteriormente en este capítulo, en la sección titulada "Animación".

GestureAt le indica al agente que apunte a una localización específica en la pantalla. Este método tiene tres parámetros: *x*, *y* y un identificador de petición.

Observe que para *MoveTo* y *GestureAt*, al igual que para otros comandos de posición, los valores *x* e *y* se especifican en coordenadas de pantalla. Esto significa que podemos indicar al agente que apunte o se mueva a zonas de la pantalla que no están ocupadas por nuestra aplicación.

Capacidades de voz del agente

Una de las características básicas de Microsoft Agent es su habilidad de hablar al usuario a través del comando *Speak*. El siguiente fragmento de código muestra cómo utilizar el comando *Speak*:

```
Request := Character.Speak('Hello', nil);
```

El primer argumento es una variante, que especifica el texto que el agente dirá. El segundo argumento es también una variante, y especifica un fichero de audio a utilizar para la salida. Aquí puede utilizarse tanto una URL como un nombre de fichero. Ambos parámetros son considerados opcionales, pero se debe especificar un valor al menos para uno de ellos. Normalmente usaremos únicamente el primer parámetro. Este suministra el texto que será utilizado en la conversación y mostrado en el globo. Si se especifica un valor para ambos parámetros, entonces el primer parámetro se mostrará en el globo, mientras que el segundo indicará el fichero de audio a reproducir en lugar del habla sintetizada.

No olvide que los usuarios de la aplicación pueden configurar Microsoft Agent a su gusto. Podrían tener desactivado el globo de texto, no haber instalado el controlador de conversión de texto a voz, o ser sordos. Su programa siempre deberá ofrecer tanto texto como voz, permitiendo al usuario decidir si desea activar uno solo de ellos o ambos.

El primer argumento puede contener una o más *marcas* (*tags*). Una marca es un indicador que le indica al agente que realice algo especial durante la conversación. Las marcas se insertan directamente en las cadenas de voz usando un carácter de barra invertida ('\'). Por ejemplo, mediante una marca *Chr* podemos configurar el carácter de la voz a normal, monótona o susurro. Por ejemplo, la cadena

```
\Chr="Whisper"\Estoy susurrando
```

será reproducida en forma de un susurro. La Tabla 5.2 lista algunas marcas comunes y sus usos.

Tabla 5.2: Marcas de habla comunes de Microsoft Agent

Marca	Descripción	Ejemplo
Chr	Controla el "carácter" de la voz como normal, monótona, o en susurro. Su sintaxis es: `\Chr="Car□cter"\`	\Chr="Whisper"\Susurrando
Emp	Enfatiza la siguiente palabra. Su sintaxis es: `\Emp\`	Esto es un \Emp\botón
Map	Mapea el texto que se habla a un globo de texto. Esto le permite al agente decir una cosa mientras que en el texto se muestra otra. Su sintaxis es: `\Map="TextoHabla"="TextoGlobo"\`	Esto es \Map="dicho"="mostrado"\
Mrk	Asigna una marca (*bookmark*) a una posición en el texto. Se disparará un evento *OnBookmark* cuando se alcance ese punto en el texto. Su sintaxis es: `\Mrk=number\`	Hazme saber cuando lleguemos \Mrk=1\aquí
Pau	Hace una pausa durante un número específico de milisegundos. Su sintaxis es: `\Pau=number\`	Pausa para \Pau=50\50 ms.
Rst	Inicializa todas las etiquetas a sus valores por defecto. Su sintaxis es: `\Rst\`	Inicializar ahora\Rst\

Animación

Cada agente puede ejecutar cierta cantidad de animaciones, que ejecutan un gesto dado. Para reproducir una animación se utiliza el comando *Play*, como por ejemplo:

```
Request := Character.Play('Greet');
```

De esta forma, se pueden añadir nuevas animaciones a los agentes sin necesidad de extender el API. La Tabla 5.3 lista algunas de las animaciones más frecuentes. Un agente común puede ejecutar alrededor de 100 animaciones, así que deberá consultar la documentación de Microsoft para ver la lista completa de acciones para cada agente.

Las siguientes descripciones se relacionan con el genio. Otros agentes también soportan estas animaciones, pero el movimiento exacto pudiera variar ligeramente.

Tabla 5.3: Animaciones comunes de Microsoft Agent

Animación	Descripción
Announce	Levanta una mano
GestureDown	Gesto hacia abajo
GestureLeft	Gesto hacia la izquierda
GestureRight	Gesto hacia la derecha
GestureUp	Gesto hacia arriba
GetAttention	Atrae la atención moviendo las manos
Greet	Saludo
LookDown	Mira hacia abajo
LookLeft	Mira a su izquierda
LookRight	Mira a su derecha
LookUp	Mira hacia arriba
Processing	El genio gira constantemente
Read	Muestra un pergamino y lee su contenido
Think	Se rasca la cabeza en modo pensativo
Wave	Movimiento ondulado de las manos
Write	Muestra un pergamino en el que escribe

Muchas animaciones también tienen un correspondiente comando "de retorno". Por ejemplo, *Read* tiene una animación correspondiente llamada *ReadReturn*, que provoca que el agente abandone lo que esté leyendo.

La animación *Processing* hace que el genio dé vueltas continuamente. Debemos usar el comando *Stop* para detener la animación *Processing* antes de que sea posible activar otra animación. La moraleja es que hay que consultar la documentación de todos y cada uno de los comandos que utilice, porque algunos de ellos tienen sus "trucos".

Notificaciones de Eventos

Como he mencionado anteriormente, Microsoft Agent notifica a la aplicación cuando ocurren ciertos eventos. Delphi hace muy simple responder a estos eventos. Seleccione el icono correspondiente al agente en el formulario principal del proyecto, y active la página *Events* del Inspector de Objetos. Verá un listado de los posibles eventos. La Tabla 5.4 lista alguno de los más utilizados. La aplicación de ejemplo *AgentDemo* demuestra el uso de algunos de ellos.

Tabla 5.4: Eventos comunes usados por Microsoft Agent

Evento	Descripción
OnBookmark	Se dispara cuando se alcanza una marca en el texto durante la ejecución del comando *Speak*.
OnClick	El usuario ha hecho clic sobre el agente.
OnDblClick	El usuario ha hecho doble clic sobre el agente.
OnDragComplete	El usuario finaliza el arrastre del agente.
OnDragStart	El usuario comienza el arrastre del agente.
OnHide	El agente ha sido ocultado.
OnMove	El agente ha sido movido.
OnRequestComplete	Una petición se ha completado.
OnRequestStart	Una petición se ha comenzado a ejecutar.
OnShow	El agente es mostrado.
OnShutdown	El usuario ha apagado el Microsoft Agent.

Ejemplo de aplicación Microsoft Agent.

Para mostrar las ideas antes expuestas, he creado una aplicación de ejemplo llamada *AgentDemo*, que le permitirá experimentar con voces, jugar con diferentes animaciones y mover al genio alrededor de la pantalla. Además demuestra el uso de peticiones y eventos para notificar las interacciones del usuario con el genio.

Cuando ejecute *AgentDemo*, verá la pantalla que muestra la Figura 5.4:

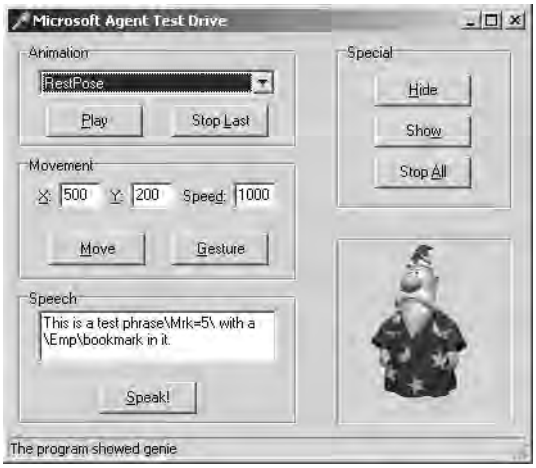


Figura 5.4: Aplicación *AgentDemo*.

El formulario de la aplicación AgentDemo está dividido en cuatro secciones: Animation, Movement, Speech, y Special. También contiene una recuadro en la esquina inferior derecha del formulario en el que se muestra el genio.

La sección *Animation* contiene un cuadro de selección que lista las animaciones que el genio puede ejecutar. Para reproducir una animación, simplemente seleccione la animación en el cuadro y pulse *Play*. Para detener una animación continua, tal como *Processing*, pulse *Stop Last*.

Nota

Observe que este ejemplo está vinculado con el genio. Si desea usarlo para mostrar por ejemplo a Merlín, necesitará modificar el programa para hacer uso del fichero de agente *Merlin.ACS*.

La sección *Movement* nos permite introducir las posiciones *X* e *Y* (en coordenadas relativas de pantalla) para mover el genio a una posición dada, además de configurar la velocidad. Pulse el botón *Move* para mover al genio, o *Gesture* para ordenar al genio apuntar a una localización de la pantalla.

Para hacer que el genio hable, escriba el texto deseado en el control de edición y pulse *Speak*. Puede también utilizar marcas dentro del texto, como muestra la Figura 5.4.

El área final de interés es la sección *Special*. Esta sección nos ofrece la posibilidad de ocultar y mostrar el genio, y podemos además detener todas las conversaciones y animaciones pendientes que estén ejecutándose.

También hemos proporcionado manejadores de eventos básicos para *OnBookmark*, *OnDblClick*, *OnDragComplete*, *OnHide* y *OnShow*. Al examinar estos manejadores de eventos se hará una buena idea de cómo manejar algunas de las notificaciones de eventos más comunes de Microsoft Agent.

Si experimenta con el proyecto durante unos pocos minutos estará mejor informado de las bases de utilización de Microsoft Agent. Los Listados 5.3 y 5.4 contienen el código fuente de la aplicación de ejemplo.

Listado 5.3: *AgentDemo- AgentDemo.dpr*

```
program AgentDemo;
uses
  Forms,
  MainForm in 'MainForm.pas' {Form1};
{$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Listado 5.4: AgentDemo - *MainForm.pas*

```
unit MainForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  ComCtrls, ExtCtrls, StdCtrls, OleCtrls, ActiveX, AgenTObjects_TLB;
const
  WM_SHOWAGENT = WM_USER + 1;
  AGENT = 'genie';
  AGENTPATH = 'C:\WinNT\MSAgent\Chars\genie.acs';
  RequestCompleted = 0;
  RequestFailed = 1;
  RequestPending = 2;
  RequestInterrupted = 3;
  RequestInProgress = 4;
  NeverShown = 0;
  UserHidSpeech = 1;
  UserShowed = 2;
  ProgramHid = 3;
  ProgramShowed = 4;
  OtherProgramHid = 5;
  OtherProgramShowed = 6;
  UserHidMenu = 7;
  LeftButton = 1;
  RightButton = 2;
  MiddleButton = 4;
type
  TForm1 = class(TForm)
    Agent1: TAgent;
    grpAnimation: TGroupBox;
    cbAnimations: TComboBox;
    btnPlay: TButton;
    btnStopLast: TButton;
    grpMovement: TGroupBox;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    XPos: TEdit;
    YPos: TEdit;
    btnMove: TButton;
    btnGesture: TButton;
    Speed: TEdit;
    grpSpeech: TGroupBox;
    Memo1: TMemo;
    btnSpeak: TButton;
    grpSpecial: TGroupBox;
    btnHide: TButton;
    btnShow: TButton;
    btnStopAll: TButton;
    Bevel1: TBevel;
```

```
    StatusBar1: TStatusBar;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormShow(Sender: TObject);
    procedure btnPlayClick(Sender: TObject);
    procedure btnStopLastClick(Sender: TObject);
    procedure btnSpeakClick(Sender: TObject);
    procedure btnMoveClick(Sender: TObject);
    procedure btnGestureClick(Sender: TObject);
    procedure btnHideClick(Sender: TObject);
    procedure btnShowClick(Sender: TObject);
    procedure btnStopAllClick(Sender: TObject);
    procedure Agent1Bookmark(Sender: TObject; BookmarkID: Integer);
    procedure Agent1DblClick(Sender: TObject;
      const CharacterID: WideString; Button, Shift, x, y: Smallint);
    procedure Agent1DragComplete(Sender: TObject;
      const CharacterID: WideString; Button, Shift, x, y: Smallint);
    procedure Agent1HideShow(Sender: TObject; const CharacterID:
WideString;
      Cause: Smallint);
  private
    { Private declarations }
    Character: IAgentCtlCharacterEx;
    FRequest: IAgentCtlRequest;
    procedure WaitFor(Request: IAgentCtlRequest);
    procedure WMShowAgent(var Message: TMessage); message WM_SHOWAGENT;
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.WaitFor(Request: IAgentCtlRequest);
var
  Status: LongInt;
begin
  repeat
    Application.ProcessMessages;
    Status := Request.Get_Status;
  until (Status  RequestPending) and (Status  RequestInProgress);
end;
procedure TForm1.FormCreate(Sender: TObject);
var
  Animations: IAgentCtlAnimationNames;
  Enum: IEnumVariant;
  V: OleVariant;
  Fetched: Cardinal;
begin
  Character := nil;
  FRequest := nil;
  Agent1.Connected := True;
```

```
  Agent1.Characters.Load(AGENT, AGENTPATH);
  Character := Agent1.Characters.Character(AGENT);
  Animations := Character.AnimationNames;
  Enum := Animations.Enum as IEnumVariant;
  while Enum.Next(1, V, Fetched) = S_OK do
    cbAnimations.Items.Add(V);
  cbAnimations.ItemIndex := 0;
end;
procedure TForm1.FormDestroy(Sender: TObject);
var
  Request: IAgentCtlRequest;
begin
  if Character  nil then begin
    Character.StopAll('');
    Request := Character.Hide(False);
    WaitFor(Request);
    Agent1.Characters.Unload(AGENT);
  end;
  Agent1.Connected := False;
end;
procedure TForm1.WMShowAgent(var Message: TMessage);
var
  pt: TPoint;
begin
  pt.x := Bevel1.Left;
  pt.y := Bevel1.Top;
  pt := ClientToScreen(pt);
  Character.MoveTo(pt.x, pt.y, 0);
  Character.Show(False);
  Character.Play('Announce');
  Character.Speak('Hello.  I am an agent.', '');
  Character.Play('RestPose');
end;
procedure TForm1.FormShow(Sender: TObject);
begin
  PostMessage(Handle, WM_SHOWAGENT, 0, 0);
end;
// Animation
procedure TForm1.btnPlayClick(Sender: TObject);
var
  Status: LongInt;
begin
  // If we're already playing something, stop it
  if FRequest  nil then begin
    Status := FRequest.Get_Status;
    if (Status = 2) or (Status = 4) then
      Character.Stop(FRequest);
  end;
  FRequest := Character.Play(cbAnimations.Text);
end;
procedure TForm1.btnStopLastClick(Sender: TObject);
begin
```

```
  if FRequest  nil then
    Character.Stop(FRequest);
end;
// Speech
procedure TForm1.btnSpeakClick(Sender: TObject);
begin
  Character.Speak(Memo1.Text, '');
end;
// Movement
procedure TForm1.btnMoveClick(Sender: TObject);
begin
  Character.MoveTo(StrToInt(XPos.Text), StrToInt(YPos.Text),
    StrToInt(Speed.Text));
end;
procedure TForm1.btnGestureClick(Sender: TObject);
begin
  Character.GestureAt(StrToInt(XPos.Text), StrToInt(YPos.Text));
end;
// Special
procedure TForm1.btnHideClick(Sender: TObject);
begin
  Character.Hide(False);
end;
procedure TForm1.btnShowClick(Sender: TObject);
begin
  Character.Show(False);
end;
procedure TForm1.btnStopAllClick(Sender: TObject);
begin
  Character.StopAll('');
end;
// Event handlers
procedure TForm1.Agent1Bookmark(Sender: TObject; BookmarkID: Integer);
begin
  StatusBar1.SimpleText := 'Triggered bookmark ' + IntToStr(BookmarkID);
end;
procedure TForm1.Agent1DblClick(Sender: TObject;
  const CharacterID: WideString; Button, Shift, x, y: Smallint);
begin
  if Button = LeftButton then
    StatusBar1.SimpleText := 'The user double-clicked on ' + CharacterID;
end;
procedure TForm1.Agent1DragComplete(Sender: TObject;
  const CharacterID: WideString; Button, Shift, x, y: Smallint);
begin
  StatusBar1.SimpleText := Format('The user moved %s to %d, %d',
    [CharacterID, x, y]);
end;
procedure TForm1.Agent1HideShow(Sender: TObject;
 const CharacterID: WideString; Cause: Smallint);
begin
  case Cause of
```

```
    NeverShown:
      StatusBar1.SimpleText := 'Nobody ever showed ' + CharacterID;
    UserHidSpeech:
      StatusBar1.SimpleText := 'The user hid ' + CharacterID
        + ' via spoken command';
    UserShowed:
      StatusBar1.SimpleText := 'The user showed ' + CharacterID;
    ProgramHid:
      StatusBar1.SimpleText := 'The program hid ' + CharacterID;
    ProgramShowed:
      StatusBar1.SimpleText := 'The program showed ' + CharacterID;
    OtherProgramHid:
      StatusBar1.SimpleText := 'Another program hid ' + CharacterID;
    OtherProgramShowed:
      StatusBar1.SimpleText := 'Another program showed ' + CharacterID;
    UserHidMenu:
      StatusBar1.SimpleText := 'The user hid ' + CharacterID + ' via
menu';
    else
      StatusBar1.SimpleText := 'Unknown show/hide status for ' +
CharacterID;
  end;
end;
end.
```

En esta sección hemos mostrado cómo instalar un control ActiveX existente en Delphi, y nos hemos familiarizado con el control ActiveX Microsoft Agent. En la próxima sección mostraremos cómo podemos crear nuestros propios controles ActiveX en Delphi.

Creación de controles ActiveX

En esta sección mostraremos cómo crear un control ActiveX a partir de un componente existente de Delphi. En Delphi, tomaremos un control visual existente (derivado de *TWinControl*) y lo convertiremos en un control ActiveX. Esto significa que podemos escribir y probar un componente VCL completamente en Delphi antes de crear el control ActiveX.

Razones para crear controles ActiveX

Con toda la capacidad de la VCL, ¿por qué necesitaría crear un control ActiveX? Por una buena razón: Ud. desea que su control pueda ser utilizado por todos los desarrolladores, incluso aquellos que no usan Delphi. Por supuesto, el mayor mercado para los controles ActiveX es el mercado de Visual Basic.

Nota

Generación de ActiveX en un paso

Una de las cualidades más resaltadas en las campañas de publicidad lanzadas para Delphi 3 fue cómo éste ofrecía la posibilidad de producir controles ActiveX en un solo paso. En otras palabras, en solo un paso es posible convertir componentes visuales en controles ActiveX. Aunque esto es técnicamente cierto, el control ActiveX resultante no siempre es aceptable. Definitivamente, es posible crear un control ActiveX en un solo paso, pero éste no siempre tiene todas las capacidades del componente VCL a partir del cual fue creado.

Convirtiendo un componente VCL a un Control ActiveX

Convirtamos el componente *TCheckListBox* en un control ActiveX. Para realizar esto, ejecute los siguientes pasos:

1. Del menú principal de Delphi, seleccionemos *File | New*. Se presentará el Depósito de Objetos.
2. En la página ActiveX, seleccione *ActiveX Library* y pulse *OK*.
3. Seleccione de nuevo *File | New*, y el icono *ActiveX Control* de la página *ActiveX* del Depósito de Objetos. Se presentará el Asistente para Controles ActiveX, como se muestra en la Figura 5.5.

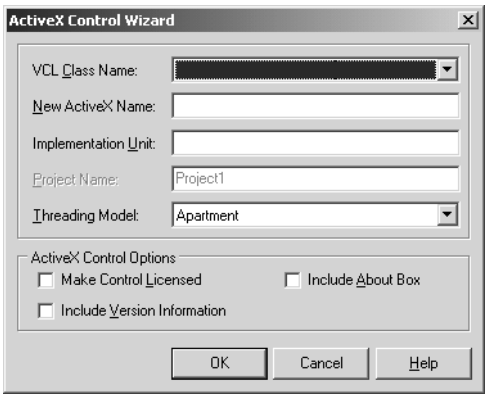


Figura 5.5: El Asistente para Controles ActiveX convierte un componente VCL en un control ActiveX.

4. En el cuadro *VCL Class Name*, seleccione el componente VCL que será convertido a un control ActiveX. Se dará cuenta de que varios de los componentes VCL instalados no están en el listado. En particular, los siguientes tipos de componentes no son incluidos:
   - Componentes que no son derivados de *TWinControl*.
   - Componentes que han sido registrados a través del procedimiento *RegisterNonActiveX*. Entre otros, estos componentes incluyen los

componentes de bases de datos, tales como *TDBEdit*, y componentes que incluyen colecciones, tales como *TListView*.

Cuando seleccione *TCheckListBox* en el cuadro de combinación, el Asistente para Controles ActiveX automáticamente genera valores por defecto para el nombre del control, la unidad de implementación, el nombre del proyecto y el modelo de hilos.

Nota

Se debe utilizar siempre el valor *tmApartment* para el modelo de hilos, porque esto es obligatorio para que el control pueda ser utilizado en Visual Basic. Además, la VCL no soporta ningún otro modelo para controles ActiveX que contengan componentes visuales VCL.

El Asistente para Controles ActiveX contiene además tres casillas de verificación:

- Marque la opción *Make Control Licensed* si desea reforzar el control de licencia en tiempo de diseño para el control ActiveX. Cuando esta opción es seleccionada, Delphi automáticamente genera un fichero de licencia, con extensión .LIC, para utilizarlo con el control ActiveX. Cuando distribuya el control ActiveX resultante a otros programadores para que lo utilicen en tiempo de diseño, deberá suministrarles el fichero de licencia. El control ActiveX verificará la existencia del fichero de licencia en tiempo de diseño. Si el fichero de licencia no existe o es incorrecto, la utilización del control en tiempo de diseño estará prohibida. Cuando distribuya el control ActiveX a sus usuarios finales, no tendrá que distribuir el fichero de licencia. Estos serán capaces de utilizarlo en tiempo de ejecución, pero no serán capaces de desarrollar una aplicación utilizándolo.
- Al menos que tenga una buena razón para no hacerlo, debe siempre seleccionar la opción *Include Version Information*. Esta opción permite añadir información sobre la versión del control ActiveX. Es importante señalar que algunos productos no pueden cargar un control ActiveX que no contenga información de versión. Visual Basic 4 es un ejemplo**.** Cuando seleccione esta opción, asegúrese de rellenar también los campos de información de versión en *Project | Options*.
- Si selecciona la opción *Include About Box*, Delphi generará una ventana de diálogo que podrá utilizar para mostrar información sobre el control ActiveX, su empresa, o cualquier otro dato que desee mostrar a los usuarios del control ActiveX. Normalmente se debe seleccionar esta opción.

Cuando haya rellenado los datos del Asistente para Controles ActiveX, este tendrá la apariencia que se muestra en la Figura 5.6.

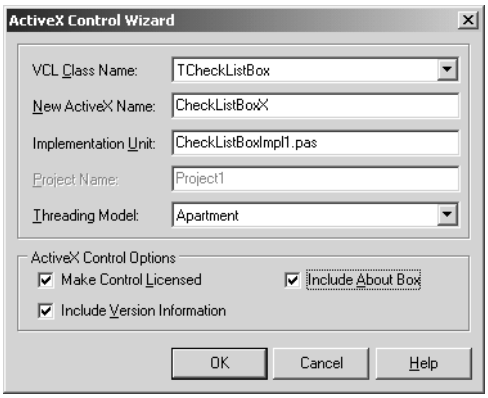


Figura 5.6: Asistente para controles ActiveX.

5. Pulse *OK* para generar el control ActiveX y los ficheros asociados. Los Listados 5.5 al 5.8 muestran el código generado para el control ActiveX.

Listado 5.5: Fichero de proyecto del control ActiveX

```
library Project1;
uses
  ComServ,
  Project1_TLB in 'Project1_TLB.pas',
  CheckListBoxImpl1 in 'CheckListBoxImpl1.pas' {CheckListBoxX: CoClass},
  About1 in 'About1.pas' {CheckListBoxXAbout},
{$E ocx}
exports
  DllGetClassObject,
  DllCanUnloadNow,
  DllRegisterServer,
  DllUnregisterServer;
{$R *.TLB}
{$R *.RES}
begin
end.
```

Listado 5.6: *CheckListBoxImpl1.pas*

```
unit CheckListBoxImpl1;
interface
uses
  Windows, ActiveX, Classes, Controls, Graphics, Menus, Forms, StdCtrls,
  ComServ, StdVCL, AXCtrls, CLBX_TLB, CheckLst;
type
  TCheckListBoxX = class(TActiveXControl, ICheckListBoxX)
  private
    { Private declarations }
    FDelphiControl: TCheckListBox;
    FEvents: ICheckListBoxXEvents;
```

```
    procedure ClickCheckEvent(Sender: TObject);
    procedure ClickEvent(Sender: TObject);
    procedure DblClickEvent(Sender: TObject);
    procedure KeyPressEvent(Sender: TObject; var Key: Char);
  protected
    { Protected declarations }
    procedure DefinePropertyPages
      (DefinePropertyPage: TDefinePropertyPage); override;
    procedure EventSinkChanged(const EventSink: IUnknown); override;
    procedure InitializeControl; override;
    function Get_AllowGrayed: WordBool; safecall;
    function Get_BorderStyle: TxBorderStyle; safecall;
    function Get_Color: OLE_COLOR; safecall;
    function Get_Columns: Integer; safecall;
    function Get_Ctl3D: WordBool; safecall;
    function Get_Cursor: Smallint; safecall;
    function Get_DoubleBuffered: WordBool; safecall;
    function Get_DragCursor: Smallint; safecall;
    function Get_DragMode: TxDragMode; safecall;
    function Get_Enabled: WordBool; safecall;
    function Get_Flat: WordBool; safecall;
    function Get_Font: IFontDisp; safecall;
    function Get_ImeMode: TxImeMode; safecall;
    function Get_ImeName: WideString; safecall;
    function Get_IntegralHeight: WordBool; safecall;
    function Get_ItemHeight: Integer; safecall;
    function Get_ItemIndex: Integer; safecall;
    function Get_Items: IStrings; safecall;
    function Get_ParentColor: WordBool; safecall;
    function Get_ParentCtl3D: WordBool; safecall;
    function Get_SelCount: Integer; safecall;
    function Get_Sorted: WordBool; safecall;
    function Get_Style: TxListBoxStyle; safecall;
    function Get_TabWidth: Integer; safecall;
    function Get_TopIndex: Integer; safecall;
    function Get_Visible: WordBool; safecall;
    function Get_VisibleDockClientCount: Integer; safecall;
    procedure _Set_Font(const Value: IFontDisp); safecall;
    procedure AboutBox; safecall;
    procedure Set_AllowGrayed(Value: WordBool); safecall;
    procedure Set_BorderStyle(Value: TxBorderStyle); safecall;
    procedure Set_Color(Value: OLE_COLOR); safecall;
    procedure Set_Columns(Value: Integer); safecall;
    procedure Set_Ctl3D(Value: WordBool); safecall;
    procedure Set_Cursor(Value: Smallint); safecall;
    procedure Set_DoubleBuffered(Value: WordBool); safecall;
    procedure Set_DragCursor(Value: Smallint); safecall;
    procedure Set_DragMode(Value: TxDragMode); safecall;
    procedure Set_Enabled(Value: WordBool); safecall;
    procedure Set_Flat(Value: WordBool); safecall;
    procedure Set_Font(var Value: IFontDisp); safecall;
    procedure Set_ImeMode(Value: TxImeMode); safecall;
```

```
    procedure Set_ImeName(const Value: WideString); safecall;
    procedure Set_IntegralHeight(Value: WordBool); safecall;
    procedure Set_ItemHeight(Value: Integer); safecall;
    procedure Set_ItemIndex(Value: Integer); safecall;
    procedure Set_Items(const Value: IStrings); safecall;
    procedure Set_ParentColor(Value: WordBool); safecall;
    procedure Set_ParentCtl3D(Value: WordBool); safecall;
    procedure Set_Sorted(Value: WordBool); safecall;
    procedure Set_Style(Value: TxListBoxStyle); safecall;
    procedure Set_TabWidth(Value: Integer); safecall;
    procedure Set_TopIndex(Value: Integer); safecall;
    procedure Set_Visible(Value: WordBool); safecall;
  end;
implementation
uses ComObj, AboutForm, GeneralPage;
{ TCheckListBoxX }

procedure TCheckListBoxX.DefinePropertyPages
  (DefinePropertyPage: TDefinePropertyPage);
begin
  {TODO: Define property pages here.  Property pages are defined by
calling
    DefinePropertyPage with the class id of the page.  For example,
      DefinePropertyPage(Class_CheckListBoxXPage); }
  DefinePropertyPage(Class_ppgGeneral);
  DefinePropertyPage(Class_DStringPropPage);
end;
procedure TCheckListBoxX.EventSinkChanged(const EventSink: IUnknown);
begin
  FEvents := EventSink as ICheckListBoxXEvents;
end;
procedure TCheckListBoxX.InitializeControl;
begin
  FDelphiControl := Control as TCheckListBox;
  FDelphiControl.OnClick := ClickEvent;
  FDelphiControl.OnClickCheck := ClickCheckEvent;
  FDelphiControl.OnDblClick := DblClickEvent;
  FDelphiControl.OnKeyPress := KeyPressEvent;
end;
function TCheckListBoxX.Get_AllowGrayed: WordBool;
begin
  Result := FDelphiControl.AllowGrayed;
end;
function TCheckListBoxX.Get_BorderStyle: TxBorderStyle;
begin
  Result := Ord(FDelphiControl.BorderStyle);
end;
function TCheckListBoxX.Get_Color: OLE_COLOR;
begin
  Result := OLE_COLOR(FDelphiControl.Color);
end;
function TCheckListBoxX.Get_Columns: Integer;
```

```
begin
  Result := FDelphiControl.Columns;
end;
function TCheckListBoxX.Get_Ctl3D: WordBool;
begin
  Result := FDelphiControl.Ctl3D;
end;
function TCheckListBoxX.Get_Cursor: Smallint;
begin
  Result := Smallint(FDelphiControl.Cursor);
end;
function TCheckListBoxX.Get_DoubleBuffered: WordBool;
begin
  Result := FDelphiControl.DoubleBuffered;
end;
function TCheckListBoxX.Get_DragCursor: Smallint;
begin
  Result := Smallint(FDelphiControl.DragCursor);
end;
function TCheckListBoxX.Get_DragMode: TxDragMode;
begin
  Result := Ord(FDelphiControl.DragMode);
end;
function TCheckListBoxX.Get_Enabled: WordBool;
begin
  Result := FDelphiControl.Enabled;
end;
function TCheckListBoxX.Get_Flat: WordBool;
begin
  Result := FDelphiControl.Flat;
end;
function TCheckListBoxX.Get_Font: IFontDisp;
begin
  GetOleFont(FDelphiControl.Font, Result);
end;
function TCheckListBoxX.Get_ImeMode: TxImeMode;
begin
  Result := Ord(FDelphiControl.ImeMode);
end;
function TCheckListBoxX.Get_ImeName: WideString;
begin
  Result := WideString(FDelphiControl.ImeName);
end;
function TCheckListBoxX.Get_IntegralHeight: WordBool;
begin
  Result := FDelphiControl.IntegralHeight;
end;
function TCheckListBoxX.Get_ItemHeight: Integer;
begin
  Result := FDelphiControl.ItemHeight;
end;
function TCheckListBoxX.Get_ItemIndex: Integer;
```

```
begin
  Result := FDelphiControl.ItemIndex;
end;
function TCheckListBoxX.Get_Items: IStrings;
begin
  GetOleStrings(FDelphiControl.Items, Result);
end;
function TCheckListBoxX.Get_ParentColor: WordBool;
begin
  Result := FDelphiControl.ParentColor;
end;
function TCheckListBoxX.Get_ParentCtl3D: WordBool;
begin
  Result := FDelphiControl.ParentCtl3D;
end;
function TCheckListBoxX.Get_SelCount: Integer;
begin
  Result := FDelphiControl.SelCount;
end;
function TCheckListBoxX.Get_Sorted: WordBool;
begin
  Result := FDelphiControl.Sorted;
end;
function TCheckListBoxX.Get_Style: TxListBoxStyle;
begin
  Result := Ord(FDelphiControl.Style);
end;
function TCheckListBoxX.Get_TabWidth: Integer;
begin
  Result := FDelphiControl.TabWidth;
end;
function TCheckListBoxX.Get_TopIndex: Integer;
begin
  Result := FDelphiControl.TopIndex;
end;
function TCheckListBoxX.Get_Visible: WordBool;
begin
  Result := FDelphiControl.Visible;
end;
function TCheckListBoxX.Get_VisibleDockClientCount: Integer;
begin
  Result := FDelphiControl.VisibleDockClientCount;
end;
procedure TCheckListBoxX._Set_Font(const Value: IFontDisp);
begin
  SetOleFont(FDelphiControl.Font, Value);
end;
procedure TCheckListBoxX.AboutBox;
begin
  ShowCheckListBoxXAbout;
end;
procedure TCheckListBoxX.ClickCheckEvent(Sender: TObject);
```

```
begin
  if FEvents  nil then FEvents.OnClickCheck;
end;
procedure TCheckListBoxX.ClickEvent(Sender: TObject);
begin
  if FEvents  nil then FEvents.OnClick;
end;
procedure TCheckListBoxX.DblClickEvent(Sender: TObject);
begin
  if FEvents  nil then FEvents.OnDblClick;
end;
procedure TCheckListBoxX.KeyPressEvent(Sender: TObject; var Key: Char);
var
  TempKey: Smallint;
begin
  TempKey := Smallint(Key);
  if FEvents  nil then FEvents.OnKeyPress(TempKey);
  Key := Char(TempKey);
end;
procedure TCheckListBoxX.Set_AllowGrayed(Value: WordBool);
begin
  FDelphiControl.AllowGrayed := Value;
end;
procedure TCheckListBoxX.Set_BorderStyle(Value: TxBorderStyle);
begin
  FDelphiControl.BorderStyle := TBorderStyle(Value);
end;
procedure TCheckListBoxX.Set_Color(Value: OLE_COLOR);
begin
  FDelphiControl.Color := TColor(Value);
end;
procedure TCheckListBoxX.Set_Columns(Value: Integer);
begin
  FDelphiControl.Columns := Value;
end;
procedure TCheckListBoxX.Set_Ctl3D(Value: WordBool);
begin
  FDelphiControl.Ctl3D := Value;
end;
procedure TCheckListBoxX.Set_Cursor(Value: Smallint);
begin
  FDelphiControl.Cursor := TCursor(Value);
end;
procedure TCheckListBoxX.Set_DoubleBuffered(Value: WordBool);
begin
  FDelphiControl.DoubleBuffered := Value;
end;
procedure TCheckListBoxX.Set_DragCursor(Value: Smallint);
begin
  FDelphiControl.DragCursor := TCursor(Value);
end;
procedure TCheckListBoxX.Set_DragMode(Value: TxDragMode);
```

```
begin
  FDelphiControl.DragMode := TDragMode(Value);
end;
procedure TCheckListBoxX.Set_Enabled(Value: WordBool);
begin
  FDelphiControl.Enabled := Value;
end;
procedure TCheckListBoxX.Set_Flat(Value: WordBool);
begin
  FDelphiControl.Flat := Value;
end;
procedure TCheckListBoxX.Set_Font(var Value: IFontDisp);
begin
  SetOleFont(FDelphiControl.Font, Value);
end;
procedure TCheckListBoxX.Set_ImeMode(Value: TxImeMode);
begin
  FDelphiControl.ImeMode := TImeMode(Value);
end;
procedure TCheckListBoxX.Set_ImeName(const Value: WideString);
begin
  FDelphiControl.ImeName := TImeName(Value);
end;
procedure TCheckListBoxX.Set_IntegralHeight(Value: WordBool);
begin
  FDelphiControl.IntegralHeight := Value;
end;
procedure TCheckListBoxX.Set_ItemHeight(Value: Integer);
begin
  FDelphiControl.ItemHeight := Value;
end;
procedure TCheckListBoxX.Set_ItemIndex(Value: Integer);
begin
  FDelphiControl.ItemIndex := Value;
end;
procedure TCheckListBoxX.Set_Items(const Value: IStrings);
begin
  SetOleStrings(FDelphiControl.Items, Value);
end;
procedure TCheckListBoxX.Set_ParentColor(Value: WordBool);
begin
  FDelphiControl.ParentColor := Value;
end;
procedure TCheckListBoxX.Set_ParentCtl3D(Value: WordBool);
begin
  FDelphiControl.ParentCtl3D := Value;
end;
procedure TCheckListBoxX.Set_Sorted(Value: WordBool);
begin
  FDelphiControl.Sorted := Value;
end;
procedure TCheckListBoxX.Set_Style(Value: TxListBoxStyle);
```

```
begin
  FDelphiControl.Style := TListBoxStyle(Value);
end;
procedure TCheckListBoxX.Set_TabWidth(Value: Integer);
begin
  FDelphiControl.TabWidth := Value;
end;
procedure TCheckListBoxX.Set_TopIndex(Value: Integer);
begin
  FDelphiControl.TopIndex := Value;
end;
procedure TCheckListBoxX.Set_Visible(Value: WordBool);
begin
  FDelphiControl.Visible := Value;
end;
initialization
  TActiveXControlFactory.Create(
    ComServer,
    TCheckListBoxX,
    TCheckListBox,
    Class_CheckListBoxX,
    1,
    '{3774F179-5D73-11D3-B872-0040F67455FE}',
    0,
    tmApartment);
end.
```

Listado 5.7: *CLBX_TLB.pas*

```
unit CLBX_TLB;
// ************************************************************************ //
// WARNING
// □□□-
// The types declared in this file were generated from data read from a
// Type Library. If this type library is explicitly or indirectly (via
// another type library referring to this type library) re-imported, or
// the 'Refresh' command of the Type Library Editor activated while
// editing the Type Library, the contents of this file will be
// regenerated and all manual modifications will be lost.
// ************************************************************************ //
// PASTLWTR : $Revision:   1.88  $
// File generated on 5/15/2001 3:23:59 PM from Type Library described
below.
// ************************************************************************ //
// NOTE:
// Items guarded by $IFDEF_LIVE_SERVER_AT_DESIGN_TIME are used by
// properties which return objects that may need to be explicitly
// created via a function call prior to any access via the property.
// These items have been disabled in order to prevent accidental use
// from within the object inspector. You may enable them by defining
// LIVE_SERVER_AT_DESIGN_TIME or by selectively removing them from the
// $IFDEF blocks. However, such items must still be programmatically
```

```
// created via a method of the appropriate CoClass before they can be used.
// ************************************************************************ //
// Type Lib: C:\Book\samples\Chap05\ActiveX\CLBX.tlb (1)
// IID\LCID: {3774F16D-5D73-11D3-B872-0040F67455FE}\0
// Helpfile:
// DepndLst:
//   (1) v2.0 stdole, (C:\WINNT\System32\StdOle2.tlb)
//   (2) v4.0 StdVCL, (C:\WINNT\System32\STDVCL40.DLL)
// ************************************************************************ //
{$TYPEDADDRESS OFF} // Unit must be compiled without type-checked
pointers.
interface
uses Windows, ActiveX, Classes, Graphics, OleServer, OleCtrls, StdVCL;
// *********************************************************************//
// GUIDS declared in the TypeLibrary. Following prefixes are used:
//   Type Libraries     : LIBID_xxxx
//   CoClasses          : CLASS_xxxx
//   DISPInterfaces     : DIID_xxxx
//   Non-DISP interfaces: IID_xxxx
// *********************************************************************//
const
  // TypeLibrary Major and minor versions
  CLBXMajorVersion = 1;
  CLBXMinorVersion = 0;
  LIBID_CLBX: TGUID = '{3774F16D-5D73-11D3-B872-0040F67455FE}';
  IID_ICheckListBoxX: TGUID = '{3774F16E-5D73-11D3-B872-0040F67455FE}';

DIID_ICheckListBoxXEvents:TGUID='{3774F170-5D73-11D3-B872-0040F67455FE}';
  CLASS_CheckListBoxX: TGUID = '{3774F172-5D73-11D3-B872-0040F67455FE}';
// *********************************************************************//
// Declaration of Enumerations defined in Type Library
// *********************************************************************//
// Constants for enum TxBorderStyle
type
  TxBorderStyle = TOleEnum;
const
  bsNone = $00000000;
  bsSingle = $00000001;
// Constants for enum TxDragMode
type
  TxDragMode = TOleEnum;
const
  dmManual = $00000000;
  dmAutomatic = $00000001;
// Constants for enum TxImeMode
type
  TxImeMode = TOleEnum;
const
  imDisable = $00000000;
  imClose = $00000001;
  imOpen = $00000002;
  imDontCare = $00000003;
```

```
    imSAlpha = $00000004;
    imAlpha = $00000005;
    imHira = $00000006;
    imSKata = $00000007;
    imKata = $00000008;
    imChinese = $00000009;
    imSHanguel = $0000000A;
    imHanguel = $0000000B;
// Constants for enum TxListBoxStyle
type
    TxListBoxStyle = TOleEnum;
const
    lbStandard = $00000000;
    lbOwnerDrawFixed = $00000001;
    lbOwnerDrawVariable = $00000002;
// Constants for enum TxMouseButton
type
    TxMouseButton = TOleEnum;
const
    mbLeft = $00000000;
    mbRight = $00000001;
    mbMiddle = $00000002;
type
// *********************************************************************//
// Forward declaration of types defined in TypeLibrary
// *********************************************************************//
    ICheckListBoxX = interface;
    ICheckListBoxXDisp = dispinterface;
    ICheckListBoxXEvents = dispinterface;
// *********************************************************************//
// Declaration of CoClasses defined in Type Library
// (NOTE: Here we map each CoClass to its Default Interface)
// *********************************************************************//
    CheckListBoxX = ICheckListBoxX;
// *********************************************************************//
// Declaration of structures, unions and aliases.
// *********************************************************************//
    PPUserType1 = ^IFontDisp; {*}
// *********************************************************************//
// Interface: ICheckListBoxX
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {3774F16E-5D73-11D3-B872-0040F67455FE}
// *********************************************************************//
    ICheckListBoxX = interface (IDispatch)
      ['{3774F16E-5D73-11D3-B872-0040F67455FE}']
      function  Get_AllowGrayed: WordBool; safecall;
      procedure Set_AllowGrayed(Value: WordBool); safecall;
      function  Get_BorderStyle: TxBorderStyle; safecall;
      procedure Set_BorderStyle(Value: TxBorderStyle); safecall;
      function  Get_Color: OLE_COLOR; safecall;
      procedure Set_Color(Value: OLE_COLOR); safecall;
      function  Get_Columns: Integer; safecall;
```

```
    procedure Set_Columns(Value: Integer); safecall;
    function  Get_Ctl3D: WordBool; safecall;
    procedure Set_Ctl3D(Value: WordBool); safecall;
    function  Get_DragCursor: Smallint; safecall;
    procedure Set_DragCursor(Value: Smallint); safecall;
    function  Get_DragMode: TxDragMode; safecall;
    procedure Set_DragMode(Value: TxDragMode); safecall;
    function  Get_Enabled: WordBool; safecall;
    procedure Set_Enabled(Value: WordBool); safecall;
    function  Get_Flat: WordBool; safecall;
    procedure Set_Flat(Value: WordBool); safecall;
    function  Get_Font: IFontDisp; safecall;
    procedure _Set_Font(const Value: IFontDisp); safecall;
    procedure Set_Font(var Value: IFontDisp); safecall;
    function  Get_ImeMode: TxImeMode; safecall;
    procedure Set_ImeMode(Value: TxImeMode); safecall;
    function  Get_ImeName: WideString; safecall;
    procedure Set_ImeName(const Value: WideString); safecall;
    function  Get_IntegralHeight: WordBool; safecall;
    procedure Set_IntegralHeight(Value: WordBool); safecall;
    function  Get_ItemHeight: Integer; safecall;
    procedure Set_ItemHeight(Value: Integer); safecall;
    function  Get_Items: IStrings; safecall;
    procedure Set_Items(const Value: IStrings); safecall;
    function  Get_ParentColor: WordBool; safecall;
    procedure Set_ParentColor(Value: WordBool); safecall;
    function  Get_ParentCtl3D: WordBool; safecall;
    procedure Set_ParentCtl3D(Value: WordBool); safecall;
    function  Get_Sorted: WordBool; safecall;
    procedure Set_Sorted(Value: WordBool); safecall;
    function  Get_Style: TxListBoxStyle; safecall;
    procedure Set_Style(Value: TxListBoxStyle); safecall;
    function  Get_TabWidth: Integer; safecall;
    procedure Set_TabWidth(Value: Integer); safecall;
    function  Get_Visible: WordBool; safecall;
    procedure Set_Visible(Value: WordBool); safecall;
    function  Get_ItemIndex: Integer; safecall;
    procedure Set_ItemIndex(Value: Integer); safecall;
    function  Get_SelCount: Integer; safecall;
    function  Get_TopIndex: Integer; safecall;
    procedure Set_TopIndex(Value: Integer); safecall;
    function  Get_DoubleBuffered: WordBool; safecall;
    procedure Set_DoubleBuffered(Value: WordBool); safecall;
    function  Get_VisibleDockClientCount: Integer; safecall;
    function  Get_Cursor: Smallint; safecall;
    procedure Set_Cursor(Value: Smallint); safecall;
    procedure AboutBox; safecall;
    property AllowGrayed: WordBool read Get_AllowGrayed write
Set_AllowGrayed;
    property BorderStyle: TxBorderStyle
      read Get_BorderStyle write Set_BorderStyle;
    property Color: OLE_COLOR read Get_Color write Set_Color;
```

```
    property Columns: Integer read Get_Columns write Set_Columns;
    property Ctl3D: WordBool read Get_Ctl3D write Set_Ctl3D;
    property DragCursor: Smallint read Get_DragCursor write
Set_DragCursor;
    property DragMode: TxDragMode read Get_DragMode write Set_DragMode;
    property Enabled: WordBool read Get_Enabled write Set_Enabled;
    property Flat: WordBool read Get_Flat write Set_Flat;
    property Font: IFontDisp read Get_Font write _Set_Font;
    property ImeMode: TxImeMode read Get_ImeMode write Set_ImeMode;
    property ImeName: WideString read Get_ImeName write Set_ImeName;
    property IntegralHeight: WordBool
      read Get_IntegralHeight write Set_IntegralHeight;
    property ItemHeight: Integer read Get_ItemHeight write Set_ItemHeight;
    property Items: IStrings read Get_Items write Set_Items;
    property ParentColor: WordBool read Get_ParentColor write
Set_ParentColor;
    property ParentCtl3D: WordBool read Get_ParentCtl3D write
Set_ParentCtl3D;
    property Sorted: WordBool read Get_Sorted write Set_Sorted;
    property Style: TxListBoxStyle read Get_Style write Set_Style;
    property TabWidth: Integer read Get_TabWidth write Set_TabWidth;
    property Visible: WordBool read Get_Visible write Set_Visible;
    property ItemIndex: Integer read Get_ItemIndex write Set_ItemIndex;
    property SelCount: Integer read Get_SelCount;
    property TopIndex: Integer read Get_TopIndex write Set_TopIndex;
    property DoubleBuffered: WordBool
      read Get_DoubleBuffered write Set_DoubleBuffered;
    property VisibleDockClientCount: Integer read
Get_VisibleDockClientCount;
    property Cursor: Smallint read Get_Cursor write Set_Cursor;
  end;
// ******************************************************************//
// DispIntf:  ICheckListBoxXDisp
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {3774F16E-5D73-11D3-B872-0040F67455FE}
// ******************************************************************//
  ICheckListBoxXDisp = dispinterface
    ['{3774F16E-5D73-11D3-B872-0040F67455FE}']
    property AllowGrayed: WordBool dispid 1;
    property BorderStyle: TxBorderStyle dispid 2;
    property Color: OLE_COLOR dispid -501;
    property Columns: Integer dispid 3;
    property Ctl3D: WordBool dispid 4;
    property DragCursor: Smallint dispid 5;
    property DragMode: TxDragMode dispid 6;
    property Enabled: WordBool dispid -514;
    property Flat: WordBool dispid 7;
    property Font: IFontDisp dispid -512;
    property ImeMode: TxImeMode dispid 8;
    property ImeName: WideString dispid 9;
    property IntegralHeight: WordBool dispid 10;
    property ItemHeight: Integer dispid 11;
    property Items: IStrings dispid 12;
```

```
    property ParentColor: WordBool dispid 13;
    property ParentCtl3D: WordBool dispid 14;
    property Sorted: WordBool dispid 15;
    property Style: TxListBoxStyle dispid 16;
    property TabWidth: Integer dispid 17;
    property Visible: WordBool dispid 18;
    property ItemIndex: Integer dispid 21;
    property SelCount: Integer readonly dispid 22;
    property TopIndex: Integer dispid 23;
    property DoubleBuffered: WordBool dispid 24;
    property VisibleDockClientCount: Integer readonly dispid 25;
    property Cursor: Smallint dispid 34;
    procedure AboutBox; dispid -552;
  end;
// *********************************************************************//
// DispIntf:  ICheckListBoxXEvents
// Flags:     (4096) Dispatchable
// GUID:      {3774F170-5D73-11D3-B872-0040F67455FE}
// *********************************************************************//
  ICheckListBoxXEvents = dispinterface
    ['{3774F170-5D73-11D3-B872-0040F67455FE}']
    procedure OnClickCheck; dispid 1;
    procedure OnClick; dispid 2;
    procedure OnDblClick; dispid 4;
    procedure OnKeyPress(var Key: Smallint); dispid 11;
  end;
// *********************************************************************//
// OLE Control Proxy class declaration
// Control Name     : TCheckListBoxX
// Help string      : CheckListBoxX Control
// Default Interface: ICheckListBoxX
// Def. Intf. DISP? : No
// Event   Interface: ICheckListBoxXEvents
// TypeFlags        : (38) CanCreate Licensed Control
// *********************************************************************//
  TCheckListBoxXOnKeyPress =
    procedure(Sender: TObject; var Key: Smallint) of object;
  TCheckListBoxX = class(TOleControl)
  private
    FOnClickCheck: TNotifyEvent;
    FOnClick: TNotifyEvent;
    FOnDblClick: TNotifyEvent;
    FOnKeyPress: TCheckListBoxXOnKeyPress;
    FIntf: ICheckListBoxX;
    function  GetControlInterface: ICheckListBoxX;
  protected
    procedure CreateControl;
    procedure InitControlData; override;
    function  Get_Items: IStrings;
    procedure Set_Items(const Value: IStrings);
  public
    procedure AboutBox;
```

```
    property  ControlInterface: ICheckListBoxX read GetControlInterface;
    property  DefaultInterface: ICheckListBoxX read GetControlInterface;
    property ItemIndex: Integer index 21
      read GetIntegerProp write SetIntegerProp;
    property SelCount: Integer index 22 read GetIntegerProp;
    property TopIndex: Integer index 23
      read GetIntegerProp write SetIntegerProp;
    property DoubleBuffered: WordBool index 24
      read GetWordBoolProp write SetWordBoolProp;
    property VisibleDockClientCount: Integer index 25 read GetIntegerProp;
  published
    property  ParentFont;
    property  TabStop;
    property  Align;
    property  ParentShowHint;
    property  PopupMenu;
    property  ShowHint;
    property  TabOrder;
    property  OnDragDrop;
    property  OnDragOver;
    property  OnEndDrag;
    property  OnEnter;
    property  OnExit;
    property  OnStartDrag;
    property AllowGrayed: WordBool index 1
      read GetWordBoolProp write SetWordBoolProp stored False;
    property BorderStyle: TOleEnum index 2
      read GetTOleEnumProp write SetTOleEnumProp stored False;
    property Color: TColor index -501
      read GetTColorProp write SetTColorProp stored False;
    property Columns: Integer index 3
      read GetIntegerProp write SetIntegerProp stored False;
    property Ctl3D: WordBool index 4
      read GetWordBoolProp write SetWordBoolProp stored False;
    property DragCursor: Smallint index 5
      read GetSmallintProp write SetSmallintProp stored False;
    property DragMode: TOleEnum index 6
      read GetTOleEnumProp write SetTOleEnumProp stored False;
    property Enabled: WordBool index -514
      read GetWordBoolProp write SetWordBoolProp stored False;
    property Flat: WordBool index 7
      read GetWordBoolProp write SetWordBoolProp stored False;
    property Font: TFont index -512
      read GetTFontProp write SetTFontProp stored False;
    property ImeMode: TOleEnum index 8
      read GetTOleEnumProp write SetTOleEnumProp stored False;
    property ImeName: WideString index 9
      read GetWideStringProp write SetWideStringProp stored False;
    property IntegralHeight: WordBool index 10
      read GetWordBoolProp write SetWordBoolProp stored False;
    property ItemHeight: Integer index 11
      read GetIntegerProp write SetIntegerProp stored False;
```

```
    property Items: IStrings read Get_Items write Set_Items stored False;
    property ParentColor: WordBool index 13
      read GetWordBoolProp write SetWordBoolProp stored False;
    property ParentCtl3D: WordBool index 14
      read GetWordBoolProp write SetWordBoolProp stored False;
    property Sorted: WordBool index 15
      read GetWordBoolProp write SetWordBoolProp stored False;
    property Style: TOleEnum index 16
      read GetTOleEnumProp write SetTOleEnumProp stored False;
    property TabWidth: Integer index 17
      read GetIntegerProp write SetIntegerProp stored False;
    property Visible: WordBool index 18
      read GetWordBoolProp write SetWordBoolProp stored False;
    property Cursor: Smallint index 34
      read GetSmallintProp write SetSmallintProp stored False;
    property OnClickCheck: TNotifyEvent
      read FOnClickCheck write FOnClickCheck;
    property OnClick: TNotifyEvent read FOnClick write FOnClick;
    property OnDblClick: TNotifyEvent read FOnDblClick write FOnDblClick;
    property OnKeyPress: TCheckListBoxXOnKeyPress
      read FOnKeyPress write FOnKeyPress;
  end;
procedure Register;
implementation
uses ComObj;
procedure TCheckListBoxX.InitControlData;
const
  CEventDispIDs: array [0..3] of DWORD = (
    $00000001, $00000002, $00000004, $0000000B);
  CLicenseKey: array [0..38] of Word = (
      $007B, $0033, $0037, $0037, $0034, $0046, $0031,
      $0037, $0039, $002D, $0035
    , $0044, $0037, $0033, $002D, $0031, $0031, $0044, $0033, $002D, $0042
    , $0038, $0037, $0032, $002D, $0030, $0030, $0034, $0030, $0046, $0036
    , $0037, $0034, $0035, $0035, $0046, $0045, $007D, $0000);
  CTFontIDs: array [0..0] of DWORD = (
    $FFFFFE00);
  CControlData: TControlData2 = (
    ClassID: '{3774F172-5D73-11D3-B872-0040F67455FE}';
    EventIID: '{3774F170-5D73-11D3-B872-0040F67455FE}';
    EventCount: 4;
    EventDispIDs: @CEventDispIDs;
    LicenseKey: @CLicenseKey;
    Flags: $0000000D;
    Version: 401;
    FontCount: 1;
    FontIDs: @CTFontIDs);
begin
  ControlData := @CControlData;
  TControlData2(CControlData).FirstEventOfs :=
    Cardinal(@@FOnClickCheck) - Cardinal(Self);
end;
```

```
procedure TCheckListBoxX.CreateControl;
  procedure DoCreate;
  begin
    FIntf := IUnknown(OleObject) as ICheckListBoxX;
  end;
begin
  if FIntf = nil then DoCreate;
end;
function TCheckListBoxX.GetControlInterface: ICheckListBoxX;
begin
  CreateControl;
  Result := FIntf;
end;
function  TCheckListBoxX.Get_Items: IStrings;
begin
  Result := DefaultInterface.Get_Items;
end;
procedure TCheckListBoxX.Set_Items(const Value: IStrings);
begin
  DefaultInterface.Set_Items(Value);
end;
procedure TCheckListBoxX.AboutBox;
begin
  DefaultInterface.AboutBox;
end;
procedure Register;
begin
  RegisterComponents('ActiveX',[TCheckListBoxX]);
end;
end.
```

Listado 5.8: *About1.pas*

```
unit About1;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  StdCtrls, ExtCtrls, Buttons;
type
  TFriendFormXAbout = class(TForm)
    CtlImage: TSpeedButton;
    NameLbl: TLabel;
    OkBtn: TButton;
    CopyrightLbl: TLabel;
    DescLbl: TLabel;
  end;
procedure ShowFriendFormXAbout;
implementation
{$R *.DFM}
procedure ShowFriendFormXAbout;
begin
```

```
  with TFriendFormXAbout.Create(nil) do
    try
      ShowModal;
    finally
      Free;
    end;
end;
end.
```

Seleccione *Project | Build Project1* desde el menú principal de Delphi para compilar el control ActiveX.

Registrando un control ActiveX

Una vez que un control ActiveX es construido, necesitamos registrarlo en Windows para poder utilizarlo. Desde el menú principal de Delphi, seleccione *Run | Register ActiveX Server*. Delphi registrará el control ActiveX y mostrará un mensaje indicando que el control fue registrado de forma satisfactoria.

Probando el control ActiveX en Visual Basic

Ahora que el control ActiveX ha sido creado y registrado en Windows, puede probarlo para estar seguro de que trabaja adecuadamente. Deberá probarlo con una copia de Visual Basic, ya que como indicamos anteriormente, los usuarios de Visual Basic son de hecho el mayor mercado para los controles ActiveX.

Nota

Para asegurarse de que el control funciona correctamente debería probarlo sobre una variedad de contenedores ActiveX. Probar nuestro control con una variedad de aplicaciones contenedoras como *Explorer*, *Word*, etc. garantizará la robustez de nuestro control.

Añadiendo una página de propiedades a un control ActiveX

Si prueba el control ActiveX en Visual Basic, se dará cuenta de que no todas las propiedades del control VCL se muestran en Visual Basic. En particular, parece no haber forma de modificar la propiedad *Items* del componente. Sin embargo, al añadir una página de propiedades al control ActiveX, será fácil acceder a la propiedad *Items* en tiempo de diseño.

Añadiendo una página de propiedades predefinidas en Delphi

Delphi proporciona cuatro páginas de propiedades predefinidas que pueden utilizarse para editar las propiedades de un control ActiveX. Las páginas de propiedades predefinidas se listan en la Tabla 5.5.

Tabla 5.5: Páginas de propiedades predefinidas de un control ActiveX

CLSID	**Descripción**
Class_DColorPropPage	Usada para editar propiedades *TColor*.
Class_DFontPropPage	Usada para editar propiedades *TFont*.
Class_DPicturePropPage	Usada para editar propiedades *TPicture*.
Class_DStringPropPage	Usada para editar propiedades *TStrings*.

Para añadir una o más páginas de propiedades a nuestro control ActiveX, debemos añadir algún código al método *DefinePropertyPages* del control:

```
procedure TCheckListBoxX.DefinePropertyPages
  (DefinePropertyPage: TDefinePropertyPage);
begin
 {TODO: Definir las p□ginas de propiedades aqu□. Las p□ginas de
propiedades
  se definen llamando a DefinePropertyPage con el ClassID de la p□gina.}
  DefinePropertyPage(Class_CheckListBoxXPage);
  DefinePropertyPage(Class_DStringPropPage);
end;
```

Añadiendo nuestra propia página de propiedades

Es sencillo crear nuestra propia página de propiedades para editar datos del control. Seleccione *File | New* del menú principal de Delphi. En el Depósito de Objetos, seleccione la página *ActiveX* y haga doble clic sobre el elemento *Property Page*. Delphi genera un formulario estándar que podrá utilizar para configurar cualquiera de las propiedades que desee de su control ActiveX. Delphi también crea un único CLSID (que es simplemente un GUID), que puede ser utilizado para añadir la página de propiedades al control ActiveX, nuevamente añadiendo código al método *DefinePropertyPages* del control.

```
procedure TCheckListBoxX.DefinePropertyPages
  (DefinePropertyPage: TDefinePropertyPage);
begin
  {TODO: Definir las p□ginas de propiedades aqu□. Las p□ginas de
propiedades
   se definen llamando a DefinePropertyPage con el ClassID de la p□gina.}
  DefinePropertyPage(Class_ppgGeneral);
  DefinePropertyPage(Class_DStringPropPage);
end;
```

Distribuyendo controles ActiveX

Después de que el control ActiveX ha sido construido y probado, es hora de empaquetarlo para su distribución. Como mínimo, necesitará distribuir el fichero *.OCX* que contiene al control ActiveX. Durante la instalación del control, deberá asegurarse de que éste se registra en Windows. Cualquier software de instalación

que se aprecie proporciona una opción para registrar el control durante la instalación. También deberá redistribuir (y registrar) *STD-VCL32.DLL*, que contiene las páginas de propiedades predefinidas listadas en la Tabla 5.5, al igual que tipos predefinidos como *IStrings*.

Si ha activado la opción de control de licencia al crear el control ActiveX, necesitará también distribuir el fichero *.LIC* para proporcionar a los desarrolladores una licencia para la utilización del control.

Ahora que conoce cómo crear un control ActiveX a partir de un componente VCL, le mostraré cómo crear un control ActiveX basado en un formulario de Delphi.

ActiveForms

Un *ActiveForm* no es más que un control ActiveX basado en un formulario de Delphi. En lugar de convertir un componente individual en un control ActiveX, podemos crear un ActiveForm tan fácilmente como se crea un formulario normal de Delphi.

Creando un ActiveForm

La Figura 5.7 muestra el ActiveForm que vamos a crear. Como puede ver, éste permite a los usuarios introducir su nombre y su dirección de correo electrónico. El procedimiento para crear un ActiveForm complejo será el mismo que para uno sencillo, así que no es necesario detenerse en la construcción de un ejemplo complicado. Es suficiente decir que podemos colocar cualquier componente de Delphi sobre un ActiveForm, incluyendo los componentes de acceso a datos.

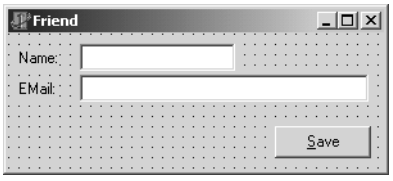


Figura 5.7: Ejemplo de ActiveForm.

Para crear un ActiveForm en un servidor COM interno al proceso, ejecute los siguientes pasos:

1. Primero, cree una nueva biblioteca ActiveX seleccionando *File* | *New* del menú principal de Delphi.
2. En la página *ActiveX* del Depósito de Objetos, haga doble clic sobre el icono *ActiveX Library*.

3. Seleccione nuevamente *File | New* del menú principal de Delphi, y haga doble clic sobre el icono *ActiveForm* en la página *ActiveX* del Depósito de Objetos.
4. El Asistente para ActiveForms aparecerá. Rellene los campos del asistente tal como se muestra en la Figura 5.8 y pulse *OK*.

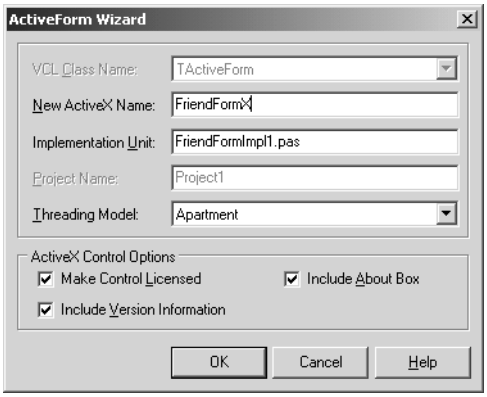


Figura 5.8: Asistente de ActiveForms de Delphi.

5. Pulse sobre el icono '*Save Project*' en la barra de herramientas de Delphi. Guarde la unidad *About1* como *AbouForm.pas*, *FriendFormImpl1* como *FriendFormImpl.pas*, y *Project1* como *FriendX.dpr*.

   Notará que Delphi crea un formulario en blanco para Ud. Utilice el familiar diseñador de formularios de Delphi para diseñar un ActiveForm exactamente de la misma forma que si diseñara un formulario estándar de Delphi. Para este ejemplo, he añadido dos controles de edición (para el nombre y el e-mail), y un botón para guardar. Además he añadido código al evento *OnClick* del botón, con el objetivo de guardar el nombre y la dirección de correo electrónico a un fichero texto.

Los Listados 5.9 al 5.12 muestran el código fuente de esta aplicación.

Listado 5.9: FriendX - *FriendX.dpr*

```
library FriendX;
uses
  ComServ,
  FriendX_TLB in 'FriendX_TLB.pas',
  FriendFormImpl in 'FriendFormImpl.pas' {FriendFormX: TActiveForm}
                                         {FriendFormX: CoClass},
  AboutForm in 'AboutForm.pas' {FriendFormXAbout};
{$E ocx}
exports
```

```
  DllGetClassObject,
  DllCanUnloadNow,
  DllRegisterServer,
  DllUnregisterServer;
{$R *.TLB}
{$R *.RES}
begin
end.
```

Listado 5.10: FriendX - *AboutForm.pas*

```
unit AboutForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  StdCtrls, ExtCtrls, Buttons;
type
  TFriendFormXAbout = class(TForm)
    CtlImage: TSpeedButton;
    NameLbl: TLabel;
    OkBtn: TButton;
    CopyrightLbl: TLabel;
    DescLbl: TLabel;
  end;
procedure ShowFriendFormXAbout;
implementation
{$R *.DFM}
procedure ShowFriendFormXAbout;
begin
  with TFriendFormXAbout.Create(nil) do
    try
      ShowModal;
    finally
      Free;
    end;
end;
end.
```

Listado 5.11: FriendX - *FriendX_TLB.pas*

```
unit FriendX_TLB;
// ************************************************************************ //
// WARNING
// □□□-
// The types declared in this file were generated from data read from a
// Type Library. If this type library is explicitly or indirectly (via
// another type library referring to this type library) re-imported, or
// the 'Refresh' command of the Type Library Editor activated while
// edditing the Type Library, the contents of this file will be
// regenerated and all manual modifications will be lost.
// ************************************************************************ //
// PASTLWTR : $Revision:   1.88  $
```

```
// File generated on 5/15/2001 1:48:21 PM from Type Library described
below.
//
************************************************************************//
// NOTE:
// Items guarded by $IFDEF_LIVE_SERVER_AT_DESIGN_TIME are used by
// properties which return objects that may need to be explicitly
// created via a function call prior to any access via the property.
// These items have been disabled in order to prevent accidental use from
// within the object inspector. You may enable them by defining
// LIVE_SERVER_AT_DESIGN_TIME or by selectively removing them from the
// $IFDEF blocks. However, such items must still be programmatically
// created via a method of the appropriate CoClass before they can be
// used.
// ************************************************************************ //
// Type Lib: C:\Book\samples\Chap05\ActiveForm\FriendX.tlb (1)
// IID\LCID: {9EC8AF31-5D6D-11D3-B872-0040F67455FE}\0
// Helpfile:
// DepndLst:
//   (1) v2.0 stdole, (C:\WINNT\System32\StdOle2.tlb)
//   (2) v4.0 StdVCL, (C:\WINNT\System32\STDVCL40.DLL)
// ************************************************************************ //
{$TYPEDADDRESS OFF} // Unit must be compiled without type-checked
pointers.
interface
uses Windows, ActiveX, Classes, Graphics, OleServer, OleCtrls, StdVCL;
// ************************************************************************//
// GUIDS declared in the TypeLibrary. Following prefixes are used:
//   Type Libraries     : LIBID_xxxx
//   CoClasses          : CLASS_xxxx
//   DISPInterfaces     : DIID_xxxx
//   Non-DISP interfaces: IID_xxxx
// ************************************************************************//
const
  // TypeLibrary Major and minor versions
  FriendXMajorVersion = 1;
  FriendXMinorVersion = 0;
  LIBID_FriendX: TGUID = '{9EC8AF31-5D6D-11D3-B872-0040F67455FE}';
  IID_IFriendFormX: TGUID = '{9EC8AF32-5D6D-11D3-B872-0040F67455FE}';
  DIID_IFriendFormXEvents: TGUID =
'{9EC8AF34-5D6D-11D3-B872-0040F67455FE}';
  CLASS_FriendFormX: TGUID = '{9EC8AF36-5D6D-11D3-B872-0040F67455FE}';
// ************************************************************************//
// Declaration of Enumerations defined in Type Library
// ************************************************************************//
// Constants for enum TxActiveFormBorderStyle
type
  TxActiveFormBorderStyle = TOleEnum;
const
  afbNone = $00000000;
  afbSingle = $00000001;
  afbSunken = $00000002;
  afbRaised = $00000003;
```

```
// Constants for enum TxPrintScale
type
  TxPrintScale = TOleEnum;
const
  poNone = $00000000;
  poProportional = $00000001;
  poPrintToFit = $00000002;
// Constants for enum TxMouseButton
type
  TxMouseButton = TOleEnum;
const
  mbLeft = $00000000;
  mbRight = $00000001;
  mbMiddle = $00000002;
type
// *********************************************************************//
// Forward declaration of types defined in TypeLibrary
// *********************************************************************//
  IFriendFormX = interface;
  IFriendFormXDisp = dispinterface;
  IFriendFormXEvents = dispinterface;
// *********************************************************************//
// Declaration of CoClasses defined in Type Library
// (NOTE: Here we map each CoClass to its Default Interface)
// *********************************************************************//
  FriendFormX = IFriendFormX;
// *********************************************************************//
// Declaration of structures, unions and aliases.
// *********************************************************************//
  PPUserType1 = ^IFontDisp; {*}
// *********************************************************************//
// Interface: IFriendFormX
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {9EC8AF32-5D6D-11D3-B872-0040F67455FE}
// *********************************************************************//
  IFriendFormX = interface (IDispatch)
    ['{9EC8AF32-5D6D-11D3-B872-0040F67455FE}']
    function  Get_Visible: WordBool; safecall;
    procedure Set_Visible(Value: WordBool); safecall;
    function  Get_AutoScroll: WordBool; safecall;
    procedure Set_AutoScroll(Value: WordBool); safecall;
    function  Get_AutoSize: WordBool; safecall;
    procedure Set_AutoSize(Value: WordBool); safecall;
    function  Get_AxBorderStyle: TxActiveFormBorderStyle; safecall;
    procedure Set_AxBorderStyle(Value: TxActiveFormBorderStyle); safecall;
    function  Get_Caption: WideString; safecall;
    procedure Set_Caption(const Value: WideString); safecall;
    function  Get_Color: OLE_COLOR; safecall;
    procedure Set_Color(Value: OLE_COLOR); safecall;
    function  Get_Font: IFontDisp; safecall;
    procedure _Set_Font(const Value: IFontDisp); safecall;
    procedure Set_Font(var Value: IFontDisp); safecall;
```

```
    function  Get_KeyPreview: WordBool; safecall;
    procedure Set_KeyPreview(Value: WordBool); safecall;
    function  Get_PixelsPerInch: Integer; safecall;
    procedure Set_PixelsPerInch(Value: Integer); safecall;
    function  Get_PrintScale: TxPrintScale; safecall;
    procedure Set_PrintScale(Value: TxPrintScale); safecall;
    function  Get_Scaled: WordBool; safecall;
    procedure Set_Scaled(Value: WordBool); safecall;
    function  Get_Active: WordBool; safecall;
    function  Get_DropTarget: WordBool; safecall;
    procedure Set_DropTarget(Value: WordBool); safecall;
    function  Get_HelpFile: WideString; safecall;
    procedure Set_HelpFile(const Value: WideString); safecall;
    function  Get_DoubleBuffered: WordBool; safecall;
    procedure Set_DoubleBuffered(Value: WordBool); safecall;
    function  Get_VisibleDockClientCount: Integer; safecall;
    function  Get_Enabled: WordBool; safecall;
    procedure Set_Enabled(Value: WordBool); safecall;
    function  Get_Cursor: Smallint; safecall;
    procedure Set_Cursor(Value: Smallint); safecall;
    procedure AboutBox; safecall;
    property Visible: WordBool read Get_Visible write Set_Visible;
    property AutoScroll: WordBool read Get_AutoScroll write
Set_AutoScroll;
    property AutoSize: WordBool read Get_AutoSize write Set_AutoSize;
    property AxBorderStyle: TxActiveFormBorderStyle
      read Get_AxBorderStyle write Set_AxBorderStyle;
    property Caption: WideString read Get_Caption write Set_Caption;
    property Color: OLE_COLOR read Get_Color write Set_Color;
    property Font: IFontDisp read Get_Font write _Set_Font;
    property KeyPreview: WordBool read Get_KeyPreview write
Set_KeyPreview;
    property PixelsPerInch: Integer
      read Get_PixelsPerInch write Set_PixelsPerInch;
    property PrintScale: TxPrintScale
      read Get_PrintScale write Set_PrintScale;
    property Scaled: WordBool read Get_Scaled write Set_Scaled;
    property Active: WordBool read Get_Active;
    property DropTarget: WordBool read Get_DropTarget write
Set_DropTarget;
    property HelpFile: WideString read Get_HelpFile write Set_HelpFile;
    property DoubleBuffered: WordBool
      read Get_DoubleBuffered write Set_DoubleBuffered;
    property VisibleDockClientCount: Integer read
Get_VisibleDockClientCount;
    property Enabled: WordBool read Get_Enabled write Set_Enabled;
    property Cursor: Smallint read Get_Cursor write Set_Cursor;
  end;
// *********************************************************************//
// DispIntf:  IFriendFormXDisp
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {9EC8AF32-5D6D-11D3-B872-004OF67455FE}
// *********************************************************************//
```

```
  IFriendFormXDisp = dispinterface
    ['{9EC8AF32-5D6D-11D3-B872-004OF67455FE}']
    property Visible: WordBool dispid 1;
    property AutoScroll: WordBool dispid 2;
    property AutoSize: WordBool dispid 3;
    property AxBorderStyle: TxActiveFormBorderStyle dispid 4;
    property Caption: WideString dispid -518;
    property Color: OLE_COLOR dispid -501;
    property Font: IFontDisp dispid -512;
    property KeyPreview: WordBool dispid 5;
    property PixelsPerInch: Integer dispid 6;
    property PrintScale: TxPrintScale dispid 7;
    property Scaled: WordBool dispid 8;
    property Active: WordBool readonly dispid 9;
    property DropTarget: WordBool dispid 10;
    property HelpFile: WideString dispid 11;
    property DoubleBuffered: WordBool dispid 12;
    property VisibleDockClientCount: Integer readonly dispid 13;
    property Enabled: WordBool dispid -514;
    property Cursor: Smallint dispid 14;
    procedure AboutBox; dispid -552;
  end;
// ******************************************************************//
// DispIntf:  IFriendFormXEvents
// Flags:     (0)
// GUID:      {9EC8AF34-5D6D-11D3-B872-004OF67455FE}
// ******************************************************************//
  IFriendFormXEvents = dispinterface
    ['{9EC8AF34-5D6D-11D3-B872-004OF67455FE}']
    procedure OnActivate; dispid 1;
    procedure OnClick; dispid 2;
    procedure OnCreate; dispid 3;
    procedure OnDblClick; dispid 5;
    procedure OnDestroy; dispid 6;
    procedure OnDeactivate; dispid 7;
    procedure OnKeyPress(var Key: Smallint); dispid 11;
    procedure OnPaint; dispid 16;
  end;
// ******************************************************************//
// OLE Control Proxy class declaration
// Control Name     : TFriendFormX
// Help string      : FriendFormX Control
// Default Interface: IFriendFormX
// Def. Intf. DISP? : No
// Event   Interface: IFriendFormXEvents
// TypeFlags        : (38) CanCreate Licensed Control
// ******************************************************************//
  TFriendFormXOnKeyPress =
    procedure(Sender: TObject; var Key: Smallint) of object;
  TFriendFormX = class(TOleControl)
  private
    FOnActivate: TNotifyEvent;
```

```
    FOnClick: TNotifyEvent;
    FOnCreate: TNotifyEvent;
    FOnDblClick: TNotifyEvent;
    FOnDestroy: TNotifyEvent;
    FOnDeactivate: TNotifyEvent;
    FOnKeyPress: TFriendFormXOnKeyPress;
    FOnPaint: TNotifyEvent;
    FIntf: IFriendFormX;
    function  GetControlInterface: IFriendFormX;
  protected
    procedure CreateControl;
    procedure InitControlData; override;
  public
    procedure AboutBox;
    property  ControlInterface: IFriendFormX read GetControlInterface;
    property  DefaultInterface: IFriendFormX read GetControlInterface;
    property Visible: WordBool index 1
      read GetWordBoolProp write SetWordBoolProp;
    property Active: WordBool index 9 read GetWordBoolProp;
    property DropTarget: WordBool index 10
      read GetWordBoolProp write SetWordBoolProp;
    property HelpFile: WideString index 11
      read GetWideStringProp write SetWideStringProp;
    property DoubleBuffered: WordBool index 12
      read GetWordBoolProp write SetWordBoolProp;
    property VisibleDockClientCount: Integer index 13 read GetIntegerProp;
    property Enabled: WordBool index -514
      read GetWordBoolProp write SetWordBoolProp;
  published
    property AutoScroll: WordBool index 2
      read GetWordBoolProp write SetWordBoolProp stored False;
    property AutoSize: WordBool index 3
      read GetWordBoolProp write SetWordBoolProp stored False;
    property AxBorderStyle: TOleEnum index 4
      read GetTOleEnumProp write SetTOleEnumProp stored False;
    property Caption: WideString index -518
      read GetWideStringProp write SetWideStringProp stored False;
    property Color: TColor index -501
      read GetTColorProp write SetTColorProp stored False;
    property Font: TFont index -512
      read GetTFontProp write SetTFontProp stored False;
    property KeyPreview: WordBool index 5
      read GetWordBoolProp write SetWordBoolProp stored False;
    property PixelsPerInch: Integer index 6
      read GetIntegerProp write SetIntegerProp stored False;
    property PrintScale: TOleEnum index 7
      read GetTOleEnumProp write SetTOleEnumProp stored False;
    property Scaled: WordBool index 8
      read GetWordBoolProp write SetWordBoolProp stored False;
    property Cursor: Smallint index 14
      read GetSmallintProp write SetSmallintProp stored False;
    property OnActivate: TNotifyEvent read FOnActivate write FOnActivate;
```

```
    property OnClick: TNotifyEvent read FOnClick write FOnClick;
    property OnCreate: TNotifyEvent read FOnCreate write FOnCreate;
    property OnDblClick: TNotifyEvent read FOnDblClick write FOnDblClick;
    property OnDestroy: TNotifyEvent read FOnDestroy write FOnDestroy;
    property OnDeactivate: TNotifyEvent
      read FOnDeactivate write FOnDeactivate;
    property OnKeyPress: TFriendFormXOnKeyPress
      read FOnKeyPress write FOnKeyPress;
    property OnPaint: TNotifyEvent read FOnPaint write FOnPaint;
  end;
procedure Register;
implementation
uses ComObj;
procedure TFriendFormX.InitControlData;
const
  CEventDispIDs: array [0..7] of DWORD = (
    $00000001, $00000002, $00000003, $00000005, $00000006, $00000007,
    $0000000B, $00000010);
  CTFontIDs: array [0..0] of DWORD = (
    $FFFFFE00);
  CControlData: TControlData2 = (
    ClassID: '{9EC8AF36-5D6D-11D3-B872-0040F67455FE}';
    EventIID: '{9EC8AF34-5D6D-11D3-B872-0040F67455FE}';
    EventCount: 8;
    EventDispIDs: @CEventDispIDs;
    LicenseKey: nil (*HR:$80040154*);
    Flags: $0000001D;
    Version: 401;
    FontCount: 1;
    FontIDs: @CTFontIDs);
begin
  ControlData := @CControlData;
  TControlData2(CControlData).FirstEventOfs :=
    Cardinal(@@FOnActivate) - Cardinal(Self);
end;
procedure TFriendFormX.CreateControl;
  procedure DoCreate;
  begin
    FIntf := IUnknown(OleObject) as IFriendFormX;
  end;
begin
  if FIntf = nil then DoCreate;
end;
function TFriendFormX.GetControlInterface: IFriendFormX;
begin
  CreateControl;
  Result := FIntf;
end;
procedure TFriendFormX.AboutBox;
begin
  DefaultInterface.AboutBox;
end;
```

```
procedure Register;
begin
  RegisterComponents('ActiveX',[TFriendFormX]);
end;
end.
```

Listado 5.12: FriendX - *FriendFormImpl.pas*

```
unit FriendFormImpl;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  ActiveX, AxCtrls, FriendX_TLB, StdVcl, StdCtrls;
type
  TFriendFormX = class(TActiveForm, IFriendFormX)
    Label1: TLabel;
    ecName: TEdit;
    ecEMail: TEdit;
    Label2: TLabel;
    btnSave: TButton;
    procedure btnSaveClick(Sender: TObject);
  private
    { Private declarations }
    FEvents: IFriendFormXEvents;
    procedure ActivateEvent(Sender: TObject);
    procedure ClickEvent(Sender: TObject);
    procedure CreateEvent(Sender: TObject);
    procedure DblClickEvent(Sender: TObject);
    procedure DeactivateEvent(Sender: TObject);
    procedure DestroyEvent(Sender: TObject);
    procedure KeyPressEvent(Sender: TObject; var Key: Char);
    procedure PaintEvent(Sender: TObject);
  protected
    { Protected declarations }
    procedure DefinePropertyPages
     (DefinePropertyPage: TDefinePropertyPage); override;
    procedure EventSinkChanged(const EventSink: IUnknown); override;
    function Get_Active: WordBool; safecall;
    function Get_AutoScroll: WordBool; safecall;
    function Get_AutoSize: WordBool; safecall;
    function Get_AxBorderStyle: TxActiveFormBorderStyle; safecall;
    function Get_Caption: WideString; safecall;
    function Get_Color: OLE_COLOR; safecall;
    function Get_Cursor: Smallint; safecall;
    function Get_DoubleBuffered: WordBool; safecall;
    function Get_DropTarget: WordBool; safecall;
    function Get_Enabled: WordBool; safecall;
    function Get_Font: IFontDisp; safecall;
    function Get_HelpFile: WideString; safecall;
    function Get_KeyPreview: WordBool; safecall;
    function Get_PixelsPerInch: Integer; safecall;
```

```
    function Get_PrintScale: TxPrintScale; safecall;
    function Get_Scaled: WordBool; safecall;
    function Get_Visible: WordBool; safecall;
    function Get_VisibleDockClientCount: Integer; safecall;
    procedure _Set_Font(const Value: IFontDisp); safecall;
    procedure AboutBox; safecall;
    procedure Set_AutoScroll(Value: WordBool); safecall;
    procedure Set_AutoSize(Value: WordBool); safecall;
    procedure Set_AxBorderStyle(Value: TxActiveFormBorderStyle); safecall;
    procedure Set_Caption(const Value: WideString); safecall;
    procedure Set_Color(Value: OLE_COLOR); safecall;
    procedure Set_Cursor(Value: Smallint); safecall;
    procedure Set_DoubleBuffered(Value: WordBool); safecall;
    procedure Set_DropTarget(Value: WordBool); safecall;
    procedure Set_Enabled(Value: WordBool); safecall;
    procedure Set_Font(var Value: IFontDisp); safecall;
    procedure Set_HelpFile(const Value: WideString); safecall;
    procedure Set_KeyPreview(Value: WordBool); safecall;
    procedure Set_PixelsPerInch(Value: Integer); safecall;
    procedure Set_PrintScale(Value: TxPrintScale); safecall;
    procedure Set_Scaled(Value: WordBool); safecall;
    procedure Set_Visible(Value: WordBool); safecall;
  public
    { Public declarations }
    procedure Initialize; override;
  end;
implementation
uses ComObj, ComServ, AboutForm;
{$R *.DFM}
{ TFriendFormX }
procedure TFriendFormX.DefinePropertyPages
  (DefinePropertyPage: TDefinePropertyPage);
begin
  { Define property pages here. Property pages are defined by calling
    DefinePropertyPage with the class id of the page. For example,
      DefinePropertyPage(Class_FriendFormXPage); }
end;
procedure TFriendFormX.EventSinkChanged(const EventSink: IUnknown);
begin
  FEvents := EventSink as IFriendFormXEvents;
end;
procedure TFriendFormX.Initialize;
begin
  inherited Initialize;
  OnActivate := ActivateEvent;
  OnClick := ClickEvent;
  OnCreate := CreateEvent;
  OnDblClick := DblClickEvent;
  OnDeactivate := DeactivateEvent;
  OnDestroy := DestroyEvent;
  OnKeyPress := KeyPressEvent;
  OnPaint := PaintEvent;
```

```
end;
function TFriendFormX.Get_Active: WordBool;
begin
  Result := Active;
end;
function TFriendFormX.Get_AutoScroll: WordBool;
begin
  Result := AutoScroll;
end;
function TFriendFormX.Get_AutoSize: WordBool;
begin
  Result := AutoSize;
end;
function TFriendFormX.Get_AxBorderStyle: TxActiveFormBorderStyle;
begin
  Result := Ord(AxBorderStyle);
end;
function TFriendFormX.Get_Caption: WideString;
begin
  Result := WideString(Caption);
end;
function TFriendFormX.Get_Color: OLE_COLOR;
begin
  Result := OLE_COLOR(Color);
end;
function TFriendFormX.Get_Cursor: Smallint;
begin
  Result := Smallint(Cursor);
end;
function TFriendFormX.Get_DoubleBuffered: WordBool;
begin
  Result := DoubleBuffered;
end;
function TFriendFormX.Get_DropTarget: WordBool;
begin
  Result := DropTarget;
end;
function TFriendFormX.Get_Enabled: WordBool;
begin
  Result := Enabled;
end;
function TFriendFormX.Get_Font: IFontDisp;
begin
  GetOleFont(Font, Result);
end;
function TFriendFormX.Get_HelpFile: WideString;
begin
  Result := WideString(HelpFile);
end;
function TFriendFormX.Get_KeyPreview: WordBool;
begin
  Result := KeyPreview;
```

```
end;
function TFriendFormX.Get_PixelsPerInch: Integer;
begin
  Result := PixelsPerInch;
end;
function TFriendFormX.Get_PrintScale: TxPrintScale;
begin
  Result := Ord(PrintScale);
end;
function TFriendFormX.Get_Scaled: WordBool;
begin
  Result := Scaled;
end;
function TFriendFormX.Get_Visible: WordBool;
begin
  Result := Visible;
end;
function TFriendFormX.Get_VisibleDockClientCount: Integer;
begin
  Result := VisibleDockClientCount;
end;
procedure TFriendFormX._Set_Font(const Value: IFontDisp);
begin
  SetOleFont(Font, Value);
end;
procedure TFriendFormX.AboutBox;
begin
  ShowFriendFormXAbout;
end;
procedure TFriendFormX.ActivateEvent(Sender: TObject);
begin
  if FEvents  nil then FEvents.OnActivate;
end;
procedure TFriendFormX.ClickEvent(Sender: TObject);
begin
  if FEvents  nil then FEvents.OnClick;
end;
procedure TFriendFormX.CreateEvent(Sender: TObject);
begin
  if FEvents  nil then FEvents.OnCreate;
end;
procedure TFriendFormX.DblClickEvent(Sender: TObject);
begin
  if FEvents  nil then FEvents.OnDblClick;
end;
procedure TFriendFormX.DeactivateEvent(Sender: TObject);
begin
  if FEvents  nil then FEvents.OnDeactivate;
end;
procedure TFriendFormX.DestroyEvent(Sender: TObject);
begin
  if FEvents  nil then FEvents.OnDestroy;
```

```
end;
procedure TFriendFormX.KeyPressEvent(Sender: TObject; var Key: Char);
var
  TempKey: Smallint;
begin
  TempKey := Smallint(Key);
  if FEvents  nil then FEvents.OnKeyPress(TempKey);
  Key := Char(TempKey);
end;
procedure TFriendFormX.PaintEvent(Sender: TObject);
begin
  if FEvents  nil then FEvents.OnPaint;
end;
procedure TFriendFormX.Set_AutoScroll(Value: WordBool);
begin
  AutoScroll := Value;
end;
procedure TFriendFormX.Set_AutoSize(Value: WordBool);
begin
  AutoSize := Value;
end;
procedure TFriendFormX.Set_AxBorderStyle(Value: TxActiveFormBorderStyle);
begin
  AxBorderStyle := TActiveFormBorderStyle(Value);
end;
procedure TFriendFormX.Set_Caption(const Value: WideString);
begin
  Caption := TCaption(Value);
end;
procedure TFriendFormX.Set_Color(Value: OLE_COLOR);
begin
  Color := TColor(Value);
end;
procedure TFriendFormX.Set_Cursor(Value: Smallint);
begin
  Cursor := TCursor(Value);
end;
procedure TFriendFormX.Set_DoubleBuffered(Value: WordBool);
begin
  DoubleBuffered := Value;
end;
procedure TFriendFormX.Set_DropTarget(Value: WordBool);
begin
  DropTarget := Value;
end;
procedure TFriendFormX.Set_Enabled(Value: WordBool);
begin
  Enabled := Value;
end;
procedure TFriendFormX.Set_Font(var Value: IFontDisp);
begin
  SetOleFont(Font, Value);
```

```
end;
procedure TFriendFormX.Set_HelpFile(const Value: WideString);
begin
  HelpFile := string(Value);
end;
procedure TFriendFormX.Set_KeyPreview(Value: WordBool);
begin
  KeyPreview := Value;
end;
procedure TFriendFormX.Set_PixelsPerInch(Value: Integer);
begin
  PixelsPerInch := Value;
end;
procedure TFriendFormX.Set_PrintScale(Value: TxPrintScale);
begin
  PrintScale := TPrintScale(Value);
end;
procedure TFriendFormX.Set_Scaled(Value: WordBool);
begin
  Scaled := Value;
end;
procedure TFriendFormX.Set_Visible(Value: WordBool);
begin
  Visible := Value;
end;
procedure TFriendFormX.btnSaveClick(Sender: TObject);
var
  F: TextFile;
begin
  AssignFile(F, 'C:\FRIENDS.TXT');
  try
    Append(F);
  except
    Rewrite(F);
  end;
  Writeln(F, ecName.Text, ': ', ecEMail.Text);
  CloseFile(F);
end;
initialization
  TActiveFormFactory.Create(
    ComServer,
    TActiveFormControl,
    TFriendFormX,
    Class_FriendFormX,
    1,
    '{9EC8AF3C-5D6D-11D3-B872-0040F67455FE}',
    OLEMISC_SIMPLEFRAME or OLEMISC_ACTSLIKELABEL,
    tmApartment);
end.
```

Probando un ActiveForm

Ahora que ya tiene un ActiveForm operativo, deberá probarlo. Afortunadamente, Delphi nos es de gran ayuda en ese aspecto. Para generar una prueba de un ActiveForm, seleccione del menú principal de Delphi *Project | Web Deployment Options*. Delphi muestra una ventana de diálogo que puede utilizar para especificar dónde será instalado el fichero .OCX del ActiveForm, al igual que el directorio de prueba donde se creará el fichero HTML que podrá utilizar para probar el ActiveForm.

Complete el cuadro de diálogo *Web Deployment Options* según se muestra en la Figura 5.9 y pulse *OK*.

Después que haya establecido las opciones de instalación, todo lo que resta por hacer es generar los ficheros de prueba. Seleccione *Project | Web Deploy* del menú principal de Delphi. Se generará el fichero .OCX del proyecto, conjuntamente con el fichero HTML de prueba. Arranque entonces su navegador de Internet, y visualice el fichero HTML generado por Delphi. Para este ejemplo, he colocado los ficheros de prueba en el directorio *C:\TEST*.

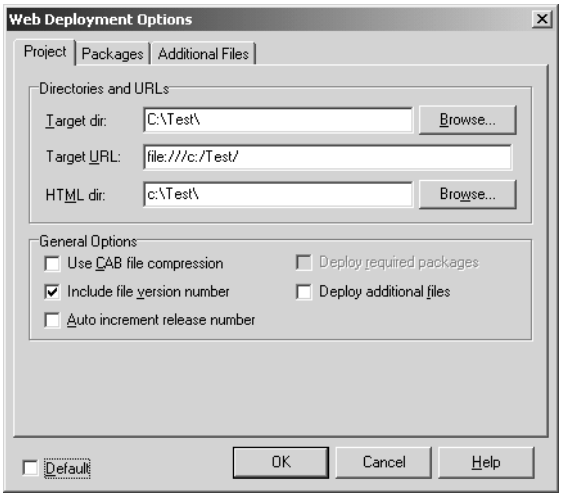


Figura 5.9: Opciones de implantación web.

La Figura 5.10 muestra Internet Explorer con el fichero de prueba HTML cargado.

Introduzca su nombre y dirección de e-mail, y pulse el botón *Save*. Obtendrá un nuevo fichero llamado *FRIENDS.TXT* en el directorio raíz de C:, con la información que haya introducido.

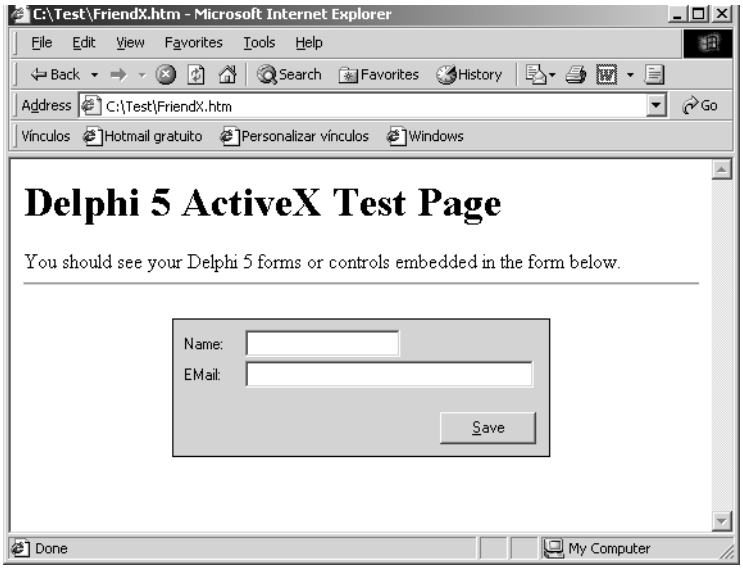


Figura 5.10: Probando el ActiveForm.

Resumen

En este capítulo hemos mostrado cómo aprovechar las ventajas de los controles ActiveX en nuestras aplicaciones Delphi. Como programador de Delphi, normalmente optará por un componente VCL, si existe, pero existen actualmente numerosos controles ActiveX que no tienen su contrapartida VCL.

Seguidamente, hemos mostramos cómo es posible convertir un componente VCL existente en un control ActiveX para que sea utilizado por otros usuarios. Si es Ud. un desarrollador de componentes, esto le abrirá un mercado adicional.

Hemos mostramos cómo se puede implementar manualmente la funcionalidad que Delphi no proporciona automáticamente cuando creamos un control ActiveX a partir de un componente VCL. También ha aprendido a añadir páginas de propiedades a un control ActiveX para ayudar a los usuarios de los mismos.

Finalmente, hemos examinado los ActiveForms, y mostrado cómo podemos crear un control ActiveX a partir de un formulario de Delphi, para posteriormente mostrar éste en Visual Basic o en Internet Explorer, por ejemplo.

En el próximo capítulo echaremos un vistazo a DCOM, que le permitirá implantar servidores y clientes COM en ordenadores conectados a una red.

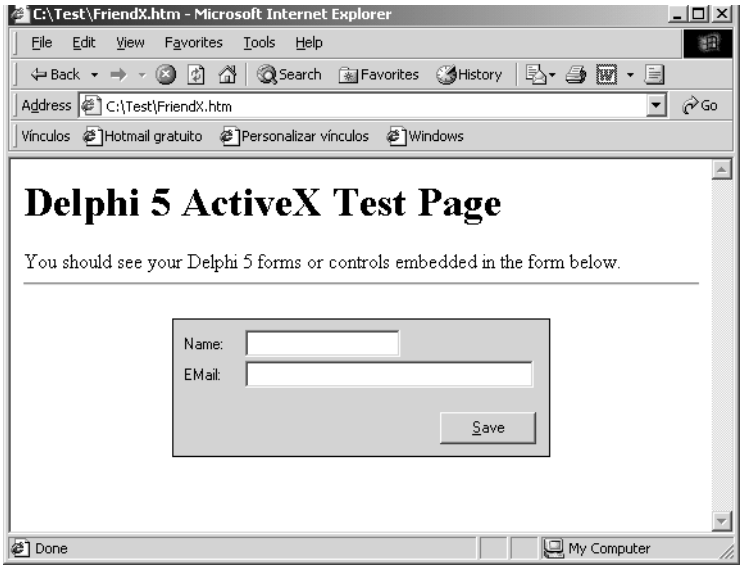


Figura 5.10: Probando el ActiveForm.

Resumen

En este capítulo hemos mostrado cómo aprovechar las ventajas de los controles ActiveX en nuestras aplicaciones Delphi. Como programador de Delphi, normalmente optará por un componente VCL, si existe, pero existen actualmente numerosos controles ActiveX que no tienen su contrapartida VCL.

Seguidamente, hemos mostramos cómo es posible convertir un componente VCL existente en un control ActiveX para que sea utilizado por otros usuarios. Si es Ud. un desarrollador de componentes, esto le abrirá un mercado adicional.

Hemos mostramos cómo se puede implementar manualmente la funcionalidad que Delphi no proporciona automáticamente cuando creamos un control ActiveX a partir de un componente VCL. También ha aprendido a añadir páginas de propiedades a un control ActiveX para ayudar a los usuarios de los mismos.

Finalmente, hemos examinado los ActiveForms, y mostrado cómo podemos crear un control ActiveX a partir de un formulario de Delphi, para posteriormente mostrar éste en Visual Basic o en Internet Explorer, por ejemplo.

En el próximo capítulo echaremos un vistazo a DCOM, que le permitirá implantar servidores y clientes COM en ordenadores conectados a una red.

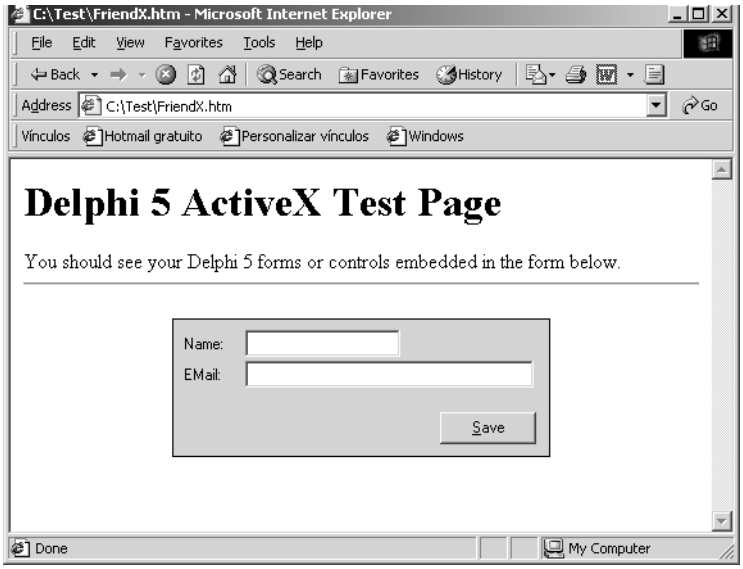


Figura 5.10: Probando el ActiveForm.

Resumen

En este capítulo hemos mostrado cómo aprovechar las ventajas de los controles ActiveX en nuestras aplicaciones Delphi. Como programador de Delphi, normalmente optará por un componente VCL, si existe, pero existen actualmente numerosos controles ActiveX que no tienen su contrapartida VCL.

Seguidamente, hemos mostramos cómo es posible convertir un componente VCL existente en un control ActiveX para que sea utilizado por otros usuarios. Si es Ud. un desarrollador de componentes, esto le abrirá un mercado adicional.

Hemos mostramos cómo se puede implementar manualmente la funcionalidad que Delphi no proporciona automáticamente cuando creamos un control ActiveX a partir de un componente VCL. También ha aprendido a añadir páginas de propiedades a un control ActiveX para ayudar a los usuarios de los mismos.

Finalmente, hemos examinado los ActiveForms, y mostrado cómo podemos crear un control ActiveX a partir de un formulario de Delphi, para posteriormente mostrar éste en Visual Basic o en Internet Explorer, por ejemplo.

En el próximo capítulo echaremos un vistazo a DCOM, que le permitirá implantar servidores y clientes COM en ordenadores conectados a una red.

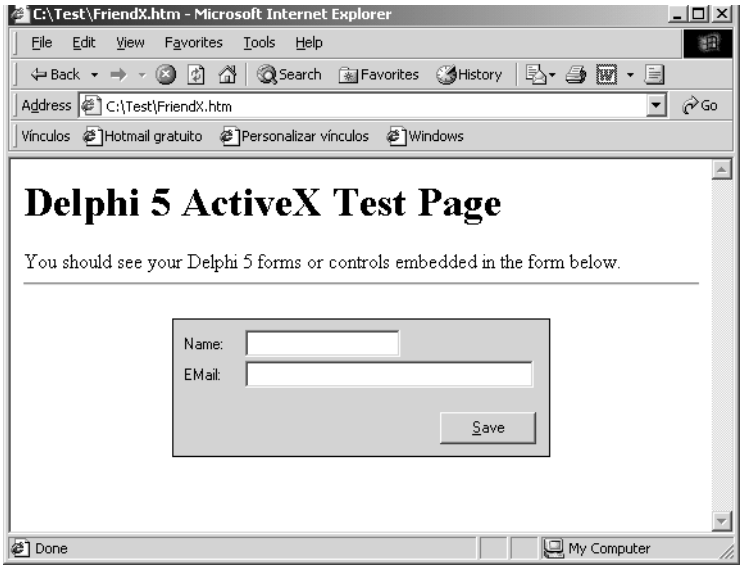


Figura 5.10: Probando el ActiveForm.

Resumen

En este capítulo hemos mostrado cómo aprovechar las ventajas de los controles ActiveX en nuestras aplicaciones Delphi. Como programador de Delphi, normalmente optará por un componente VCL, si existe, pero existen actualmente numerosos controles ActiveX que no tienen su contrapartida VCL.

Seguidamente, hemos mostramos cómo es posible convertir un componente VCL existente en un control ActiveX para que sea utilizado por otros usuarios. Si es Ud. un desarrollador de componentes, esto le abrirá un mercado adicional.

Hemos mostramos cómo se puede implementar manualmente la funcionalidad que Delphi no proporciona automáticamente cuando creamos un control ActiveX a partir de un componente VCL. También ha aprendido a añadir páginas de propiedades a un control ActiveX para ayudar a los usuarios de los mismos.

Finalmente, hemos examinado los ActiveForms, y mostrado cómo podemos crear un control ActiveX a partir de un formulario de Delphi, para posteriormente mostrar éste en Visual Basic o en Internet Explorer, por ejemplo.

En el próximo capítulo echaremos un vistazo a DCOM, que le permitirá implantar servidores y clientes COM en ordenadores conectados a una red.

6 DCOM

Hasta el momento, todo el trabajo que hemos realizado con COM ha sido sobre un único ordenador. En este capítulo examinamos COM Distribuido (DCOM), que realmente no es otra cosa que el acceso a un servidor COM a través de una red. Después de cubrir las bases de DCOM, el capítulo le guiará a través de la programación con DCOM del lado del cliente y del servidor, así como la implementación de conjuntos de datos remotos con DCOM.

Bases de DCOM

DCOM es una tecnología que le permite utilizar todo lo que ha aprendido sobre COM y aplicarlo a una red. Mediante DCOM se hace posible implantar un servidor COM en un ordenador y un cliente que acceda a dicho servidor en otra máquina.

Uno de los aspectos más agradables de DCOM es que una vez que se comprende COM, se sabe prácticamente todo lo necesario para utilizar DCOM. No hay nada nuevo que aprender.

Windows 98, NT y 2000 incluyen DCOM como parte del sistema operativo. Si utiliza Windows 95, deberá visitar:
http://www.microsoft.com/com/dcom/dcom95/download.asp

y descargar DCOM95 para Windows 95. Igualmente deberá descargar la utilidad de configuración de DCOM95.

Si está ejecutando Windows 98, podrá descargar la última versión de DCOM desde: *http://www.microsoft.com/com/dcom/dcom98/dcom1_3.asp*

Programación con DCOM

En este momento ya tiene DCOM instalado en su ordenador y está listo para crear su primer servidor y cliente DCOM. Con Delphi escribir un servidor o cliente DCOM no es más difícil que escribir un servidor o cliente COM estándar. Esta sección le guiará a través de los pasos necesarios para crear un servidor y cliente DCOM.

Creando un Servidor DCOM

Los primeros pasos para la creación de un servidor DCOM son idénticos a los que se utilizan para crear un servidor COM. Comience una nueva aplicación. Seleccione *File | New* en el menú principal de Delphi, y active la página *ActiveX* del Depósito de Objetos. Seleccione *Automation Object* y pulse OK.

Delphi muestra el Asistente para Objetos de Automatización. Rellene los campos según se muestra en la Figura 6.1 y pulse OK.

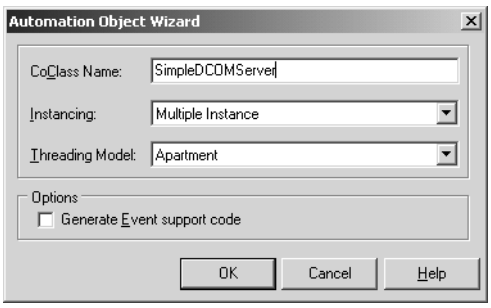


Figura 6.1: El Asistente para Objetos de Automatización en acción.

Ahora guarde el proyecto. Grabe la unidad *Unit1* como *MainForm.pas*, la unidad *Unit2* como *SimpleServer.pas*, y el proyecto como *DCOMServer.dpr*.

Observe que no hemos hecho nada para indicarle a Delphi que estamos creando un servidor DCOM y no un servidor COM estándar**.** Lo único que hemos hecho es asegurarnos de crear un servidor externo al proceso, en lugar de un servidor interno al proceso. Después de todo, el servidor se ejecutará en una máquina remota, lo cual es lo más externo al proceso que se puede obtener.

Delphi debe iniciar automáticamente el Editor de Bibliotecas de Tipos después de que usted rellene el Asistente de Objetos de Automatización. Si esto no es así, seleccione *View | Type Library* del menú principal.

Pulse sobre el nodo *ISimpleDCOMServer* en la Lista de Objetos del Editor de Bibliotecas de Tipos, y pulse el botón *New Method* en la barra de herramientas. Llame al nuevo método *AreYouThere*. Seleccione la página *Parameters* y añada un parámetro llamado *Result*, de tipo *VARIANT_BOOL **, y con un modificador de *[out, retval]*. Si está usando Delphi 3 ó 4, en lugar de añadir un parámetro puede establecer el tipo devuelto por el método *AreYouThere* como *WordBool*.

Nota

No olvide lo discutido en los capítulos anteriores sobre servidores externos al proceso, la serialización y los tipos compatibles con la automatización. Esa discusión es importante para este

capítulo, y deberá siempre asegurarse de utilizar tipos compatibles con la automatización en los métodos que añada a los servidores DCOM. Si está usando Delphi 4 o posterior, el Editor de Bibliotecas de Tipos se encarga de garantizar el cumplimiento de esta regla.

Pulse el botón Refresh Implementation en el Editor de Bibliotecas de Tipos, y a continuación cierre el Editor. Delphi ha creado el esqueleto del código de la función TSimpleDCOMServer.AreYouThere. Modifique la función de la siguiente forma:

```
function TSimpleDCOMServer.AreYouThere: WordBool;
begin
  Result := True;
end;
```

Como puede ver, esta función no hace mucho. Simplemente responde "Sí, estoy aquí". Usaremos este sencillo ejemplo en las siguientes secciones para verificar que DCOM está correctamente instalado su ordenador.

Los Listados 6.1 al 6.4 muestran el código fuente para el servidor *DCOMServer*.

Listado 6.1: DCOMServer - *DCOMServer.dpr*

```
program DCOMServer;
uses
  Forms,
  MainForm in 'MainForm.pas' {frmMain},
  DCOMServer_TLB in 'DCOMServer_TLB.pas',
  SimpleServer in 'SimpleServer.pas' {SimpleDCOMServer: CoClass};
{$R *.TLB}
{$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(TfrmMain, frmMain);
  Application.Run;
end.
```

Listado 6.2: DCOMServer - *SimpleServer.pas*

```
unit SimpleServer;
interface
uses
  ComObj, ActiveX, DCOMServer_TLB;
type
  TSimpleDCOMServer = class(TAutoObject, ISimpleDCOMServer)
  protected
    { Protected declarations }
    function AreYouThere: WordBool; safecall;
  end;
implementation
uses ComServ;
function TSimpleDCOMServer.AreYouThere: WordBool;
begin
```

```
  Result := True;
end;
initialization
  TAutoObjectFactory.Create(ComServer, TSimpleDCOMServer,
    Class_SimpleDCOMServer,
    ciMultiInstance, tmApartment);
end.
```

Listado 6.3: DCOMServer - *DCOMServer_TLB.pas*

```
unit DCOMServer_TLB;
// ************************************************************************ //
// WARNING                                                                  //
// □□□-                                                                     //
// The types declared in this file were generated from data read from a     //
// Type Library. If this type library is explicitly or indirectly (via      //
// another type library referring to this type library) re-imported, or     //
// the 'Refresh' command of the Type Library Editor activated while         //
// editing the Type Library, the contents of this file will be              //
// regenerated and all manual modifications will be lost.                   //
// ************************************************************************ //
// PASTLWTR : $Revision:   1.11.1.75  $
// File generated on 5/15/2001 1:48:02 PM from Type Library described
below.
// ************************************************************************ //
// Type Lib: C:\Book\samples\Chap06\DCOMServer\DCOMServer.tlb
// IID\LCID: {682C7F20-28C9-11D3-B802-0040F67455FE}\0
// Helpfile:
// HelpString: Project1 Library
// Version:    1.0
// ************************************************************************ //
interface
uses Windows, ActiveX, Classes, Graphics, OleCtrls, StdVCL;
// ************************************************************************//
// GUIDS declared in the TypeLibrary. Following prefixes are used:          //
//   Type Libraries     : LIBID_xxxx                                        //
//   CoClasses          : CLASS_xxxx                                        //
//   DISPInterfaces     : DIID_xxxx                                         //
//   Non-DISP interfaces: IID_xxxx                                          //
// ************************************************************************//
const
  LIBID_DCOMServer: TGUID = '{682C7F20-28C9-11D3-B802-0040F67455FE}';
  IID_ISimpleDCOMServer: TGUID = '{682C7F21-28C9-11D3-B802-0040F67455FE}';
  CLASS_SimpleDCOMServer: TGUID =
'{682C7F23-28C9-11D3-B802-0040F67455FE}';
type
// ************************************************************************//
// Forward declaration of interfaces defined in Type Library                //
// ************************************************************************//
  ISimpleDCOMServer = interface;
  ISimpleDCOMServerDisp = dispinterface;
// ************************************************************************//
```

```
// Declaration of CoClasses defined in Type Library                    //
// (NOTE: Here we map each CoClass to its Default Interface)           //
// ********************************************************************//
  SimpleDCOMServer = ISimpleDCOMServer;
// ********************************************************************//
// Interface: ISimpleDCOMServer
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {682C7F21-28C9-11D3-B802-0040F67455FE}
// ********************************************************************//
  ISimpleDCOMServer = interface(IDispatch)
    ['{682C7F21-28C9-11D3-B802-0040F67455FE}']
    function AreYouThere: WordBool; safecall;
  end;
// ********************************************************************//
// DispIntf:  ISimpleDCOMServerDisp
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {682C7F21-28C9-11D3-B802-0040F67455FE}
// ********************************************************************//
  ISimpleDCOMServerDisp = dispinterface
    ['{682C7F21-28C9-11D3-B802-0040F67455FE}']
    function AreYouThere: WordBool; dispid 1;
  end;
  CoSimpleDCOMServer = class
    class function Create: ISimpleDCOMServer;
    class function CreateRemote(const MachineName: string):
ISimpleDCOMServer;
  end;
implementation
uses ComObj;
class function CoSimpleDCOMServer.Create: ISimpleDCOMServer;
begin
  Result := CreateComObject(CLASS_SimpleDCOMServer) as ISimpleDCOMServer;
end;
class function CoSimpleDCOMServer.CreateRemote
  (const MachineName: string): ISimpleDCOMServer;
begin
  Result := CreateRemoteComObject(MachineName, CLASS_SimpleDCOMServer)
    as ISimpleDCOMServer;
end;
end.
```

Listado 6.4: DCOMServer - *MainForm.pas*

```
unit MainForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  StdCtrls;
type
  TfrmMain = class(TForm)
    Label1: TLabel;
```

```
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  frmMain: TfrmMain;
implementation
{$R *.DFM}
end.
```

Instalación de un servidor DCOM

Ahora tenemos un servidor COM (y DCOM) funcional. Instalar y registrar un servidor DCOM es un proceso complicado, principalmente debido a cuestiones de seguridad de Windows. Esta sección le indicará lo que deberá hacer para instalar un servidor DCOM en un servidor Windows NT/2000.

Instalación en un servidor Windows NT/2000

Después de compilar el servidor DCOM a un ejecutable, deberá instalarlo en el servidor. Esta sección asume que está utilizando un servidor Windows NT/2000, configurado como servidor de dominio (lo cual es la configuración recomendada) y como servidor DCOM.

Nota

Es posible utilizar Windows NT/2000 como servidor DCOM aún cuando éste no esté configurado como servidor de dominio. En ese caso es necesario que el cliente tenga una cuenta en el servidor NT/2000. Además, por simplicidad, la cuenta en el servidor deberá tener una clave de acceso idéntica al *login* del cliente.

Los siguientes pasos le guiarán a través del proceso de implantación del servidor DCOM sobre el servidor.

1. Copie *DCOMServer.exe* al servidor y ejecútelo para registrarlo en Windows. Tan pronto la aplicación se inicie, puede cerrarlo inmediatamente.
2. En el menú *Inicio* del servidor, ejecute *Programas | Herramientas Administrativas | Administrador de Usuarios para dominios*. Asumiendo que su red está operativa, ya deberá tener una cuenta de usuario configurada para el ordenador donde se ejecutará el cliente DCOM. En mi red, esa cuenta se llama *Eric*.
3. Seleccione *Usuario | Nuevo Grupo Local* del menú principal del Administrador de Usuarios.

4. Introduzca el nombre de grupo '*Servidores DCOM*' y una descripción adecuada, y pulse el botón *Añadir*. Windows NT/2000 mostrará el cuadro de diálogo *Añadir usuarios y Grupos*.
5. Haga doble clic sobre cada uno de los usuarios que desee añadir al grupo (en otras palabras, sobre cada uno de los usuarios a los que desee dar acceso al servidor DCOM).
6. Pulse OK en ambos cuadros de diálogo para completar la creación del nuevo grupo.
7. Cierre el Administrador de Usuarios y ejecute *DCOMCNFG.EXE*, situado en el directorio *C:\WinNT\System32*. El servidor DCOM recién creado, *SimpleDCOMServer Object*, aparecerá en la lista de los servidores disponibles, como muestra la Figura 6.2.

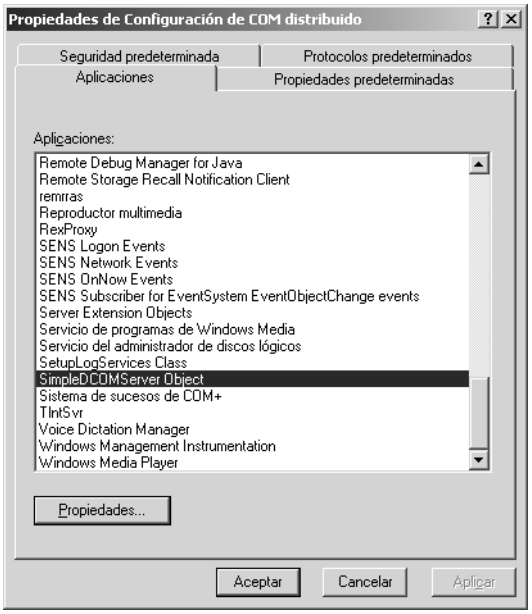


Figura 6.2: *DCOMCNFG* muestra nuestro nuevo servidor DCOM.

8. Seleccione *SimpleDCOMServer* y pulse el botón *Propiedades*. Aparecerá el cuadro de diálogo de propiedades del servidor DCOM. Seleccione la página *Ubicación* y asegúrese de que el servidor está configurado para ejecutarse en el servidor (Figura 6.3).

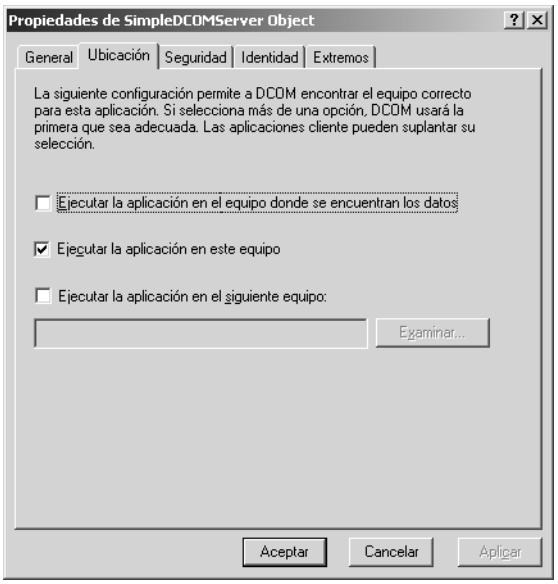


Figura 6.3: DCOMCNFG puede indicarle a Windows en qué ordenador se ejecutará el servidor.

9. Seleccione la página de *Seguridad*, y en ella el botón de radio *Usar permisos de acceso personalizados, usar permisos personalizados de ejecución* y luego *usar permisos predeterminados*. Seguidamente pulse el botón '*Modificar...*' para definir qué usuarios tendrán acceso al servidor DCOM. Aparecerá el cuadro de diálogo que se muestra en la Figura 6.4.

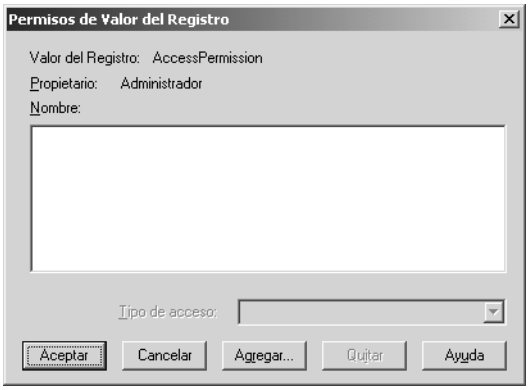


Figura 6.4: Definiendo los usuarios que tienen acceso al servidor DCOM.

10. Pulse el botón '*Añadir...*'. Windows mostrará la lista de grupos definidos en el dominio.

11. Haga doble clic sobre el grupo '*Servidores DCOM*' para añadirlo a la lista de grupos de usuarios con acceso al servidor DCOM. Pulse el botón OK para cerrar el cuadro de diálogo *Añadir Usuarios y Grupos*. Pulse nuevamente OK para cerrar el cuadro de diálogo de edición de permisos.
12. Ahora pulse el botón '*Modificar...*' situado cerca del centro del cuadro de diálogo de la pestaña '*Seguridad*' del cuadro de diálogo de propiedades del servidor DCOM, y defina qué usuarios tendrán permiso para activar desde un equipo remoto el servidor DCOM. Repita la misma secuencia de pasos que acaba de utilizar para definir quiénes tendrán acceso al servidor DCOM.
13. Pulse OK en todos los cuadros de diálogo abiertos para finalizar la configuración del servidor DCOM.

Instalación en Windows 95/98/ME

Como hemos indicado al principio de este capítulo, los servidores DCOM pueden ejecutarse sobre Windows 95/98/ME y Windows NT/2000. Sin embargo, si desea hacer un uso eficiente del servidor DCOM deberá utilizar Windows NT/2000 como servidor DCOM. Si está decidido a usar Windows 9x/ME para alojar el servidor, deberá tener presente un conjunto de aspectos:

- No es posible iniciar automáticamente la ejecución de un servidor DCOM sobre un ordenador remoto equipado con Windows 95/98/ME. Para que la máquina funcione como servidor DCOM, la aplicación servidora debe ser cargada en memoria antes de que cualquier cliente trate de conectarse a ella.
- Deberá cambiar la entrada del registro *HKEY_LOCAL_MACHINE\Software\Microsoft\OLE\EnableRemoteConnect* a '*Y*' en el equipo Windows95/98/ME para que DCOM funcione correctamente.
- También deberá cambiar la entrada del registro *HKEY_LOCAL_MACHINE\Software\Microsoft\OLE\LegacyAuthenticationLevel* al valor 1 en el equipo Windows 95/98/ME. Si la clave no existe, deberá crearla con el tipo *DWORD*.
- Podría necesitar ejecutar *RPCSS.EXE* en el equipo Windows 95/98/ME, aunque esto no debería ser necesario con DCOM versión 1.1 o posteriores.

Asimismo le recomiendo consultar el artículo técnico de Microsoft *#Q165101* (*How to use Windows 95, Windows 98 or Windows ME Computer as a DCOM Server*), localizado en .

Note que Windows 95/98/ME no presenta limitación alguna para ejecutar clientes DCOM. Las únicas limitaciones se presentan cuando actúa como servidor DCOM.

Después de haber configurado el registro en Windows 9x**,** deberá ejecutar *DCOMServer.exe* para registrarlo en Windows. Deje el servidor ejecutándose: recuerde que es imposible conectarse a un servidor DCOM situado en un equipo Windows 9x a menos que éste ya esté cargado en memoria.

Creando un cliente DCOM

Ahora ya tiene un servidor DCOM funcional instalado en un ordenador. Tenga presente que no hemos hecho nada especial en el código fuente para diferenciar este servidor de un servidor COM estándar. De hecho, *DCOMServer* es un servidor COM perfectamente válido, y puede ser utilizado como tal.

Sigamos adelante, y creemos una aplicación cliente para probar el servidor. Inicie una nueva aplicación. Desde el menú principal de Delphi, seleccione *Project | Import Type Library*. Si *DCOMServer (Version 1.0)* no aparece en el listado de servidores, pulse el botón '*Add...*' e importe la biblioteca de tipos *DCOMServer.tlb*. Pulse el botón OK, y Delphi generará el fichero de importación *DCOMServer_TLB.pas*.

Nota

Hemos acabado de crear el servidor DCOM, así que podría simplemente haber usado DCOMserver_TLB.pas en lugar de importar la biblioteca de tipos. Sin embargo, en muchos casos tendrá que acceder a un servidor creado por otro. Deseo estar seguro de que ha comprendido el proceso de importar una biblioteca de tipos.

Los Listados 6.5 y 6.6 contienen el código fuente para un cliente DCOM sencillo.

Listado 6.5: DCOMClient - *DCOMClient.dpr*

```
program DCOMClient;
uses
  Forms,
  MainForm in 'MainForm.pas' {frmMain},
  DCOMServer_TLB in
   'C:\Program Files\Borland\Delphi6\Imports\DCOMServer_TLB.pas';
{$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(TfrmMain, frmMain);
  Application.Run;
end.
```

Listado 6.6: DCOMClient - *MainForm.pas*

```
unit MainForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  StdCtrls, DCOMServer_TLB, ExtCtrls;
type
  TfrmMain = class(TForm)
    pnlBottom: TPanel;
    btnTest: TButton;
    pnlClient: TPanel;
```

```
    Label1: TLabel;
    ecServer: TEdit;
    procedure btnTestClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  frmMain: TfrmMain;
implementation
{$R *.DFM}
procedure TfrmMain.btnTestClick(Sender: TObject);
var
  FServer: ISimpleDCOMServer;
begin
  FServer := CoSimpleDCOMServer.CreateRemote(ecServer.Text);
  if FServer.AreYouThere then
    ShowMessage('I am here.')
  else
    ShowMessage('Where''d I go?');
end;
end.
```

Este código es casi idéntico al código escrito en los capítulos anteriores al crear clientes COM. De hecho, la única diferencia es que aquí se utiliza *CoSimpleDCOMServer.CreateRemote* en lugar de *CoSimpleDCOMServer.Create*. Sin embargo, si analiza el código fuente de *DCOMServer_TLB.pas*, verá que la función *CoSimpleDCOMServer.Create* está todavía definida. Eso significa que se puede utilizar el servidor como servidor local (externo al proceso) o como servidor remoto (DCOM) cambiando una simple línea en el código del cliente.

Truco

Se puede lanzar un servidor COM local llamando a CreateRemote y pasándole '\\.' o el nombre del ordenador local como parámetro. De esta forma no es necesario cambiar ni siquiera una línea de código - simplemente cambiar el nombre de la máquina en la llamada a CreateRemote.

Hay un detalle que deberá tener presente antes de que pueda ejecutar la aplicación cliente. Deberá registrar la biblioteca de tipos del servidor DCOM en el ordenador cliente. La forma más sencilla de hacerlo es usando TREGSVR, de la siguiente forma:

```
TRegSvr  DCOMServer.tlb
```

Después de registrar la biblioteca de tipos, será posible ejecutar la aplicación cliente. La Figura 6.5 muestra el cliente en tiempo de ejecución.

Introduzca el nombre del servidor (en mi caso, *SERVER*), y pulse el botón '*Test*'. Si todo marcha bien, verá un cuadro de mensaje en el que se leerá "*I am here*". Si obtiene un mensaje de error indicando que el acceso al servidor fue denegado, probablemente se trata de que ha configurado algo incorrectamente mediante DCOMCNFG.

Bien, esto ha sido extraordinariamente simple. Ahora ya ha visto todo lo que necesita conocer sobre DCOM para utilizarlo en sus aplicaciones Delphi. En la próxima sección, le mostraré cómo usar DCOM para implementar un conjunto de datos remoto.

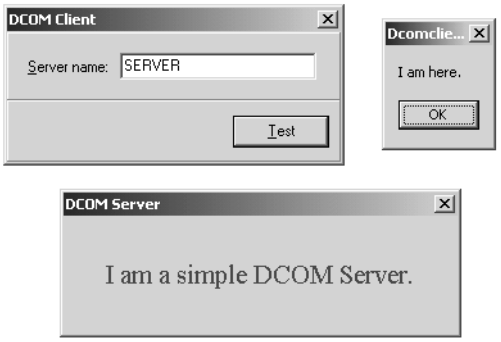


Figura 6.5: *DCOMClient* permite al usuario seleccionar el servidor DCOM en tiempo de ejecución.

Nota

Todo lo que ha aprendido sobre servidores COM externos al proceso se aplica también a servidores DCOM. Por ejemplo, puede utilizar los mismos métodos aprendidos en el Capítulo 4, "Automatización", para implementar eventos COM y funciones de respuesta en un servidor DCOM. De hecho, en la próxima sección crearemos una aplicación que hace justamente eso.

Implementación de conjuntos de datos remotos con DCOM

Un conjunto de datos remotos, como puede imaginar, es simplemente un conjunto de datos que reside en una máquina remota. Normalmente, cuando desarrolla aplicaciones de bases de datos con Delphi puede tomar dos caminos diferentes. Puede utilizar soluciones poco costosas basadas en servidores de ficheros locales, como Paradox o dBase, o puede adquirir la versión cliente/servidor de Delphi y usar el BDE contra una base de datos SQL, o MIDAS.

Lo que presentaremos aquí es una tercera opción: puede utilizar DCOM para colocar el servidor de bases de datos en un ordenador, a la vez que sitúa la aplicación cliente en otra máquina.

Esta aplicación ha sido desarrollada usando Delphi 5, para aprovechar las ventajas de los nuevos componentes para servidores COM introducidos en Delphi 5. Si está utilizando una versión anterior de Delphi, la aplicación no se compilará tal cual. Sin embargo, usando las técnicas descritas en el Capítulo 4 para soportar eventos de disp-interfaces, podrá modificar el código para que funcione en versiones anteriores de Delphi.

Definiendo la aplicación

Antes de concentrarnos en la codificación, tomemos un momento para definir el problema que vamos a resolver. Vamos a crear una base de datos que contiene una única tabla para una aplicación sencilla de control de inventario. La estructura de la tabla se muestra en Tabla 6.1.

Tabla 6.1: Estructura de la tabla *Parts*

Campo	Tipo
PartNo	Código de artículo. Alfanumérico de 10 caracteres. Clave primaria.
Description	Descripción del artículo. Alfanumérico de 25 caracteres.
UnitPrice	Precio unitario. de tipo *Currency*.
OnHand	Existencia en almacén, de tipo entero.
ReOrder	Existencia mínima permisible, de tipo entero.

Como puede ver, se trata de una estructura muy sencilla. Cada registro de la tabla almacena un código de artículo, su descripción, el precio unitario y la cantidad mínima al alcanzarse la cual se emite automáticamente un pedido.

La aplicación en sí está formada por tres programas. El servidor DCOM se encarga de la interfaz con la base de datos. Nadie se comunica con la base de datos si no es a través del servidor DCOM. Adicionalmente al servidor DCOM, hay una aplicación administrativa que permite al usuario añadir y eliminar artículos, o actualizar el precio de los mismos.

El último programa consiste de una aplicación de terminal punto de venta. Esta aplicación permite a los usuarios consultar el precio de un artículo y actualizar el inventario cuando se vende un artículo.

Creando el servidor

El servidor contendrá todo el código necesario para interactuar con la base de datos. Para este ejemplo utilizaré una base de datos de dBase. El BDE no será instalado en el ordenador cliente, así que el servidor deberá contener toda la

funcionalidad requerida por la aplicación. Tomemos unos momentos para considerar las funciones básicas que realizará la aplicación:

- Obtener un listado de algunos o todos los artículos en la tabla.
- Añadir un nuevo artículo a la tabla.
- Eliminar un artículo de la tabla.
- Modificar la descripción o precio unitario de un artículo.
- Consultar información sobre un artículo, como descripción y precio unitario.
- Descontar una cantidad de artículos del inventario.

Probablemente, la aplicación podría cumplir un número mayor de tareas, pero las enumeradas en el listado anterior son suficientes para empezar. Observe que no hemos incluido en la lista una función para realizar un pedido de uno o más artículos. Los pedidos no los realiza la aplicación cliente – el servidor emitirá automáticamente un nuevo pedido cuando la cantidad en existencias rebase el límite inferior (Al menos en teoría. No voy a incluir código en el servidor para emitir el pedido, pero le mostraré cómo implementar esta característica).

Serializando registros

Ya que la base de datos reside en el servidor, vamos a necesitar algún mecanismo para la transferencia de los datos del servidor al cliente. Para satisfacer esta necesidad, utilizaré un *array* variante. Recuerde que hemos examinado los *arrays* variantes en el Capítulo 4, donde indiqué que los *arrays* variantes pueden ser utilizados para transferir estructuras complejas a través de las fronteras de procesos.

Podemos escoger dos métodos diferentes para serializar los datos. En primer lugar, podemos definir una estructura de registro en memoria, y pasar estructuras de ese tipo como *arrays* de bytes. Por ejemplo,

```
type
  TPartRecord = packed record
    PartNumber: string[10];
    Description: string[25];
    UnitPrice: Double;
    OnHand: Integer;
  end;
```

Observe que he dejado fuera el campo *Reorder*. La aplicación cliente no necesita saber nada con relación a este campo, así que podemos ahorrar una pequeña cantidad de tiempo y espacio al no empaquetar esa información.

Con este método, después de que el servidor lee los datos de la tabla y rellena los campos del registro *TPartRecord*, puede empaquetar los datos en un *array* variante, de la siguiente forma:

```
function RecordToVariant(Part: TPartRecord): OleVariant;
var
  P: Pointer;
begin
  Result := VarArrayCreate([0, sizeof(TPartRecord)], varByte);
  P := VarArrayLock(Result);
  Move(Part, P^, sizeof(Part));
  VarArrayUnlock(Result);
end;
```

Una vez que el *array* variante es serializado hacia el cliente, éste puede convertirlo en un *TPartRecord*, mediante el siguiente código:

```
function VariantToRecord(V: OleVariant): TPartRecord;
var
  P: Pointer;
begin
  P := VarArrayLock(V);
  Move(P^, Result, sizeof(TPartRecord));
  VarArrayUnlock(V);
end;
```

Aunque este proceso funciona, hay una sola cosa que no me gusta de él: si la estructura del registro cambia, el servidor y el cliente deberán ser actualizados para acomodarlos a la nueva estructura.

En lugar de ello, vamos a utilizar dos *arrays* variantes para serializar los datos. El primer *array* será enviado del cliente al servidor, y contiene una lista de los campos en los que el cliente está interesado. El segundo *array* es devuelto por el servidor al cliente, y contiene una lista de los valores de los campos, en el mismo orden de los campos del primer *array*. Por ejemplo, si el cliente está interesado solamente en el código de artículo y su descripción, enviará el siguiente *array* al servidor:

```
Campo
PARTNO
DESCRIPTION
```

Si añadimos otro campo a la tabla (por ejemplo *ExpectedDate*, que indicaría cuándo se espera que arribe la próxima remesa del artículo), el código de serialización podrá simplemente añadir otro campo al final de esta lista. Una vez que el servidor lea los datos de la base de datos, devolverá un *array* variante de *arrays* variantes al cliente, y la respuesta se asemejará a lo siguiente:

```
Filas
Fila 1
Fila 2
...
```

Cada fila está compuesta de un número de campos, como.

```
Valores de campos
P123
Widget
...
```

La aplicación servidora

Con la discusión anterior todavía en mente, echemos un vistazo a los Listados 6.7 al 6.12. Ellos contienen el código fuente para la parte del lado del servidor de esta aplicación. El Listado 6.7 muestra el programa principal del servidor.

Listado 6.7: PartSrv - *PartSrv.dpr*

```
program PartSrv;
uses
  Forms,
  MainForm in 'MainForm.pas' {Form1},
  PartSrv_TLB in 'PartSrv_TLB.pas',
  PartServerImpl in 'PartServerImpl.pas' {PartServer: CoClass},
  Parts in 'Parts.pas',
  DataModule in 'DataModule.pas' {PartDM: TDataModule};
{$R *.TLB}
{$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.CreateForm(TPartDM, PartDM);
  Application.Run;
end.
```

El Listado 6.8 muestra el fichero de importación de la biblioteca de tipos para el servidor.

Listado 6.8: PartSrv - *PartSrv_TLB.pas*

```
unit PartSrv_TLB;
// ************************************************************************ //
// WARNING                                                                  //
// □□□-                                                                     //
// The types declared in this file were generated from data read from a     //
// Type Library. If this type library is explicitly or indirectly (via      //
// another type library referring to this type library) re-imported, or     //
// the 'Refresh' command of the Type Library Editor activated while         //
// editing the Type Library, the contents of this file will be              //
// regenerated and all manual modifications will be lost.                   //
// ************************************************************************ //
// PASTLWTR : $Revision:   1.11.1.75  $
// File generated on 5/15/01 11:26:35 AM from Type Library described
below.
// ************************************************************************ //
// Type Lib: C:\Book\samples\Chap06\PartServer2\PartSrv.tlb
// IID\LCID: {046CB030-4824-11D3-B83F-0040F67455FE}\0
// Helpfile:
// HelpString: PartSrv Library
// Version:    1.0
// ************************************************************************ //
interface
```

```
uses Windows, ActiveX, Classes, Graphics, OleCtrls, StdVCL;
// ************************************************************************//
// GUIDS declared in the TypeLibrary. Following prefixes are used:      //
//   Type Libraries     : LIBID_xxxx                                       //
//   CoClasses          : CLASS_xxxx                                       //
//   DISPInterfaces     : DIID_xxxx                                        //
//   Non-DISP interfaces: IID_xxxx                                         //
// ************************************************************************//
const
  LIBID_PartSrv: TGUID = '{046CB030-4824-11D3-B83F-0040F67455FE}';
  IID_IPartServer: TGUID = '{046CB031-4824-11D3-B83F-0040F67455FE}';
  DIID_IPartServerEvents: TGUID =
'{046CB033-4824-11D3-B83F-0040F67455FE}';
  CLASS_PartServer: TGUID = '{046CB035-4824-11D3-B83F-0040F67455FE}';
// ************************************************************************//
// Declaration of Enumerations defined in Type Library                    //
// ************************************************************************//
// FieldNumbers constants
type
  FieldNumbers = TOleEnum;
const
  fnPartNumber = $00000000;
  fnDescription = $00000001;
  fnUnitPrice = $00000002;
  fnOnHand = $00000003;
  fnReorder = $00000004;
type
// ************************************************************************//
// Forward declaration of interfaces defined in Type Library              //
// ************************************************************************//
  IPartServer = interface;
  IPartServerDisp = dispinterface;
  IPartServerEvents = dispinterface;
// ************************************************************************//
// Declaration of CoClasses defined in Type Library                       //
// (NOTE: Here we map each CoClass to its Default Interface)              //
// ************************************************************************//
  PartServer = IPartServer;
// ************************************************************************//
// Interface: IPartServer
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {046CB031-4824-11D3-B83F-0040F67455FE}
// ************************************************************************//
  IPartServer = interface (IDispatch)
    ['{046CB031-4824-11D3-B83F-0040F67455FE}']
    function GetAll(varFields: OleVariant): OleVariant; safecall;
    function GetPart(const PartNumber: WideString;
      varFields: OleVariant): OleVariant; safecall;
    procedure AddPart(varFields:OleVariant;varValues:OleVariant);safecall;
    procedure DeletePart(const PartNumber: WideString); safecall;
    procedure ModifyPart(const PartNumber: WideString;
      varFields: OleVariant; varValues: OleVariant); safecall;
```

```
    procedure UsePart(const PartNumber: WideString;
      Quantity: Integer); safecall;
  end;
// *********************************************************************//
// DispIntf:  IPartServerDisp
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {046CB031-4824-11D3-B83F-0040F67455FE}
// *********************************************************************//
  IPartServerDisp = dispinterface
    ['{046CB031-4824-11D3-B83F-0040F67455FE}']
    function GetAll(varFields: OleVariant): OleVariant; dispid 1;
    function GetPart(const PartNumber: WideString;
      varFields: OleVariant): OleVariant; dispid 2;
    procedure AddPart(varFields:OleVariant;varValues:OleVariant);dispid 3;
    procedure DeletePart(const PartNumber: WideString); dispid 4;
    procedure ModifyPart(const PartNumber: WideString;
      varFields: OleVariant; varValues: OleVariant); dispid 5;
    procedure UsePart(const PartNumber: WideString;
      Quantity: Integer); dispid 6;
  end;
// *********************************************************************//
// DispIntf:  IPartServerEvents
// Flags:     (0)
// GUID:      {046CB033-4824-11D3-B83F-0040F67455FE}
// *********************************************************************//
  IPartServerEvents = dispinterface
    ['{046CB033-4824-11D3-B83F-0040F67455FE}']
    procedure OnAdd(varFields: OleVariant; varValues:OleVariant);dispid 1;
    procedure OnDelete(const PartNumber: WideString); dispid 2;
    procedure OnModify(const PartNumber: WideString;varFields: OleVariant;
      varValues: OleVariant); dispid 3;
  end;
  CoPartServer = class
    class function Create: IPartServer;
    class function CreateRemote(const MachineName: string): IPartServer;
  end;
implementation
uses ComObj;
class function CoPartServer.Create: IPartServer;
begin
  Result := CreateComObject(CLASS_PartServer) as IPartServer;
end;
class function CoPartServer.CreateRemote(const MachineName: string):
IPartServer;
begin
  Result := CreateRemoteComObject(MachineName, CLASS_PartServer)
    as IPartServer;
end;
end.
```

El Listado 6.9 define una clase auxiliar que es usada para copiar datos desde y hacia un conjunto de datos, y desde/hacia un *array* variante.

Listado 6.9: PartSrv - *Parts.pas*

```
unit Parts;
interface
uses
  DB, PartSrv_TLB;
type
  TPart = class
    PartNumber: string;
    Description: string;
    UnitPrice: Double;
    OnHand: Integer;
    Reorder: Integer;
  public
    function ToVariant(varFields: OleVariant): OleVariant;
    procedure FromVariant(varFields: OleVariant; varValues: OleVariant);
    procedure CreateFromDataSet(DataSet: TDataSet);
  end;
implementation
{ TPartRecord }
procedure TPart.CreateFromDataSet(DataSet: TDataSet);
begin
  PartNumber := DataSet.FieldByName('PARTNO').AsString;
  Description := DataSet.FieldByName('DESC').AsString;
  UnitPrice := DataSet.FieldByName('UNITPRICE').AsFloat;
  OnHand := DataSet.FieldByName('ONHAND').AsInteger;
  Reorder := DataSet.FieldByName('REORDER').AsInteger;
end;
procedure TPart.FromVariant(varFields: OleVariant; varValues: OleVariant);
var
  Index: Integer;
  FieldNumber: Integer;
begin
  for Index := VarArrayLowBound(varFields, 1)
      to VarArrayHighBound(varFields, 1) do begin
    FieldNumber := varFields[Index];
    case FieldNumber of
      fnPartNumber: PartNumber := varValues[Index];
      fnDescription: Description := varValues[Index];
      fnUnitPrice: UnitPrice := varValues[Index];
      fnOnHand: OnHand := varValues[Index];
      fnReorder: Reorder := varValues[Index];
    end;
  end;
end;
function TPart.ToVariant(varFields: OleVariant): OleVariant;
var
  Index: Integer;
```

```
  FieldNumber: Integer;
begin
  Result := VarArrayCreate([VarArrayLowBound(varFields, 1),
              VarArrayHighBound(varFields, 1)], varVariant);
  for Index := VarArrayLowBound(varFields, 1)
      to VarArrayHighBound(varFields, 1) do begin
    FieldNumber := varFields[Index];
    case FieldNumber of
      fnPartNumber: Result[Index] := PartNumber;
      fnDescription: Result[Index] := Description;
      fnUnitPrice: Result[Index] := UnitPrice;
      fnOnHand: Result[Index] := OnHand;
      fnReorder: Result[Index] := Reorder;
    end;
  end;
end;
end.
```

El Listado 6.10 muestra el módulo de datos para el servidor COM. El módulo de datos contiene código para añadir un artículo a la base de datos, eliminar un artículo de la base de datos, y actualizar un artículo. Para simplificar, hemos utilizado dBase y un objeto *TTable* para acceder a la tabla *Parts*. En una situación real, preferirá utilizar un *TQuery*, ya que con éste podrá restringir la cantidad de filas devueltas a la aplicación cliente.

Listado 6.10: PartSrv - *DataModule.pas*

```
unit DataModule;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  Db, DBTables;
type
  TPartDM = class(TDataModule)
    dbParts: TDatabase;
    tblParts: TTable;
    tblPartsPARTNO: TStringField;
    tblPartsDESC: TStringField;
    tblPartsUNITPRICE: TFloatField;
    tblPartsONHAND: TFloatField;
    tblPartsREORDER: TFloatField;
  private
    { Private declarations }
  public
    { Public declarations }
    procedure GetAllParts(varFields: OleVariant; var varValues:
OleVariant);
    procedure GetPart(PartNumber: string;
      varFields: OleVariant; var varValues: OleVariant);
```

```
    function UsePart(PartNumber: string; Quantity: Integer): Integer;
    function AddPart(varFields: OleVariant; varValues: OleVariant):
string;
    function DeletePart(PartNo: string): Boolean;
    function ModifyPart(PartNo: string; varFields: OleVariant;
      varValues: OleVariant): Boolean;
  end;
var
  PartDM: TPartDM;
implementation
uses
  Parts, PartSrv_TLB;
{$R *.DFM}
{ TPartDM }
function TPartDM.AddPart(varFields: OleVariant;
  varValues: OleVariant): string;
var
  Part: TPart;
begin
  Part := TPart.Create;
  try
    Part.FromVariant(varFields, varValues);
    with tblParts do begin
      Append;
      tblPartsPARTNO.AsString := Part.PartNumber;
      tblPartsDESC.AsString := Part.Description;
      tblPartsUNITPRICE.AsFloat := Part.UnitPrice;
      tblPartsONHAND.AsInteger := Part.OnHand;
      tblPartsREORDER.AsInteger := Part.Reorder;
      Post;
      Result := Part.PartNumber;
    end;
  finally
    Part.Free;
  end;
end;
function TPartDM.DeletePart(PartNo: string): Boolean;
begin
  Result := tblParts.Locate('PARTNO', PartNo, []);
  if Result then
    tblParts.Delete;
end;
procedure TPartDM.GetAllParts(varFields: OleVariant;
  var varValues: OleVariant);
var
  Part: TPart;
  Index: Integer;
begin
  Part := TPart.Create;
  try
    with tblParts do begin
      varValues := VarArrayCreate([0, RecordCount - 1], varVariant);
```

```
      Index := 0;
      First;
      while not EOF do begin
        Part.CreateFromDataSet(tblParts);
        varValues[Index] := Part.ToVariant(varFields);
        Inc(Index);
        Next;
      end;
    end;
  finally
    Part.Free;
  end;
end;
procedure TPartDM.GetPart(PartNumber: string; varFields: OleVariant;
  var varValues: OleVariant);
var
  Part: TPart;
begin
  if tblParts.Locate('PARTNO', PartNumber, []) then begin
    Part := TPart.Create;
    try
      Part.CreateFromDataSet(tblParts);
      varValues := Part.ToVariant(varFields);
    finally
      Part.Free;
    end;
  end;
end;
function TPartDM.ModifyPart(PartNo: string; varFields: OleVariant;
  varValues: OleVariant): Boolean;
var
  Index: Integer;
begin
  Result := False;
  if tblParts.Locate('PARTNO', PartNo, []) then begin
    with tblParts do begin
      Edit;
      try
        for Index := VarArrayLowBound(varFields, 1)
            to varArrayHighBound(varFields, 1) do begin
          case varFields[Index] of
            fnPartNumber: tblPartsPARTNO.AsString := varValues[Index];
            fnDescription: tblPartsDESC.AsString := varValues[Index];
            fnUnitPrice: tblPartsUNITPRICE.AsFloat := varValues[Index];
            fnOnHand: tblPartsONHAND.AsInteger := varValues[Index];
            fnReorder: tblPartsREORDER.AsInteger := varValues[Index];
          end;
        end;
        Post;
        Result := True;
      except
        Cancel;
```

```
      end;
    end;
  end;
end;
function TPartDM.UsePart(PartNumber: string; Quantity: Integer): Integer;
begin
  if tblParts.Locate('PARTNO', PartNumber, []) then begin
    if Quantity > tblPartsONHAND.AsInteger then
      raise Exception.CreateFmt('You requested %d of part %s.  ' +
        'That part currently has an onhand quantity of %d.',
        [Quantity, PartNumber, tblPartsONHAND.AsInteger]);
    tblParts.Edit;
    Result := tblPartsONHAND.AsInteger - Quantity;
    tblPartsONHAND.AsInteger := Result;
    tblParts.Post;
    // Check remaining inventory here.  If Result <
tblPartsREORDER.AsInteger,
    // then trigger a routine to order more parts automatically.
  end else
    raise Exception.CreateFmt('Unknown part number %s.', [PartNumber]);
end;
end.
```

El Listado 6.11 define la clase que implementa la interfaz COM *IPartServer*. Debido a que el servidor enviará una notificación a todos los clientes conectados siempre que alguien añada, elimine o modifique una pieza, he utilizado las técnicas de disp-interfaces descritas en el Capítulo 4 para implementar los eventos COM.

Listado 6.11: PartSvr - *PartServerImpl*

```
unit PartServerImpl;
interface
uses
  ComObj, ActiveX, AxCtrls, PartSrv_TLB;
type
  TPartServer = class(TAutoObject, IConnectionPointContainer, IPartServer)
  private
    { Private declarations }
    FConnectionPoints: TConnectionPoints;
    FEvents: IPartServerEvents;
    FObjectID: Integer;
  public
    procedure Initialize; override;
    destructor Destroy; override;
  protected
    { Protected declarations }
    property ConnectionPoints: TConnectionPoints read FConnectionPoints
      implements IConnectionPointContainer;
    procedure EventSinkChanged(const EventSink: IUnknown); override;
    function GetAll(varFields: OleVariant): OleVariant; safecall;
    function GetPart(const PartNumber: WideString;
```

```
      varFields: OleVariant): OleVariant; safecall;
    procedure AddPart(varFields, varValues: OleVariant); safecall;
    procedure DeletePart(const PartNumber: WideString); safecall;
    procedure ModifyPart(const PartNumber: WideString; varFields,
      varValues: OleVariant); safecall;
    procedure UsePart(const PartNumber: WideString; Quantity: Integer);
      safecall;
    function GetEnumerator: IEnumConnections;
  end;
implementation
uses Windows, ComServ, DataModule;
procedure TPartServer.EventSinkChanged(const EventSink: IUnknown);
begin
  FEvents := EventSink as IPartServerEvents;
end;
procedure TPartServer.Initialize;
begin
  inherited Initialize;
  FConnectionPoints := TConnectionPoints.Create(Self);
  if AutoFactory.EventTypeInfo  nil then
    FConnectionPoints.CreateConnectionPoint(AutoFactory.EventIID,
      ckMulti, EventConnect);
  RegisterActiveObject(self as IUnknown, CLASS_PartServer,
    ACTIVEOBJECT_WEAK, FObjectID);
end;
destructor TPartServer.Destroy;
begin
  RevokeActiveObject(FObjectID, nil);
  inherited Destroy;
end;
function TPartServer.GetAll(varFields: OleVariant): OleVariant;
begin
  PartDM.GetAllParts(varFields, Result);
end;
function TPartServer.GetPart(const PartNumber: WideString;
  varFields: OleVariant): OleVariant;
begin
  PartDM.GetPart(PartNumber, varFields, Result);
end;
procedure TPartServer.AddPart(varFields, varValues: OleVariant);
var
  Enum: IEnumConnections;
  ConnectData: TConnectData;
  Fetched: Cardinal;
begin
  PartDM.AddPart(varFields, varValues);
  Enum := GetEnumerator;
  if Enum  nil then begin
    while Enum.Next(1, ConnectData, @Fetched) = S_OK do
      if ConnectData.pUnk  nil then
        (ConnectData.pUnk as IPartServerEvents).OnAdd(varFields,
varValues);
```

```
  end;
end;
procedure TPartServer.DeletePart(const PartNumber: WideString);
var
  Enum: IEnumConnections;
  ConnectData: TConnectData;
  Fetched: Cardinal;
begin
  if PartDM.DeletePart(PartNumber) then begin
    Enum := GetEnumerator;
    if Enum  nil then begin
      while Enum.Next(1, ConnectData, @Fetched) = S_OK do
        if ConnectData.pUnk  nil then
          (ConnectData.pUnk as IPartServerEvents).OnDelete(PartNumber);
    end;
  end;
end;
procedure TPartServer.ModifyPart(const PartNumber: WideString;
  varFields, varValues: OleVariant);
var
  Enum: IEnumConnections;
  ConnectData: TConnectData;
  Fetched: Cardinal;
begin
  PartDM.ModifyPart(PartNumber, varFields, varValues);
  Enum := GetEnumerator;
  if Enum  nil then begin
    while Enum.Next(1, ConnectData, @Fetched) = S_OK do
      if ConnectData.pUnk  nil then
        (ConnectData.pUnk as IPartServerEvents).OnModify(PartNumber,
           varFields, varValues);
  end;
end;
procedure TPartServer.UsePart(const PartNumber: WideString;
  Quantity: Integer);
var
  NewQuantity: Integer;
  Enum: IEnumConnections;
  ConnectData: TConnectData;
  Fetched: Cardinal;
  varFields: OleVariant;
  varValues: OleVariant;
begin
  NewQuantity := PartDM.UsePart(PartNumber, Quantity);

  varFields := VarArrayCreate([0, 0], varVariant);
  varValues := VarArrayCreate([0, 0], varVariant);
  varFields[0] := fnOnHand;
  varValues[0] := NewQuantity;
  Enum := GetEnumerator;
  if Enum  nil then begin
    while Enum.Next(1, ConnectData, @Fetched) = S_OK do
```

```
      if ConnectData.pUnk  nil then
        (ConnectData.pUnk as IPartServerEvents).OnModify(PartNumber,
          varFields, varValues);
  end;
end;
function TPartServer.GetEnumerator: IEnumConnections;
var
  Container: IConnectionPointContainer;
  ConnectionPoint: IConnectionPoint;
begin
  OleCheck(QueryInterface(IConnectionPointContainer, Container));
  OleCheck(Container.FindConnectionPoint(AutoFactory.EventIID,
    ConnectionPoint));
  ConnectionPoint.EnumConnections(Result);
end;
initialization
  TAutoObjectFactory.Create(ComServer, TPartServer, Class_PartServer,
    ciMultiInstance, tmApartment);
end.
```

El Listado 6.12 muestra el código fuente del formulario principal del servidor. Es un formulario sencillo, que muestra una rejilla con el contenido de la tabla *Parts*. He hecho esto para que puede ver cómo la base de datos es actualizada en la medida en que la aplicación cliente interactúa con el servidor.

Listado 6.12: PartSrv - *MainForm.pas*

```
unit MainForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  StdCtrls, Grids, DBGrids, ExtCtrls, Db;
type
  TForm1 = class(TForm)
    DataSource1: TDataSource;
    pnlBottom: TPanel;
    pnlClient: TPanel;
    DBGrid1: TDBGrid;
    btnClose: TButton;
    procedure btnCloseClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
uses DataModule;
{$R *.DFM}
```

```
procedure TForm1.btnCloseClick(Sender: TObject);
begin
  Close;
end;
end.
```

Después de compilar el servidor, deberá instalarlo en el ordenador servidor, y configurarlo mediante el procedimiento descrito anteriormente en este capítulo, en la sección "Instalación de un servidor DCOM".

La Figura 6.6 muestra la interfaz de usuario del servidor.

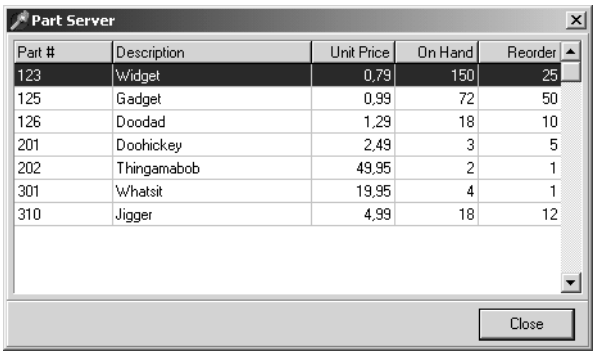


Figura 6.6: Aunque *PartSrv* es un servidor DCOM, puede ser ejecutado también como aplicación independiente.

Creando el cliente administrativo

Ahora que el servidor de artículos ha sido creado, instalado y configurado, creemos la primera de las dos aplicaciones clientes que accederán a él.

El cliente administrativo se utiliza para añadir nuevos artículos, eliminar artículos y actualizar información acerca de estos, como su precio unitario, cantidad disponible, y cantidad mínima aceptable.

Los Listados 6.13 al 6.15 muestran el código fuente del cliente administrativo. El código principal de la aplicación está contenido en el Listado 6.14.

Listado 6.13: PartAdmin - *PartAdmin.dpr*

```
program PartAdmin;
uses
  Forms,
  MainForm in 'MainForm.pas' {frmMain},
  PartForm in 'PartForm.pas' {frmPart},
  PartSrv_TLB in 'PartSrv_TLB.pas';
{$R *.RES}
begin
  Application.Initialize;
```

```
  Application.CreateForm(TfrmMain, frmMain);
  Application.Run;
end.
```

Listado 6.14: PartAdmin - *MainForm.pas*

```
unit MainForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  Grids, ComObj, ActiveX, PartSrv_TLB, Menus, ExtCtrls, OleServer,
StdCtrls;
type
  TfrmMain = class(TForm)
    MainMenu1: TMainMenu;
    pnlClient: TPanel;
    StringGrid1: TStringGrid;
    pnlBottom: TPanel;
    File1: TMenuItem;
    Exit1: TMenuItem;
    Edit1: TMenuItem;
    InsertPart1: TMenuItem;
    EditPart1: TMenuItem;
    DeletePart1: TMenuItem;
    PartServer1: TPartServer;
    ecServer: TEdit;
    btnConnect: TButton;
    Label1: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure Exit1Click(Sender: TObject);
    procedure InsertPart1Click(Sender: TObject);
    procedure DeletePart1Click(Sender: TObject);
    procedure EditPart1Click(Sender: TObject);
    procedure PartServer1Add(Sender: TObject; varFields,
      varValues: OleVariant);
    procedure PartServer1Delete(Sender: TObject;
      var PartNumber: OleVariant);
    procedure PartServer1Modify(Sender: TObject;
      var PartNumber: OleVariant; varFields, varValues: OleVariant);
    procedure btnConnectClick(Sender: TObject);
  private
    { Private declarations }
    FFields: OleVariant;
    procedure LoadPartGrid;
    procedure LoadRow(Row: Integer; varFields, varValues: OleVariant);
  public
    { Public declarations }
  end;
var
  frmMain: TfrmMain;
implementation
```

```
uses PartForm;
{$R *.DFM}
procedure TfrmMain.FormCreate(Sender: TObject);
begin
  FFields := VarArrayCreate([0, 4], varInteger);
  FFields[0] := fnPartNumber;
  FFields[1] := fnDescription;
  FFields[2] := fnUnitPrice;
  FFields[3] := fnOnHand;
  FFields[4] := fnReorder;
end;
procedure TfrmMain.btnConnectClick(Sender: TObject);
begin
  // If no server was entered, we'll use the local machine
  if ecServer.Text  '' then
    PartServer1.RemoteMachineName := ecServer.Text;
  PartServer1.Connect;
  // Load all parts from the database server
  LoadPartGrid;
  // Enable the edit menu
  Edit1.Enabled := True;
end;
procedure TfrmMain.LoadRow(Row: Integer; varFields,varValues: OleVariant);
var
  ValueIndex: Integer;
  ColIndex: Integer;
begin
  for ValueIndex := VarArrayLowBound(varValues, 1)
      to VarArrayHighBound(varValues, 1) do
    for ColIndex := VarArrayLowBound(FFields, 1)
        to VarArrayHighBound(FFields, 1) do
      if FFields[ColIndex] = varFields[ValueIndex] then
        StringGrid1.Cells[ColIndex, Row] := varValues[ValueIndex];
end;
procedure TfrmMain.LoadPartGrid;
var
  varValues: OleVariant;
  Index: Integer;
  FieldNumber: Integer;
  Row: Integer;
begin
  varValues := PartServer1.GetAll(FFields);
  StringGrid1.RowCount := VarArrayHighBound(varValues, 1) -
    VarArrayLowBound(varValues, 1) + 2;
  StringGrid1.ColCount := VarArrayHighBound(FFields, 1) -
    VarArrayLowBound(FFields, 1) + 1;
  for Index := VarArrayLowBound(FFields, 1)
      to VarArrayHighBound(FFields, 1) do begin
    FieldNumber := FFields[Index];
    case FieldNumber of
      fnPartNumber: StringGrid1.Cells[Index, 0] := 'Part #';
      fnDescription: StringGrid1.Cells[Index, 0] := 'Description';
```

```
      fnUnitPrice: StringGrid1.Cells[Index, 0] := 'Unit Price';
      fnOnHand: StringGrid1.Cells[Index, 0] := 'On Hand';
      fnReorder: StringGrid1.Cells[Index, 0] := 'Reorder';
    end;
  end;
  Row := 1;
  for Index := VarArrayLowBound(varValues, 1)
      to VarArrayHighBound(varValues, 1) do begin
    LoadRow(Row, FFields, varValues[Index]);
    Inc(Row);
  end;
end;
procedure TfrmMain.Exit1Click(Sender: TObject);
begin
  Close;
end;
procedure TfrmMain.InsertPart1Click(Sender: TObject);
var
  frmPart: TfrmPart;
  varValues: OleVariant;
begin
  frmPart := TfrmPart.Create(nil);
  try
    if frmPart.ShowModal = mrOk then begin
      varValues := VarArrayCreate([0, 4], varVariant);
      varValues[0] := frmPart.PartNumber;
      varValues[1] := frmPart.Description;
      varValues[2] := frmPart.UnitPrice;
      varValues[3] := frmPart.OnHand;
      varValues[4] := frmPart.Reorder;
      PartServer1.AddPart(FFields, varValues);
    end;
  finally
    frmPart.Free;
  end;
end;
procedure TfrmMain.DeletePart1Click(Sender: TObject);
begin
  if StringGrid1.Row > 0 then
    PartServer1.DeletePart(StringGrid1.Cells[0, StringGrid1.Row]);
end;
procedure TfrmMain.EditPart1Click(Sender: TObject);
var
  PartNumber: string;
  frmPart: TfrmPart;
  varValues: OleVariant;
begin
  if StringGrid1.Row > 0 then begin
    PartNumber := StringGrid1.Cells[0, StringGrid1.Row];
    varValues := PartServer1.GetPart(PartNumber, FFields);
    frmPart := TfrmPart.Create(nil);
    try
```

```
      frmPart.ecPartNumber.Text := varValues[0];
      frmPart.ecDescription.Text := varValues[1];
      frmPart.ecUnitPrice.Text := varValues[2];
      frmPart.ecOnHand.Text := varValues[3];
      frmPart.ecReorder.Text := varValues[4];
      if frmPart.ShowModal = mrOk then begin
        varValues := VarArrayCreate([0, 4], varVariant);
        varValues[0] := frmPart.PartNumber;
        varValues[1] := frmPart.Description;
        varValues[2] := frmPart.UnitPrice;
        varValues[3] := frmPart.OnHand;
        varValues[4] := frmPart.Reorder;
        PartServer1.ModifyPart(PartNumber, FFields, varValues);
      end;
    finally
      frmPart.Free;
    end;
  end;
end;
procedure TfrmMain.PartServer1Add(Sender: TObject; varFields,
  varValues: OleVariant);
begin
  // A new part was added - reload the whole grid
  LoadPartGrid;
end;
procedure TfrmMain.PartServer1Delete(Sender: TObject;
  var PartNumber: OleVariant);
begin
  // A part was deleted - reload the whole grid
  LoadPartGrid;
end;
procedure TfrmMain.PartServer1Modify(Sender: TObject;
  var PartNumber: OleVariant; varFields, varValues: OleVariant);
var
  Index: Integer;
  Found: Boolean;
begin
  Found := False;
  Index := 1;
  // If we can find the original part, just reload the row
  while (not Found) and (Index < StringGrid1.RowCount) do begin
    if StringGrid1.Cells[0, Index] = PartNumber then begin
      LoadRow(Index, varFields, varValues);
      Found := True;
    end else
      Inc(Index);
  end;
  // Couldn't find the part, so reload the whole grid
  if not Found then
    LoadPartGrid;
end;
end.
```

El Listado 6.14 contiene el núcleo del cliente administrativo. El método *FormCreate* inicializa un *array* variante con los campos que son de interés del cliente administrativo. Este cliente tiene acceso a toda la información de la base de datos, así que asigno al campo *FFields* una lista de todos los campos.

btnConnectClick es llamado cuando el usuario pulsa el botón *Connect* (ver Figura 6.7). Si el usuario deja en blanco el campo para introducir el nombre del servidor, entonces el cliente trata de conectarse a un servidor local. En caso contrario, el cliente trata de conectarse al servidor situado en el ordenador indicado en ese campo. Si la conexión es exitosa, el código llama a *LoadPartGrid*, que recupera todos los registros del servidor y los muestra en una rejilla.

PartServer1Add, *PartServer1Delete* y *PartServer1Modify* son métodos de la interfaz *IPartsEvents*. Son lanzados desde el servidor siempre que cualquier usuario (y no sólo una única instancia del cliente administrativo) añada, borre o actualice un artículo de la base de datos *Parts*. Las interfaces de eventos se trataron en profundidad en el capítulo 5.

InsertPart1Click solicita al usuario que introduzca información sobre un nuevo artículo, y luego llama al método *AddPart* del servidor para añadir el artículo a la base de datos. Como hemos descrito al comienzo de esta sección, la aplicación cliente envía la lista de campos a añadir, así como sus valores.

DeletePart1Click elimina el artículo actual en la rejilla, y *EditPart1Click* permite que el usuario edite los datos del artículo seleccionado. El método de edición relee el registro desde el servidor para asegurarse de tener los datos más recientes, y luego envía los datos actualizados utilizando los dos *arrays* variantes.

Listado 6.15: PartAdmin - *PartForm.pas*

```
unit PartForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  StdCtrls, ExtCtrls;
type
  TfrmPart = class(TForm)
    pnlClient: TPanel;
    pnlBottom: TPanel;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    ecPartNumber: TEdit;
```

```
    ecDescription: TEdit;
    ecUnitPrice: TEdit;
    ecOnHand: TEdit;
    ecReorder: TEdit;
    btnOK: TButton;
    btnCancel: TButton;
    procedure btnOKClick(Sender: TObject);
  private
    { Private declarations }
    FUnitPrice: Double;
    FReorder: Integer;
    FOnHand: Integer;
    FDescription: string;
    FPartNumber: string;
  public
    { Public declarations }
    property PartNumber: string read FPartNumber;
    property Description: string read FDescription;
    property UnitPrice: Double read FUnitPrice;
    property OnHand: Integer read FOnHand;
    property Reorder: Integer read FReorder;
  end;
implementation
{$R *.DFM}
procedure TfrmPart.btnOKClick(Sender: TObject);
begin
  FPartNumber := ecPartNumber.Text;
  FDescription := ecDescription.Text;
  FUnitPrice := StrToFloat(ecUnitPrice.Text);
  FOnHand := StrToInt(ecOnHand.Text);
  FReorder := StrToInt(ecReorder.Text);
end;
end.
```

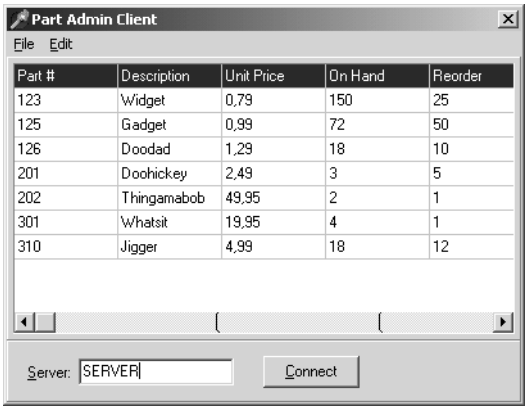


Figura 6.7: La aplicación *PartAdmin* permite a los usuarios añadir, modificar y eliminar artículos del inventario.

Creando el cliente punto de venta

El segundo programa cliente de esta aplicación es un terminal punto de venta. Este se ejecutará desde el mostrador de la tienda que ejecuta la aplicación.

Los Listados 6.16 y 6.17 muestran el código fuente del cliente punto de venta.

Listado 6.16: PartStock - *PartStock.dpr*

```
program PartStock;
uses
  Forms,
  MainForm in 'MainForm.pas' {frmMain};
{$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(TfrmMain, frmMain);
  Application.Run;
end.
```

Listado 6.17: PartStock - *MainForm.pas*

```
unit MainForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, Grids, ComObj, ActiveX, PartSrv_TLB, Menus, ExtCtrls,
  OleServer, StdCtrls;
type
  TfrmMain = class(TForm)
    MainMenu1: TMainMenu;
    pnlClient: TPanel;
    StringGrid1: TStringGrid;
    pnlBottom: TPanel;
    File1: TMenuItem;
    Exit1: TMenuItem;
    Edit1: TMenuItem;
    InsertPart1: TMenuItem;
    EditPart1: TMenuItem;
    DeletePart1: TMenuItem;
    PartServer1: TPartServer;
    ecServer: TEdit;
    btnConnect: TButton;
    Label1: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure Exit1Click(Sender: TObject);
    procedure InsertPart1Click(Sender: TObject);
    procedure DeletePart1Click(Sender: TObject);
    procedure EditPart1Click(Sender: TObject);
    procedure PartServer1Add(Sender: TObject; varFields,
      varValues: OleVariant);
    procedure PartServer1Delete(Sender: TObject;
```

```
      var PartNumber: OleVariant);
    procedure PartServer1Modify(Sender: TObject;
      var PartNumber: OleVariant; varFields, varValues: OleVariant);
    procedure btnConnectClick(Sender: TObject);
  private
    { Private declarations }
    FFields: OleVariant;
    procedure LoadPartGrid;
    procedure LoadRow(Row: Integer; varFields, varValues: OleVariant);
  public
    { Public declarations }
  end;
var
  frmMain: TfrmMain;
implementation
uses PartForm;
{$R *.DFM}
procedure TfrmMain.FormCreate(Sender: TObject);
begin
  FFields := VarArrayCreate([0, 4], varInteger);
  FFields[0] := fnPartNumber;
  FFields[1] := fnDescription;
  FFields[2] := fnUnitPrice;
  FFields[3] := fnOnHand;
  FFields[4] := fnReorder;
end;
procedure TfrmMain.btnConnectClick(Sender: TObject);
begin
  // If no server was entered, we'll use the local machine
  if ecServer.Text  '' then
    PartServer1.RemoteMachineName := ecServer.Text;
  PartServer1.Connect;
  // Load all parts from the database server
  LoadPartGrid;
  // Enable the edit menu
  Edit1.Enabled := True;
end;
procedure TfrmMain.LoadRow(Row: Integer; varFields, varValues:
OleVariant);
var
  ValueIndex: Integer;
  ColIndex: Integer;
begin
  for ValueIndex := VarArrayLowBound(varValues, 1)
      to VarArrayHighBound(varValues, 1) do
    for ColIndex := VarArrayLowBound(FFields, 1)
        to VarArrayHighBound(FFields, 1) do
      if FFields[ColIndex] = varFields[ValueIndex] then
        StringGrid1.Cells[ColIndex, Row] := varValues[ValueIndex];
end;
procedure TfrmMain.LoadPartGrid;
var
```

```
  varValues: OleVariant;
  Index: Integer;
  FieldNumber: Integer;
  Row: Integer;
begin
  varValues := PartServer1.GetAll(FFields);
  StringGrid1.RowCount := VarArrayHighBound(varValues, 1) -
    VarArrayLowBound(varValues, 1) + 2;
  StringGrid1.ColCount := VarArrayHighBound(FFields, 1) -
    VarArrayLowBound(FFields, 1) + 1;
  for Index := VarArrayLowBound(FFields, 1)
      to VarArrayHighBound(FFields, 1) do begin
    FieldNumber := FFields[Index];
    case FieldNumber of
      fnPartNumber: StringGrid1.Cells[Index, 0] := 'Part #';
      fnDescription: StringGrid1.Cells[Index, 0] := 'Description';
      fnUnitPrice: StringGrid1.Cells[Index, 0] := 'Unit Price';
      fnOnHand: StringGrid1.Cells[Index, 0] := 'On Hand';
      fnReorder: StringGrid1.Cells[Index, 0] := 'Reorder';
    end;
  end;
  Row := 1;
  for Index := VarArrayLowBound(varValues, 1)
      to VarArrayHighBound(varValues, 1) do begin
    LoadRow(Row, FFields, varValues[Index]);
    Inc(Row);
  end;
end;
procedure TfrmMain.Exit1Click(Sender: TObject);
begin
  Close;
end;
procedure TfrmMain.InsertPart1Click(Sender: TObject);
var
  frmPart: TfrmPart;
  varValues: OleVariant;
begin
  frmPart := TfrmPart.Create(nil);
  try
    if frmPart.ShowModal = mrOk then begin
      varValues := VarArrayCreate([0, 4], varVariant);
      varValues[0] := frmPart.PartNumber;
      varValues[1] := frmPart.Description;
      varValues[2] := frmPart.UnitPrice;
      varValues[3] := frmPart.OnHand;
      varValues[4] := frmPart.Reorder;
      PartServer1.AddPart(FFields, varValues);
    end;
  finally
    frmPart.Free;
  end;
end;
```

```
procedure TfrmMain.DeletePart1Click(Sender: TObject);
begin
  if StringGrid1.Row > 0 then
    PartServer1.DeletePart(StringGrid1.Cells[0, StringGrid1.Row]);
end;

procedure TfrmMain.EditPart1Click(Sender: TObject);
var
  PartNumber: string;
  frmPart: TfrmPart;
  varValues: OleVariant;
begin
  if StringGrid1.Row > 0 then begin
    PartNumber := StringGrid1.Cells[0, StringGrid1.Row];
    varValues := PartServer1.GetPart(PartNumber, FFields);
    frmPart := TfrmPart.Create(nil);
    try
      frmPart.ecPartNumber.Text := varValues[0];
      frmPart.ecDescription.Text := varValues[1];
      frmPart.ecUnitPrice.Text := varValues[2];
      frmPart.ecOnHand.Text := varValues[3];
      frmPart.ecReorder.Text := varValues[4];
      if frmPart.ShowModal = mrOk then begin
        varValues := VarArrayCreate([0, 4], varVariant);
        varValues[0] := frmPart.PartNumber;
        varValues[1] := frmPart.Description;
        varValues[2] := frmPart.UnitPrice;
        varValues[3] := frmPart.OnHand;
        varValues[4] := frmPart.Reorder;
        PartServer1.ModifyPart(PartNumber, FFields, varValues);
      end;
    finally
      frmPart.Free;
    end;
  end;
end;
procedure TfrmMain.PartServer1Add(Sender: TObject; varFields,
  varValues: OleVariant);
begin
  // A new part was added - reload the whole grid
  LoadPartGrid;
end;
procedure TfrmMain.PartServer1Delete(Sender: TObject;
  var PartNumber: OleVariant);
begin
  // A part was deleted - reload the whole grid
  LoadPartGrid;
end;
procedure TfrmMain.PartServer1Modify(Sender: TObject;
  var PartNumber: OleVariant; varFields, varValues: OleVariant);
var
  Index: Integer;
```

```
  Found: Boolean;
begin
  Found := False;
  Index := 1;
  // If we can find the original part, just reload the row
  while (not Found) and (Index < StringGrid1.RowCount) do begin
    if StringGrid1.Cells[0, Index] = PartNumber then begin
      LoadRow(Index, varFields, varValues);
      Found := True;
    end else
      Inc(Index);
  end;
  // Couldn't find the part, so reload the whole grid
  if not Found then
    LoadPartGrid;
end;
end.
```

La mayor parte del código de esta unidad es idéntico al de la unidad del cliente administrativo. La diferencia principal en este programa es que el cliente punto de venta no tiene acceso al campo *Reorder* de la base de datos. Ya que el *array FFields* no incluye la constante *fnReorder*, esa cantidad nunca será enviada del servidor a este cliente.

El método *UsePart1Click* se ejecuta cuando el usuario pulsa el botón *Use* del programa. Aquí se pregunta al usuario cuántas unidades del artículo seleccionado en la rejilla se sacarán del inventario, y se llama al método *UsePart* del servidor para actualizar la base de datos. En este ejemplo, si no existe el *stock* suficiente para satisfacer la petición, una excepción es lanzada en el servidor alertando del error. He hecho esto para mostrarle cómo la excepción es presentada en la máquina cliente, aún cuando haya sido lanzada en una máquina remota.

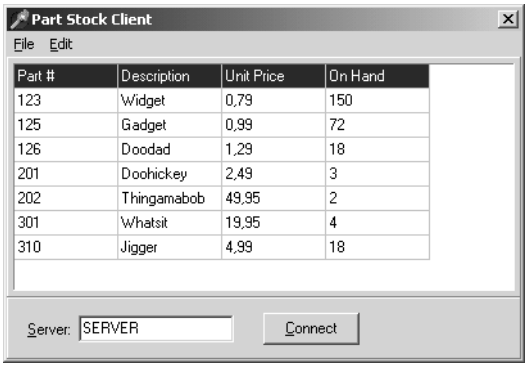


Figura 6.8: La aplicación *PartStock* ha sido pensada para ejecutarla en el mostrador de la tienda.

Resumen

En este capítulo hemos visto cómo crear servidores y clientes DCOM. Como ha visto, un servidor DCOM no se diferencia de un servidor COM tradicional - sólo que reside en una máquina remota.

La parte más difícil de la instalación de un servidor DCOM es configurar correctamente la máquina que funcionará como servidor. Siempre que sea posible, deberá utilizar Windows NT/2000 (servidor de dominio) como servidor DCOM.

Una vez que el servidor DCOM está operativo, la aplicación cliente simplemente llama al método *CreateRemote* de la co-clase, en vez de llamar a *Create*.

En la segunda mitad del capítulo, hemos mostrado cómo se puede usar DCOM para implementar bases de datos remotas. Hemos desarrollado una aplicación de ejemplo que podría utilizarse como programa de inventario en un almacén de piezas de repuesto para coches, por ejemplo.

En el próximo capítulo introduciremos los ficheros de almacenamiento estructurado.

Resumen

En este capítulo hemos visto cómo crear servidores y clientes DCOM. Como ha visto, un servidor DCOM no se diferencia de un servidor COM tradicional - sólo que reside en una máquina remota.

La parte más difícil de la instalación de un servidor DCOM es configurar correctamente la máquina que funcionará como servidor. Siempre que sea posible, deberá utilizar Windows NT/2000 (servidor de dominio) como servidor DCOM.

Una vez que el servidor DCOM está operativo, la aplicación cliente simplemente llama al método *CreateRemote* de la co-clase, en vez de llamar a *Create*.

En la segunda mitad del capítulo, hemos mostrado cómo se puede usar DCOM para implementar bases de datos remotas. Hemos desarrollado una aplicación de ejemplo que podría utilizarse como programa de inventario en un almacén de piezas de repuesto para coches, por ejemplo.

En el próximo capítulo introduciremos los ficheros de almacenamiento estructurado.

7 Almacenamiento Estructurado

En este capítulo presentaremos el almacenamiento estructurado. Como las interfaces, el almacenamiento estructurado es una tecnología que tiene simultáneamente aplicaciones de propósito general y aplicaciones predefinidas por COM. Por esta razón, este capítulo le indica cómo puede aprovechar las ventajas del almacenamiento estructurado en sus aplicaciones. En el Capítulo 8, "Almacenamiento Estructurado y OLE", examinaremos dos aplicaciones predefinidas por COM del almacenamiento estructurado: los conjuntos de propiedades y los documentos compuestos.

Definiendo el almacenamiento estructurado

Los ficheros de almacenamiento estructurado (también conocidos como *DocFiles*) proporcionan un mecanismo potente y estructurado para almacenar de forma persistente información en un fichero. En pocas palabras, el almacenamiento estructurado permite implementar un sistema completo de ficheros dentro de un único fichero.

Considere una aplicación que necesita almacenar cierta cantidad de datos de diferentes tipos en sus ficheros de datos. Por ejemplo, digamos que está Ud. desarrollando una aplicación para controlar costes de automóviles. Suponga que la aplicación controla la siguiente información:

- Costes de combustible
- Cambios de aceite
- Registros de servicios

Para la gasolina, la aplicación controla la fecha, la cantidad de galones comprados y la cantidad de kilómetros que muestra el cuentakilómetros en el momento del aprovisionamiento. Para los cambios de aceite, se controla la fecha y el kilometraje. Y en el caso de los registros de servicios, se controla la fecha, la descripción del servicio realizado y el coste del servicio.

Claramente, estas tres partes del programa usarán diferentes estructuras de datos para almacenar la información correspondiente. Tradicionalmente, estos datos podrían ser almacenados de diferentes formas:

- En una base de datos, utilizando tres tablas con diferentes estructuras. El problema de este enfoque es que requiere de la sobrecarga que representa un motor de base de datos. Una aplicación pequeña, monousuario podría no necesitar (o desear) esa sobrecarga adicional.
- En tres ficheros de datos separados llamados *GAS.DAT*, *OIL.DAT* y *SERVICE.DAT*. El problema de esta aproximación es que genera numerosos ficheros de datos de los cuales el usuario tendrá que preocuparse.
- En un único fichero de datos, con las listas de datos serializadas una tras otra. El problema de este enfoque es que la aplicación deberá leer el fichero entero para acceder a la información almacenada en el final del fichero.

El almacenamiento estructurado añade una nueva alternativa al panorama. Mediante el almacenamiento estructurado, es posible combinar la segunda y tercera alternativas anteriores, evitando los problemas asociados a cada una.

Conceptualmente, un fichero de almacenamiento estructurado se asemeja a la estructura de directorios de un disco duro. Continuando con el ejemplo hipotético de los coches, eche un vistazo a la Figura 7.1. Esta muestra la estructura de un almacenamiento estructurado que podría ser utilizado para almacenar esa información.

Figura 7.1: Un fichero de almacenamiento estructurado se asemeja a una estructura de directorios.

En los términos del almacenamiento estructurado, el equivalente de un directorio es un almacén (*storage*), y el equivalente de un fichero es un flujo (*stream*). En la Figura 7.1 pueden reconocerse un almacén (raíz) y cuatro flujos en este almacenamiento estructurado.

Esta estructura ofrece una ventaja adicional que no debe ser subestimada. Puede utilizar un flujo separado, como yo he hecho en la Figura 7.1, para almacenar la información de versión asociada con el fichero. Cuando una nueva versión de la aplicación sea liberada (quizás para permitirle controlar múltiples vehículos), podrá

modificar la estructura del fichero para que sea similar a la que se muestra en la Figura 7.2. Su aplicación podrá leer el flujo asociado a la versión del almacenamiento estructurado sin tener que acceder a ningún otro dato. Si el número de versión no coincide con la versión del programa, podrá actualizar el fichero en el momento, o mostrar un mensaje de error y terminar la ejecución.

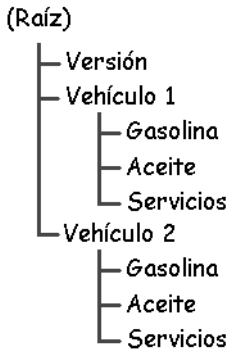


Figura 7.2: Estructura modificada del fichero estructurado.

Los ficheros de almacenamiento estructurado pueden ser fácilmente actualizados cuando se produce una nueva versión de la aplicación. Ahora que conoce qué es el almacenamiento estructurado, examinaremos qué se necesita para usar los ficheros de almacenamiento estructurado en las aplicaciones Delphi.

Programación con almacenamiento estructurado

Esta sección se centrará en las tareas comunes de programación asociadas a la creación, apertura, lectura y escritura de datos en ficheros de almacenamiento estructurado desde aplicaciones Delphi.

Creando un fichero de almacenamiento estructurado

La función que se utiliza para crear un fichero de almacenamiento estructurado se llama *StgCreateDocFile*, y está definida en *ActiveX.pas* como:

```
function StgCreateDocFile(pwcsName: POleStr; grfMode: Longint;
  Reserved: Longint; out stgOpen: IStorage): HResult; stdcall;
```

PwcsName es el nombre del fichero a crear. *GrfMode* especifica el modo de acceso con el que se abrirá el fichero, y puede tomar como valor una combinación de las constantes listadas en la Tabla 7.1. *Reserved* es un parámetro reservado por Microsoft para su uso futuro – debe asignársele cero. Por último, *stgOpen* devuelve una referencia a una interfaz *IStorage*, que podrá utilizarse en lo adelante para manipular el fichero.

Nota

StgOpen realmente se refiere al almacén raíz del almacenamiento estructurado. No se refiere al almacenamiento estructurado como tal. Las diferencias entre el almacenamiento estructurado y la raíz del almacenamiento es análoga a la diferencia entre un disco duro y el directorio raíz de éste.

Tabla 7.1: Valores válidos de grfMode para la función StgCreateDocFile

Valor	Descripción
STGM_DIRECT	Cada modificación al fichero es grabada tan pronto como esta se produce.
STGM_TRANSACTED	Las modificaciones son buferizadas en memoria, y sólo son grabadas al fichero cuando se emite explícitamente un comando de confirmación (Commit). Las modificaciones pueden ser deshechas mediante un comando de retroceso (Revert).
STGM_SIMPLE	Soporta una forma simplificada de almacenamiento estructurado. Se refuerza un conjunto de limitaciones, cuyo cumplimiento permite que las operaciones sobre el fichero se ejecuten mucho más rápidamente. Más adelante comentaremos más sobre el almacenamiento estructurado simple.
STGM_READ	Abre el fichero con acceso de lectura.
STGM_WRITE	Abre el fichero con acceso de escritura.
STGM_READWRITE	Abre el fichero con accesos de lectura y escritura.
STGM_SHARE_DENY_NONE	No deniega a otros la posibilidad de abrir el fichero.
STGM_SHARE_DENY_READ	Deniega a otros la posibilidad de abrir el fichero con acceso de lectura.
STGM_SHARE_DENY_WRITE	Deniega a otros la posibilidad de abrir el fichero con acceso de escritura.
STGM_SHARE_EXCLUSIVE	Deniega a otros la posibilidad de abrir el fichero con acceso de lectura o de escritura.
STGM_PRIORITY	Impide que otros realicen cambios al fichero mientras esté abierto. Esta opción debe combinarse con STGM_DIRECT y STGM_READ.

Valor	Descripción
STGM_DELETEONRELEASE	El fichero es borrado automáticamente cuando la interfaz es liberada. Se utiliza para crear ficheros temporales.
STGM_CREATE	Crea un nuevo fichero. Si ya existiera un fichero con el mismo nombre, éste será borrado.
STGM_CONVERT	Crea un nuevo fichero. Si ya existiera un fichero con el mismo nombre, se preservarán los datos existentes en el flujo CONTENTS del fichero.
STGM_FAILIFTHERE	Devuelve el código de error STG_E_FILEALREADYEXISTS si ya existe un fichero con el mismo nombre.
STGM_NOSCRATCH	Cuando se especifica STGM_TRANSACTED , usualmente Windows95 crea un fichero de trabajo para almacenar los cambios hasta que se ejecute el comando Commit. STGM_NOSCRATCH permite a Windows 95 utilizar el área no usada del fichero de almacenamiento estructurado como espacio de trabajo, permitiendo un uso más eficiente de memoria.
STGM_NOSNAPSHOT	Se utiliza solamente en combinación con STGM_TRANSACTED y sin STGM_SHARE_EXCLUSIVE ni STGM_SHARE_DENY_WRITE. Esta opción impide que Windows realice una copia instantánea del fichero. Los cambios al fichero se graban al final de éste. Si se especifica STGM_NOSNAPSHOT, todos los usuarios del fichero deben especificar esta opción.

El siguiente fragmento de código creará un fichero de almacenamiento estructurado llamado *MyFile.SS*.

```
var
 SsgRoot: IStorage;
begin
  OleCheck(StgCreateDocFile('C:\MyFile.SS',
    STGM_CREATE or STGM_READWRITE or STGM_SHARE_EXCLUSIVE, 0, stgRoot));
...
end;
```

Nota

Almacenamiento Estructurado Simple

STGM_SIMPLE sólo es utilizable cuando se combina con STGM_DIRECT, STGM_CREATE, STGM_READWRITE y STGM_SHARE_EXCLUSIVE. Este modo obliga a la aplicación a cumplir los siguientes requisitos:

- No se puede utilizar sub-almacenes. Sólo se pueden crear flujos en el almacén raíz.
- Después que un flujo haya sido cerrado, no podemos volver a abrirlo.
- Los almacenes y flujos no pueden ser serializados.
- Todos los flujos deben tener al menos 4096 bytes de longitud. Cualquier flujo que contenga menos de 4096 bytes será extendido automáticamente hasta 4096 bytes.
- Sólo se ofrece soporte a un subconjunto de métodos de IStorage e IStream. En particular, sólo los métodos QueryInterface, AddRef, Release, CreateStream, Commit y SetClass son soportados por IStorage. Los métodos soportados por IStream incluyen a QueryInterface, AddRef, Release, Seek y SetSize. También está soportados los métodos Read y Write de la interfaz ISequentialStream. Cualquier llamada a otro método devolverá el código de error STGM_E_INVALID_FUNCTION.

Abriendo un fichero de almacenamiento estructurado existente

La función que se utiliza para abrir un fichero de almacenamiento estructurado ya existente es *StgOpenStorage*, definida de la siguiente forma:

```
function StgOpenStorage(pwcsName: POleStr; stgPriority: IStorage;
  grfMode: Longint; snbExclude: TSNB; reserved: Longint;
  out stgOpen: IStorage): HResult; stdcall;
```

Nuevamente, *pwcsName* es el nombre del fichero de almacenamiento estructurado a abrir. *StgPriority* es normalmente **nil**. Si no fuera **nil**, apuntaría a una instancia anterior del almacén raíz abierto. No hablaremos más sobre *stgPriority* en este libro. *GrfMode* especifica el modo de acceso con el que se abre el fichero, y puede tomar como valor una combinación de las constantes presentadas en la Tabla 7.1. *SnbExclude* también es normalmente **nil**. En caso contrario, apuntaría a una estructura *SNB* que indica qué elementos del almacenamiento deben ser excluidos al abrir el fichero. No discutiremos más este parámetro en el libro. *Reserved* está reservado por Microsoft, y debe asignársele cero. Por último, *StgOpen* recibirá una referencia a la interfaz *IStorage* asociada al fichero de almacenamiento estructurado abierto.

El siguiente fragmento de código muestra cómo abrir un fichero de almacenamiento estructurado existente:

```
var
  StgRoot: IStorage;
begin
  OleCheck(StgOpenStorage('C:\MyFile.SS', nil,
    STGM_READWRITE or STGM_SHARE_EXCLUSIVE, nil, 0, stgRoot));
...
end;
```

Trabajando con flujos

Después de haber obtenido una referencia al almacén raíz de un fichero de almacenamiento estructurado, se abre la posibilidad de crear un flujo en él. Recuerde que un almacén es conceptualmente equivalente a un directorio, y que un flujo es el equivalente aproximado de un fichero. Por tanto, deberá crear un flujo antes de que pueda escribir datos en él. Las siguientes secciones discuten cómo crear, abrir, leer y escribir en un flujo.

Creando un flujo

Para crear un nuevo flujo utilizaremos *IStorage.CreateStream*. *CreateStream* se define mediante el siguiente código:

```
function CreateStream(pwcsName: POleStr; grfMode: Longint,
  reserved1, reserved2: Longint; out stm: IStream): HResult; stdcall;
```

PwcsName es el nombre del flujo a crear. *GrfMode* es el modo de acceso al flujo. *Reserved1* y *reserved2* están reservados y se le debe asignar cero. *Stm* recibirá una referencia a la interfaz *IStream* asociada al nuevo flujo.

El siguiente fragmento de código muestra cómo crear un flujo:

```
var
  stgRoot: IStorage;
  stmData: IStream;
begin
  // Crear un nuevo fichero de almacenamiento estructurado
  OleCheck(StgCreateDocFile('C:\MyFile.SS',
    STGM_CREATE or STGM_READWRITE or STGM_SHARE_EXCLUSIVE, 0, stgRoot));
  // Crear un flujo en el almac□n ra□z
  OleCheck(stgRoot.CreateStream('MyData',
    STGM_CREATE or STGM_READWRITE or STGM_SHARE_EXCLUSIVE, 0, 0,
stmData));
  // ...
end;
```

Escribiendo en un flujo

Ahora que ya ha creado un flujo, deseará grabar datos en él. El método estándar para escribir en un flujo es a través del método *IStream.Write*:

```
var
  stgRoot: IStorage;
  stmData: IStream;
  MyIntValue: Integer;
  BytesWritten: LongInt;
begin
  // crear un nuevo fichero estructurado
  OleCheck(StgCreateDocFile('C:\MyFile.SS',
    STGM_CREATE or STGM_READWRITE or STGM_SHARE_EXCLUSIVE, 0, stgRoot));
  // crear un flujo en el almac□n
  OleCheck(StgRoot.CreateStream('MyData',
```

```
    STGM_CREATE or STGM_READWRITE or STGM_SHARE_EXCLUSIVE, 0, 0,
stmData));
  MyIntValue := 100;
  // grabar al flujo
  OleCheck(stmData.Write(@MyintValue, sizeof(Integer),  @BytesWritten));
  // ...
end;
```

Delphi proporciona una clase llamada *TOleStream* que hace esta tarea aún más sencilla. *TOleStream* implementa un envoltorio alrededor de la interfaz *IStream*. De este modo, se puede escribir a un flujo de almacenamiento estructurado usando los mismos métodos que se utilizan cuando se trabaja con cualquier flujo estándar de Delphi.

Por ejemplo, he aquí un fragmento de código que crea un fichero de almacenamiento estructurado, crea un flujo dentro de aquél, y graba un mapa de bits al flujo usando *TOleStream*.

```
var
  stgRoot: IStorage;
  stmData: IStream;
  OS: TOleStream;
begin
  // crear un Nuevo fichero estructurado
  OleCheck(StgCreateDocFile('C:\MyFile.SS',
    STGM_CREATE or STGM_READWRITE or STGM_SHARE_EXCLUSIVE, 0, stgRoot));
  // crear un flujo en el almac□n
  OleCheck(StgRoot.CreateStream('MyData',
    STGM_CREATE or STGM_READWRITE or STGM_SHARE_EXCLUSIVE, 0, 0,
    stmData));
  // encapsular en flujo con un TOleStream
  OS := TOleStream.Create(stmData);
  try
    Image1.Picture.Bitmap.SaveToStream(OS);
  finally
    OS.Free;
  end;
  // ...
end;
```

Abriendo un flujo

Abrir un flujo es similar a crearlo, con la diferencia de que se utiliza *IStorage.OpenStream*, definido como:

```
function OpenStream(pwcsName: POleStr; reserved1: Pointer; grfMode:
Longint;
  reserved2: Longint; out stm: IStream): HResult; stdcall;
```

PwcsName es el nombre del flujo que se desea abrir. *Reserved1* es un parámetro reservado y debe asignársele cero. *GrfMode* es el modo de acceso que se utilizará para abrir el flujo. *Reserved2* es también reservado, y debe ser cero. *Stm* recibirá una referencia a la interfaz *IStream* asociada al flujo abierto.

He aquí un fragmento de código que muestra cómo abrir un fichero de almacenamiento estructurado existente y un flujo en el mismo:

```
var
  stgRoot: IStorage;
  stmData: IStream;
begin
  OleCheck(StgOpenStorage('C:\MyFile.SS', nil,
    STGM_READWRITE or STGM_SHARE_EXCLUSIVE, nil, 0, stgRoot));
  OleCheck(StgRoot.OpenStream('MyData', 0,
    STGM_READWRITE or STGM_SHARE_EXCLUSIVE, 0, stmData));
  // ...
end;
```

Leyendo de un flujo

Leer desde un flujo implica utilizar el método *IStream.Read*, como muestra el siguiente código:

```
var
  stgRoot: IStorage;
  stmData: IStream;
  MyIntValue: Integer;
  BytesRead: LongInt;
begin
  OleCheck(StgOpenStorage('C:\MyFile.SS', nil,
    STGM_READWRITE or STGM_SHARE_EXCLUSIVE, nil, 0, stgRoot));
  OleCheck(StgRoot.OpenStream('MyData', nil,
    STGM_READWRITE or STGM_SHARE_EXCLUSIVE, 0, stmData));
  OleCheck(stmData.Read(@MyIntValue, sizeof(Integer), @BytesRead));
  // .....
end;
```

De manera similar a como se escribe a un flujo, normalmente se debe utilizar *TOleStream* para leer de un flujo. El siguiente fragmento de código muestra cómo podemos leer una imagen de un flujo usando *TOleStream*:

```
var
  stgRoot: IStorage;
  stmData: IStream;
  OS: TOleStream;
begin
  OleCheck(StgOpenStorage('C:\MyFile.SS', nil,
    STGM_READWRITE or STGM_SHARE_EXCLUSIVE, nil, 0, stgRoot));
  OleCheck(stgRoot.OpenStream('MyData', nil,
    STGM_READWRITE or STGM_SHARE_EXCLUSIVE, 0, stmData));
  OS := TOleStream.Create(stmData);
  try
    Image1.Picture.Bitmap.LoadFromStream(OS);
  finally
    OS.Free;
  end;
  // .....
end;
```

Creando y utilizando almacenes adicionales

Ud. no está limitado a tener solamente un almacén en sus ficheros estructurados, aunque normalmente este será el caso. Para crear almacenes adicionales, utilice el método *CreateStorage* de *IStorage*, definido de la siguiente forma:

```
function CreateStorage(pwcsName: POleStr; grfMode: Longint;
  dwStgFmt: Longint; reserved2: Longint; out stg: IStorage): HResult;
stdcall;
```

PwcsName es por supuesto el nombre del almacén a crear. *GrfMode* nuevamente especifica el modo de acceso a utilizar al crear el almacén. *DwStgFmt* es normalmente puesto a cero. *Reserved2* es parámetro reservado y debe asignársele cero. *Stg* recibirá una referencia a la interfaz *IStorage* asociada al nuevo almacén.

Para abrir un almacén ya existente (diferente del raiz), utilice la función *IStorage.OpenStorage*:

```
function OpenStorage(pwcsName: POleStr; const stgPriority: IStorage;
  grfMode: Longint; snbExclude: TSNB;
  reserved: Longint; out stg: IStorage): HResult; stdcall;
```

PwcsName es el nombre del almacén a abrir. *StgPriority* debe ser **nil**, o la llamada a *OpenStorage* devolverá *STG_E_INVALIDPARAMETER*. *GrfMode* especifica el modo de acceso a utilizar sobre el almacén abierto. *SnbExclude* también debe ser **nil**. *Reserved* es parámetro reservado y debe asignársele cero. Por último, *stg* recibirá una referencia a la interfaz *IStorage* asociada al almacén abierto.

Después de crear o abrir un almacén adicional, puede usarlo exactamente como si se tratase del almacén raíz. En otras palabras, puede crear y acceder a flujos en el almacén raíz, y además podrá crear sub-almacenes bajo éste.

Iterando a través de un fichero de almacenamiento estructurado

Cuando utilice ficheros de almacenamiento estructurado en sus aplicaciones, obviamente conocerá el diseño del fichero. Sin embargo, en ocasiones puede ser útil visualizar la estructura de un fichero estructurado que haya sido creado por otra aplicación.

Debido a que los ficheros de almacenamiento estructurado se autodocumentan, es posible enumerar los almacenes y flujos contenidos en el fichero. Para esto se debe utilizar el método *IStorage.EnumElements*, definido como:

```
function EnumElements(reserved1: Longint; reserved2: Pointer;
  reserved3: Longint; out enm: IEnumStatStg): HResult; stdcall;
```

Reserved1, *reserved2*, y *reserved3* son parámetros reservados, y deben ser cero. *Enm* recibirá una referencia a la interfaz *IEnumStatStg*. *IEnumStatStg* brinda los métodos necesarios para enumerar los elementos individuales del fichero de

almacenamiento estructurado. El único método en el que estamos interesados es *Next*, definido de la siguiente forma:

```
function Next(celt: Longint; out elt;
  pceltFetched: PLongint): HResult; stdcall;
```

Celt especifica el número de elementos a recuperar de una vez. *Elt* apunta a un *array* que recibirá los elementos, y *pceltFetched* es un puntero a una variable que recibirá la cantidad de elementos realmente recuperados. Puede pasar **nil** en este último parámetro si la cantidad de elementos solicitados es 1.

Nota

Incluso si solicita más de un elemento a la vez, la implementación de IEnumStatStg no está obligada a devolver más de un elemento a la vez. La mayoría de los enumeradores COM (incluyendo IEnumStatStg) sólo devuelven un elemento cada vez, independientemente de cuántos solicite. Asegúrese de verificar siempre el valor de pceltFetched para saber cuántos elementos fueron realmente devueltos.

pceltFetched apunta a un *array* de registros *TStatStg*. *TStatStg* se define de la siguiente forma:

```
type
  tagSTATSTG = record
    pwcsName: POleStr;
    dwType: Longint;
    cbSize: Largeint;
    mtime: TFileTime;
    ctime: TFileTime;
    atime: TFileTime;
    grfMode: Longint;
    grfLocksSupported: Longint;
    clsid: TCLSID;
    grfStateBits: Longint;
    reserved: Longint;
  end;
  TStatStg = tagSTATSTG;
```

Normalmente, estará más interesado en el campo *pwcsName* del registro. Este campo contiene el nombre del elemento en cuestión.

Combinando los dos métodos mostrados, podemos crear un procedimiento que haga algo con cada uno de los elementos del fichero de almacenamiento estructurado, como sigue:

```
procedure IterateSS(stg: IStorage);
var
  EnumStatStg: IEnumStatStg;
  StatStg: TStatStg;
begin
```

```
    OleCheck(stg.EnumElements(0, nil, 0, EnumStatStg));
    while EnumStatStg.Next(1, StatStg, nil)= S_OK do begin
     // hacer algo aquí con StatStg...
    end;
  end;
```

Note que puede llamar a *IterateSS* pasándole una referencia a cualquier almacén, incluyendo el raíz.

Comprimiendo un fichero de almacenamiento estructurado

Un fichero de almacenamiento estructurado es en muchas formas similar a la estructura de directorios de su disco duro, y también padece uno de los problemas comunes de estos – la fragmentación. También tiene un problema que no presentan los discos - no puede recuperar automáticamente el espacio que ha dejado de utilizarse. Cuando se elimina un fichero del disco duro, el espacio ocupado por el fichero es inmediatamente liberado y queda disponible para ser usado por otros ficheros. Cuando borramos un flujo de un fichero de almacenamiento estructurado, el espacio está disponible para ser utilizado por otros flujos, pero el fichero de almacenamiento estructurado no se reduce para recuperar el "hueco" dejado por el flujo eliminado.

Desafortunadamente, no existe un método sencillo para optimizar un fichero de almacenamiento estructurado. Sin embargo, eso no significa que no sea posible hacerlo. Para defragmentar los almacenes y flujos de un fichero de almacenamiento estructurado, así como recuperar todo el espacio posible, se debe hacer lo siguiente:

1. Abrir el fichero en modo exclusivo.
2. Crear un nuevo fichero temporal.
3. Copiar todos los almacenes y flujos del fichero original al fichero temporal.
4. Cerrar el fichero temporal.
5. Cerrar el fichero original.
6. Borrar el fichero original.
7. Renombrar el fichero temporal, dándole el mismo nombre del fichero original.

El siguiente procedimiento muestra cómo lograr esto:

```
procedure CompressStructuredStorageFile(AFileName: WideString);
var
  stgOriginal: IStorage;
  stgTemp: IStorage;
  TempFileName: WideString;
begin
```

```
    // Paso 1 – Abrir fichero original
    OleCheck(StgOpenStorage(PWideChar(AFileName), nil,
      STGM_READ or STGM_SHARE_EXCLUSIVE, nil, 0, stgOriginal));
    // Paso 2 – Crear nuevo fichero temporal
    TempFileName := ChangeFileExt(AFileName, '.$$$');
    OleCheck(StgCreateDocFile(PWideChar(TempFileName),
      STGM_WRITE or STGM_SHARE_EXCLUSIVE, 0, stgTemp));
    // Paso 3 – Copiar fichero antiguo al nuevo
    stgOriginal.CopyTo(0, nil, nil, stgTemp);
    // Paso 4 – Cerrar fichero temporal
    stgTemp := nil;
    // Paso 5 – Cerrar fichero original
    stgOriginal := nil;
    // Paso 6 – Borrar fichero original
    DeleteFile(AfileName);
    // Paso 7 – Renombrar fichero temporal
    RenameFile(TempFileName, AFileName);
  end;
```

Ya hemos examinado todos los métodos que se utilizan en el fragmento de código anterior, excepto *CopyTo*. *IStorage.CopyTo* se utiliza para copiar el contenido de un almacén a otro almacén, y se define de la siguiente forma:

```
function CopyTo(ciidExclude: Longint; rgiidExclude: PIID;
  snbExclude: TSNB; const stgDest: IStorage): HResult; stdcall;
```

CiidExclude contiene la cantidad de elementos especificados por *rgiidExclude*. Si *rgiidExclude* es **nil**, entonces *ciidExclude* es ignorado. *SnbExclude* también puede ser **nil**. Juntos, estos tres parámetros nos permiten especificar un conjunto de almacenes y/o flujos que no serán copiados por *CopyTo*. Más bien se trata de una característica avanzada, y rara vez la utilizará, por lo que no discutiremos estos parámetros en este capítulo. *StgDest* especifica la interfaz *IStorage* asociada al objeto de almacenamiento de destino. El uso más común de este método, mostrado en el ejemplo anterior, copia un almacén entero, incluyendo sub-almacenes y flujos, a un almacén de destino.

Ejemplo: Uso de ficheros de almacenamiento estructurado

Ahora que ya conoce las bases de utilización del almacenamiento estructurado, vamos a crear una aplicación que utilice el almacenamiento estructurado como formato nativo de ficheros. Esta aplicación es un tanto larga en comparación con otros ejemplos de este libro, pero no es difícil de comprender.

Al inicio del capítulo hemos mencionado una hipotética aplicación relacionada con automóviles. Hagamos de esta aplicación una realidad. Por consideraciones de espacio, he implementado únicamente las partes relacionadas con el control de la gasolina y el aceite, y sólo permitimos el seguimiento de un único vehículo simple.

Podrá extender el código mostrado aquí para gestionar los servicios y cualquier otra cosa que desee.

La aplicación *CarDemo* almacena todos los datos en un único fichero de almacenamiento estructurado llamado *CARS.CAR*. *CARS.CAR* contiene un único almacén (el almacén raíz) y tres flujos: uno para el carburante, otro para el aceite, y un tercero para almacenar información general sobre el coche, como una descripción, el modelo y año.

La aplicación está dividida en siete ficheros fuente, que se presentan en los Listados del 7.1 al 7.7. La mayor parte del código se explica por sí misma, aunque he utilizado un truco para facilitar la serialización de los datos relacionados con la gasolina y el aceite. Comentaré cada fichero por su turno.

El Listado 7.2 contiene el código fuente del formulario principal de la aplicación.

FormCreate intenta abrir el fichero de almacenamiento estructurado *CARS.CAR*. Si el intento no es exitoso, este evento crea un fichero de almacenamiento estructurado con ese nombre. Un programa más robusto deberá examinar el código de error devuelto por *StgOpenStorage* y crear el fichero de datos únicamente si éste no existe. *FormCreate* dejará el fichero abierto durante el resto de la ejecución del programa.

Obviaré el resto de *FormCreate* por el momento, y en lugar de ello me centraré en *FormDestroy*. *FormDestroy* abre el flujo *Info* para escritura. Recuerde que el nombre del coche y el modelo serán guardados en este flujo. Aquí he utilizado una clase *TWriter* para grabar el nombre y el modelo. Utilizar un método tan específico de Delphi como *TWriter* implica que no seremos capaces de leer fácilmente esta información desde otro lenguaje de programación, como Visual C++ o Visual Basic, pero para este ejemplo eso no me preocupa mucho.

Nota

El uso de TWriter puede parecer excesivo en esta situación, ya que sólo se van a grabar una cadena y un entero. Sin embargo, si se tratase de una aplicación real, podría ser fácilmente extendida para incluir el precio de compra del coche, la fecha de compra, la tasa de interés, y muchos otros datos. TWriter incluye numerosos métodos que hacen más sencillo el trabajo de escribir diferentes tipos de datos.

Volviendo a FormCreate, una vez que el fichero estructurado es abierto, el programa intenta abrir el flujo Info. En caso de éxito, se utiliza un componente TReader para leer la información que es guardada por el componente TWriter en FormDestroy. Si el flujo no puede ser abierto se crea uno nuevo, como se muestra en el Listado 7.2.

Listado 7.1: CarDemo - *CarDemo.dpr*

```
program CarDemo;
uses
  Forms,
  MainForm in 'MainForm.pas' {frmMain},
  GasForm in 'GasForm.pas' {frmGas},
  OilForm in 'OilForm.pas' {frmOil},
  GasItemForm in 'GasItemForm.pas' {frmGasItem},
  CarInfoForm in 'CarInfoForm.pas' {frmCarInfo},
  OilItemForm in 'OilItemForm.pas' {frmOilItem};
{$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(TfrmMain, frmMain);
  Application.Run;
end.
```

Listado 7.2: CarDemo – *MainForm.pas*

```
unit MainForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  Menus, ActiveX, AxCtrls, ComObj;
type
  TfrmMain = class(TForm)
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    Exit1: TMenuItem;
    N1: TMenuItem;
    Gas1: TMenuItem;
    Oil1: TMenuItem;
    Edit1: TMenuItem;
    CarInformation1: TMenuItem;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure Gas1Click(Sender: TObject);
    procedure Oil1Click(Sender: TObject);
    procedure CarInformation1Click(Sender: TObject);
  private
    FRootStorage: IStorage;
    FCarName: string;
    FYear: Integer;
    function ActivateForm(ACaption: string): Boolean;
    { Private declarations }
  public
    { Public declarations }
    property RootStorage: IStorage read FRootStorage;
  end;
var
  frmMain: TfrmMain;
```

```
implementation
uses GasForm, OilForm, CarInfoForm;
{$R *.DFM}
procedure TfrmMain.FormCreate(Sender: TObject);
var
  stm: IStream;
  OS: TOleStream;
  Reader: TReader;
begin
  if not SUCCEEDED(StgOpenStorage('CARS.CAR', nil,
      STGM_READWRITE or STGM_SHARE_EXCLUSIVE, nil, 0, FRootStorage)) then
    OleCheck(StgCreateDocFile('CARS.CAR',
      STGM_CREATE or STGM_READWRITE or STGM_SHARE_EXCLUSIVE,
      0, FRootStorage));
  if SUCCEEDED(FRootStorage.OpenStream('Info', nil,
    STGM_READ or STGM_SHARE_EXCLUSIVE, 0, stm)) then begin
    OS := TOleStream.Create(stm);
    try
      Reader := TReader.Create(OS, 1024);
      try
        FCarName := Reader.ReadString;
        FYear := Reader.ReadInteger;
      finally
        Reader.Free;
      end;
    finally
      OS.Free;
    end;
  end else begin
    OleCheck(FRootStorage.CreateStream('Info',
      STGM_CREATE or STGM_READ or STGM_SHARE_EXCLUSIVE, 0, 0, stm));
  end;
end;
procedure TfrmMain.FormDestroy(Sender: TObject);
var
  stm: IStream;
  OS: TOleStream;
  Writer: TWriter;
begin
  OleCheck(FRootStorage.OpenStream('Info', nil,
    STGM_WRITE or STGM_SHARE_EXCLUSIVE, 0, stm));
  OS := TOleStream.Create(stm);
  try
    Writer := TWriter.Create(OS, 1024);
    try
      Writer.WriteString(FCarName);
      Writer.WriteInteger(FYear);
    finally
      Writer.Free;
    end;
  finally
    OS.Free;
```

```
  end;
end;
function TfrmMain.ActivateForm(ACaption: string): Boolean;
var
  Index: Integer;
begin
  Result := False;
  for Index := 0 to MDIChildCount - 1 do
    if MDIChildren[Index].Caption = ACaption then begin
      if MDIChildren[Index].WindowState = wsMinimized then
        MDIChildren[Index].WindowState := wsNormal;
      MDIChildren[Index].SetFocus;
      Result := True;
      Exit;
    end;
end;
procedure TfrmMain.Gas1Click(Sender: TObject);
var
  frmGas: TfrmGas;
begin
  if not ActivateForm('Gas') then begin
    frmGas := TfrmGas.Create(self);
    frmGas.Show;
  end;
end;
procedure TfrmMain.Oil1Click(Sender: TObject);
var
  frmOil: TfrmOil;
begin
  if not ActivateForm('Oil') then begin
    frmOil := TfrmOil.Create(self);
    frmOil.Show;
  end;
end;
procedure TfrmMain.CarInformation1Click(Sender: TObject);
var
  frmCarInfo: TfrmCarInfo;
begin
  frmCarInfo := TfrmCarInfo.Create(nil);
  try
    frmCarInfo.ecDescription.Text := FCarName;
    frmCarInfo.ecYear.Text := IntToStr(FYear);
    if frmCarInfo.ShowModal = mrOk then begin
      FCarName := frmCarInfo.Description;
      FYear := frmCarInfo.Year;
    end;
  finally
    frmCarInfo.Free;
  end;
end;
End.
```

El Listado 7.3 contiene el código fuente para mostrar la información del coche. Permite al usuario ver y/o cambiar el nombre y año del modelo.

Listado 7.3: CarDemo - *CarInfoForm.pas*

```
unit CarInfoForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
    Dialogs, StdCtrls, ExtCtrls;

type
  TfrmCarInfo = class(Tform)
    PnlBottom: TPanel;
    pnlClient: TPanel;
    btnOK: TButton;
    btnCancel: Tbutton;
    Label1: TLabel;
    Label2: TLabel;
    ecDescription: TEdit;
    ecYear: TEdit;
    procedure btnOKClick(Sender: TObjetct);
  private
    FYear: Integer;
    FDescription: string;
    { Public declarations }
  public
    { Public declarations }
    property Description: string read FDescription;
    property Year: Integer read FYear;
  end;

implementation

{SR *.DFM}

procedure TfrmCarInfo.btnOKClick(Sender: TObject);
begin
  FDescription := ecDescription.Text;
  try
    Fyear := StrToInt(ecYear.Text);
  except
    ModalResult := mrNone;
    ActiveControl := ecYear;
    ShowMessage('Year is not a valid number.');
    exit;
  end;
end;
end.
```

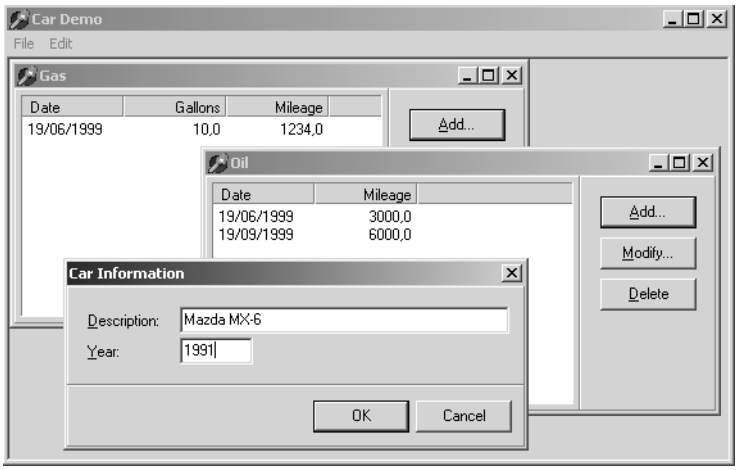


Figura 7.3: Visualizando información de mi coche.

En el Listado 7.4 se implementa *TFrmGas*. *TfrmGas* se utiliza para introducir la fecha y el kilometraje registrado cuando se hace una compra de gasolina. Aquí también se definen las clases auxiliares *TGasItem* y *TGasItems*. He decidido almacenar las líneas de detalle en una colección por dos razones. Primero, una colección es una estructura de datos ideal para almacenar datos homogéneos. Más importante, sin embargo, es que con un poco de finura podemos convencer al mecanismo de *streaming* de Delphi para leer y escribir la colección entera a un flujo con un par de líneas de código.

Eche un vistazo al método *FormDestroy* mostrado en el Listado 7.4. Este se repite a continuación para su referencia.

```
OleCheck(frmMain.RootStorage.CreateStream('Gas',
  STGM_CREATE or STGM_WRITE or STGM_SHARE_EXCLUSIVE, 0, 0, stm));
Dummy := TDummy.Create(nil);
Dummy.Items.Assign(FItems);
try
  OS := TOleStream.Create(stm);
  try
    OS.WriteComponent(Dummy);
  finally
    OS.Free;
  end;
finally
  Dummy.Free;
end;
```

Después de crear el flujo *Gas*, la aplicación crea un componente temporal que se utiliza para serializar los elementos. Delphi no soporta directamente la serialización de colecciones. Sin embargo, si una colección es una propiedad publicada de un componente, entonces Delphi automáticamente serializa la colección cuando tiene que

serializar el componente. Así que asigno la colección al componente *TDummy* justo el tiempo necesario para serializarlo, y seguidamente destruyo el componente *TDummy*.

FormCreate se ocupa de leer las líneas de detalle de pedidos de gasolina de una manera análoga.

Listado 7.4: CarDemo - *GasForm.pas*

```
unit GasForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, ComObj, ActiveX, AxCtrls, ComCtrls, ExtCtrls, StdCtrls;
type
  TGasItem = class(TCollectionItem)
  private
    FDate: TDateTime;
    FGallons: Double;
    FMileage: Double;
  public
    procedure Assign(Source: TPersistent); override;
  published
    property Date: TDateTime read FDate write FDate;
    property Gallons: Double read FGallons write FGallons;
    property Mileage: Double read FMileage write FMileage;
  end;
  TGasItems = class(TCollection)
  public
    constructor Create;
    function Add: TGasItem;
    function GetItem(Index: Integer): TGasItem;
    procedure SetItem(Index: Integer; Value: TGasItem);
    property Items[Index: Integer]: TGasItem
      read GetItem write SetItem;
  end;
  TfrmGas = class(TForm)
    Panel1: TPanel;
    Panel2: TPanel;
    list: TListView;
    btnAdd: TButton;
    btnModify: TButton;
    btnDelete: TButton;
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure FormDestroy(Sender: TObject);
    procedure btnAddClick(Sender: TObject);
    procedure btnModifyClick(Sender: TObject);
    procedure btnDeleteClick(Sender: TObject);
  private
    { Private declarations }
    FItems: TGasItems;
```

```
    procedure AddListItem(GasItem: TGasItem);
    procedure SetListItem(LI: TListItem; GasItem: TGasItem);
    procedure EnableButtons;
  public
    { Public declarations }
  end;
implementation
uses MainForm, GasItemForm;
{$R *.DFM}
type
  TDummy = class(TComponent)
  private
    FItems: TGasItems;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    property Items: TGasItems read FItems write FItems;
  end;
{ TfrmGas }
procedure TfrmGas.AddListItem(GasItem: TGasItem);
var
  LI: TListItem;
begin
  LI := list.Items.Add;
  LI.Caption := DateToStr(GasItem.Date);
  LI.SubItems.Add(FloatToStrF(GasItem.Gallons, ffFixed, 8, 1));
  LI.SubItems.Add(FloatToStrF(GasItem.Mileage, ffFixed, 8, 1));
  list.Selected := LI;
end;
procedure TfrmGas.SetListItem(LI: TListItem; GasItem: TGasItem);
begin
  LI.Caption := DateToStr(GasItem.Date);
  LI.SubItems[0] := FloatToStrF(GasItem.Gallons, ffFixed, 8, 1);
  LI.SubItems[1] := FloatToStrF(GasItem.Mileage, ffFixed, 8, 1);
end;
procedure TfrmGas.EnableButtons;
begin
  btnModify.Enabled := list.Selected  nil;
  btnDelete.Enabled := list.Selected  nil;
end;
procedure TfrmGas.FormCreate(Sender: TObject);
var
  stm: IStream;
  Dummy: TDummy;
  OS: TOleStream;
  Index: Integer;
  GasItem: TGasItem;
begin
  FItems := TGasItems.Create;
  if SUCCEEDED(frmMain.RootStorage.OpenStream('Gas', nil,
      STGM_READ or STGM_SHARE_EXCLUSIVE, 0, stm)) then begin
```

```
    Dummy := TDummy.Create(nil);
    try
      OS := TOleStream.Create(stm);
      try
        OS.ReadComponent(Dummy);
        FItems.Assign(Dummy.FItems);
      finally
        OS.Free;
      end;
    finally
      Dummy.Free;
    end;
  end else begin
    OleCheck(frmMain.RootStorage.CreateStream('Gas',
      STGM_CREATE or STGM_READ or STGM_SHARE_EXCLUSIVE, 0, 0, stm));
  end;
  for Index := 0 to FItems.Count - 1 do begin
    GasItem := FItems.Items[Index];
    AddListItem(GasItem);
  end;
end;
procedure TfrmGas.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  Action := caFree;
end;
procedure TfrmGas.FormDestroy(Sender: TObject);
var
  stm: IStream;
  OS: TOleStream;
  Dummy: TDummy;
begin
  OleCheck(frmMain.RootStorage.CreateStream('Gas',
    STGM_CREATE or STGM_WRITE or STGM_SHARE_EXCLUSIVE, 0, 0, stm));
  Dummy := TDummy.Create(nil);
  try
    Dummy.Items.Assign(FItems);
    OS := TOleStream.Create(stm);
    try
      OS.WriteComponent(Dummy);
    finally
      OS.Free;
    end;
  finally
    Dummy.Free;
  end;
  FItems.Free;
end;
procedure TfrmGas.btnAddClick(Sender: TObject);
var
  frmGasItem: TfrmGasItem;
  GasItem: TGasItem;
```

```
begin
  frmGasItem := TfrmGasItem.Create(nil);
  try
    if frmGasItem.ShowModal = mrOk then begin
      GasItem := FItems.Add;
      GasItem.Date := frmGasItem.Date;
      GasItem.Gallons := frmGasItem.Gallons;
      GasItem.Mileage := frmGasItem.Mileage;
      AddListItem(GasItem);
      EnableButtons;
    end;
  finally
    frmGasItem.Free;
  end;
end;
procedure TfrmGas.btnModifyClick(Sender: TObject);
var
  frmGasItem: TfrmGasItem;
  GasItem: TGasItem;
begin
  frmGasItem := TfrmGasItem.Create(nil);
  try
    GasItem := FItems.Items[list.Selected.Index];
    frmGasItem.dtDate.Date := GasItem.Date;
    frmGasItem.ecGallons.Text := FloatToStr(GasItem.Gallons);
    frmGasItem.ecMileage.Text := FloatToStr(GasItem.Mileage);
    if frmGasItem.ShowModal = mrOk then begin
      GasItem.Date := frmGasItem.Date;
      GasItem.Gallons := frmGasItem.Gallons;
      GasItem.Mileage := frmGasItem.Mileage;
      SetListItem(list.Selected, GasItem);
    end;
  finally
    frmGasItem.Free;
  end;
end;
procedure TfrmGas.btnDeleteClick(Sender: TObject);
var
  Index: Integer;
  GasItem: TGasItem;
begin
  Index := list.Selected.Index;
  list.Items.Delete(Index);
  GasItem := FItems.Items[Index];
  GasItem.Free;
  EnableButtons;
end;
{ TGasItem }
procedure TGasItem.Assign(Source: TPersistent);
begin
  if Source is TGasItem then begin
    FDate := TGasItem(Source).FDate;
```

```
    FGallons := TGasItem(Source).FGallons;
    FMileage := TGasItem(Source).FMileage;
  end else
    inherited Assign(Source);
end;
{ TGasItems }
function TGasItems.Add: TGasItem;
begin
  Result := inherited Add as TGasItem;
end;
constructor TGasItems.Create;
begin
  inherited Create(TGasItem);
end;
function TGasItems.GetItem(Index: Integer): TGasItem;
begin
  Result := inherited GetItem(Index) as TGasItem;
end;
procedure TGasItems.SetItem(Index: Integer; Value: TGasItem);
begin
  inherited SetItem(Index, Value);
end;
{ TDummy }
constructor TDummy.Create(AOwner: TComponent);
begin
  FItems := TGasItems.Create;
end;
destructor TDummy.Destroy;
begin
  FItems.Free;
end;
end.
```

El Listado 7.5 muestra el código fuente del cuadro de diálogo *TfrmGasItem*. El usuario introducirá la información asociada a una compra de gasolina (fecha, cantidad de galones y kilometraje) en este cuadro de diálogo.

Listado 7.5: CarDemo – *GasItemForm.pas*

```
unit GasItemForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  StdCtrls, ComCtrls, ExtCtrls;
type
  TfrmGasItem = class(TForm)
    pnlBottom: TPanel;
    pnlClient: TPanel;
    btnOK: TButton;
    btnCancel: TButton;
```

```
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    dtDate: TDateTimePicker;
    ecGallons: TEdit;
    ecMileage: TEdit;
    procedure btnOKClick(Sender: TObject);
  private
    FDate: TDateTime;
    FGallons: Double;
    FMileage: Double;
    { Private declarations }
  public
    { Public declarations }
    property Date: TDateTime read FDate;
    property Gallons: Double read FGallons;
    property Mileage: Double read FMileage;
  end;
implementation
{$R *.DFM}
procedure TfrmGasItem.btnOKClick(Sender: TObject);
begin
  FDate := dtDate.Date;
  try
    FGallons := StrToFloat(ecGallons.Text);
  except
    ModalResult := mrNone;
    ActiveControl := ecGallons;
    ShowMessage('Gallons is not a valid number.');
    Exit;
  end;
  try
    FMileage := StrToFloat(ecMileage.Text);
  except
    ModalResult := mrNone;
    ActiveControl := ecMileage;
    ShowMessage('Mileage is not a valid number.');
    Exit;
  end;
end;
end.
```

Los Listados 7.6 y 7.7 son casi idénticos a los Listados 7.4 y 7.5, excepto que contienen el código para controlar las compras de aceite en lugar de carburante.

Listado 7.6: CarDemo - *OilForm.pas*

```
unit OilForm;
interface
Uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,Dialogs,
```

```
  ComCtrls, StdCtrls, ExtCtrls, ActiveX, AxCtrls, ComObj;
type
  TOilItem = class(TCollectionItem)
  private
    FDate: TDateTime;
    FMileage: Double;
  public
    procedure Assign(Source: TPersistent); override;
  published
    property Date: TDateTime read FDate write FDate;
    property Mileage: Double read FMileage write FMileage;
  end;
  TOilItems = class(TCollection)
  public
    constructor Create;
    function Add: TOilItem;
    function GetItem(Index: Integer): TOilItem;
    procedure SetItem(Index: Integer; Value: TOilItem);
    property Items[Index: Integer]: TOilItem
      read GetItem write SetItem;
  end;
  TfrmOil = class(TForm)
    Panel1: TPanel;
    btnAdd: TButton;
    btnModify: TButton;
    btnDelete: TButton;
    Panel2: TPanel;
    list: TListView;
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure FormDestroy(Sender: TObject);
    procedure btnAddClick(Sender: TObject);
    procedure btnModifyClick(Sender: TObject);
    procedure btnDeleteClick(Sender: TObject);
  private
    { Private declarations }
    FItems: TOilItems;
    procedure AddListItem(OilItem: TOilItem);
    procedure SetListItem(LI: TListItem; OilItem: TOilItem);
    procedure EnableButtons;
  public
    { Public declarations }
  end;
var
  frmOil: TfrmOil;
implementation
uses MainForm, OilItemForm;
{$R *.DFM}
type
  TDummy = class(TComponent)
  private
    FItems: TOilItems;
```

```
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    property Items: TOilItems read FItems write FItems;
  end;
{ TfrmOil }
procedure TfrmOil.AddListItem(OilItem: TOilItem);
var
  LI: TListItem;
begin
  LI := list.Items.Add;
  LI.Caption := DateToStr(OilItem.Date);
  LI.SubItems.Add(FloatToStrF(OilItem.Mileage, ffFixed, 8, 1));
  list.Selected := LI;
end;
procedure TfrmOil.SetListItem(LI: TListItem; OilItem: TOilItem);
begin
  LI.Caption := DateToStr(OilItem.Date);
  LI.SubItems[0] := FloatToStrF(OilItem.Mileage, ffFixed, 8, 1);
end;
procedure TfrmOil.EnableButtons;
begin
  btnModify.Enabled := list.Selected  nil;
  btnDelete.Enabled := list.Selected  nil;
end;
procedure TfrmOil.FormCreate(Sender: TObject);
var
  stm: IStream;
  Dummy: TDummy;
  OS: TOleStream;
  Index: Integer;
  OilItem: TOilItem;
begin
  FItems := TOilItems.Create;
  if SUCCEEDED(frmMain.RootStorage.OpenStream('Oil', nil,
    STGM_READ or STGM_SHARE_EXCLUSIVE, 0, stm)) then begin
    Dummy := TDummy.Create(nil);
    try
      OS := TOleStream.Create(stm);
      try
        OS.ReadComponent(Dummy);
        FItems.Assign(Dummy.FItems);
      finally
        OS.Free;
      end;
    finally
      Dummy.Free;
    end;
  end else begin
    OleCheck(frmMain.RootStorage.CreateStream('Oil',
      STGM_CREATE or STGM_READ or STGM_SHARE_EXCLUSIVE, 0, 0, stm));
```

```
  end;
  for Index := 0 to FItems.Count - 1 do begin
    OilItem := FItems.Items[Index];
    AddListItem(OilItem);
  end;
end;
procedure TfrmOil.FormClose(Sender: TObject;
var Action: TCloseAction);
begin
  Action := caFree;
end;
procedure TfrmOil.FormDestroy(Sender: TObject);
var
  stm: IStream;
  OS: TOleStream;
  Dummy: TDummy;
begin
  OleCheck(frmMain.RootStorage.CreateStream('Oil',
    STGM_CREATE or STGM_WRITE or STGM_SHARE_EXCLUSIVE, 0, 0, stm));
  Dummy := TDummy.Create(nil);
  try
    Dummy.Items.Assign(FItems);
    OS := TOleStream.Create(stm);
    try
      OS.WriteComponent(Dummy);
    finally
      OS.Free;
    end;
  finally
    Dummy.Free;
  end;
  FItems.Free;
end;
procedure TfrmOil.btnAddClick(Sender: TObject);
var
  frmOilItem: TfrmOilItem;
  OilItem: TOilItem;
begin
  frmOilItem := TfrmOilItem.Create(nil);
  try
    if frmOilItem.ShowModal = mrOk then begin
      OilItem := FItems.Add;
      OilItem.Date := frmOilItem.Date;
      OilItem.Mileage := frmOilItem.Mileage;
      AddListItem(OilItem);
      EnableButtons;
    end;
  finally
    frmOilItem.Free;
  end;
end;
procedure TfrmOil.btnModifyClick(Sender: TObject);
```

```
var
  frmOilItem: TfrmOilItem;
  OilItem: TOilItem;
begin
  frmOilItem := TfrmOilItem.Create(nil);
  try
    OilItem := FItems.Items[list.Selected.Index];
    frmOilItem.dtDate.Date := OilItem.Date;
    frmOilItem.ecMileage.Text := FloatToStr(OilItem.Mileage);
    if frmOilItem.ShowModal = mrOk then begin
      OilItem.Date := frmOilItem.Date;
      OilItem.Mileage := frmOilItem.Mileage;
      SetListItem(list.Selected, OilItem);
    end;
  finally
    frmOilItem.Free;
  end;
end;
procedure TfrmOil.btnDeleteClick(Sender: TObject);
var
  Index: Integer;
  OilItem: TOilItem;
begin
  Index := list.Selected.Index;
  list.Items.Delete(Index);
  OilItem := FItems.Items[Index];
  OilItem.Free;
  EnableButtons;
end;
{ TOilItem }
procedure TOilItem.Assign(Source: TPersistent);
begin
  if Source is TOilItem then begin
    FDate := TOilItem(Source).FDate;
    FMileage := TOilItem(Source).FMileage;
  end else
    inherited Assign(Source);
end;
{ TOilItems }
function TOilItems.Add: TOilItem;
begin
  Result := inherited Add as TOilItem;
end;
constructor TOilItems.Create;
begin
  inherited Create(TOilItem);
end;
function TOilItems.GetItem(Index: Integer): TOilItem;
begin
  Result := inherited GetItem(Index) as TOilItem;
end;
procedure TOilItems.SetItem(Index: Integer; Value: TOilItem);
```

```
begin
  inherited SetItem(Index, Value);
end;
{ TDummy }
constructor TDummy.Create(AOwner: TComponent);
begin
  FItems := TOilItems.Create;
end;
destructor TDummy.Destroy;
begin
  FItems.Free;
end;
end.
```

Listado 7.7: CarDemo - *OilItemFrom.pas*

```
unit OilItemForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  StdCtrls, ComCtrls, ExtCtrls;
type
  TfrmOilItem = class(TForm)
    pnlBottom: TPanel;
    btnOK: TButton;
    btnCancel: TButton;
    pnlClient: TPanel;
    Label1: TLabel;
    Label3: TLabel;
    dtDate: TDateTimePicker;
    ecMileage: TEdit;
    procedure btnOKClick(Sender: TObject);
  private
    { Private declarations }
    FDate: TDateTime;
    FMileage: Double;
  public
    { Public declarations }
    property Date: TDateTime read FDate;
    property Mileage: Double read FMileage;
  end;
var
  frmOilItem: TfrmOilItem;
implementation
{$R *.DFM}
procedure TfrmOilItem.btnOKClick(Sender: TObject);
begin
  FDate := dtDate.Date;
  try
    FMileage := StrToFloat(ecMileage.Text);
  except
```

```
      ModalResult := mrNone;
      ActiveControl := ecMileage;
      ShowMessage('Mileage is not a valid number.');
      Exit;
    end;
  end;
  end.
```

Cuando ejecute la aplicación, se presentará el formulario principal vacío. Seleccione *File | Gas* del menú principal para ver el historial de compras de carburante o introducir una nueva compra. Seleccione *File | Oil* para ver o introducir datos asociados a las compras de aceite.

La Figura 7.4 muestra la aplicación *CarDemo* con una muestra de los datos de la gasolina y cambios de aceite.

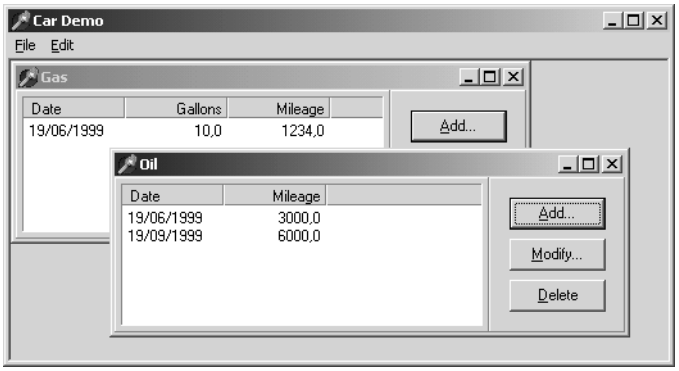


Figura 7.4: *CarDemo* permite controlar las compras de gasolina y los cambios de aceite de su coche.

Ejemplo: Visualizar un fichero de almacenamiento estructurado

Después de haber visto un ejemplo de aplicación que utiliza el almacenamiento estructurado como formato nativo de ficheros, echemos un vistazo a una aplicación que puede mostrar la estructura interna de cualquier fichero de almacenamiento estructurado.

Como elemento adicional, *SSView* contiene un elemento de menú *Compress!* que nos permite comprimir un fichero de almacenamiento estructurado después de haber visto su contenido.

Los Listados 7.8 y 7.9 contienen el código completo de la aplicación *SSView*.

Listado 7.8: SSView - *SSView.dpr*

```
program SSView;
uses
  Forms,
```

```
  MainForm in 'MainForm.pas' {Form1};
{$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Listado 7.9: SSView - *MainForm.pas*

```
unit MainForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, Menus, ComCtrls, ExtCtrls, ActiveX, ComObj;
type
  TForm1 = class(TForm)
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    FileExit1: TMenuItem;
    OpenDialog1: TOpenDialog;
    tree: TTreeView;
    FileOpen1: TMenuItem;
    N1: TMenuItem;
    pnlHeader: TPanel;
    Bevel1: TBevel;
    Compress1: TMenuItem;
    procedure FileExit1Click(Sender: TObject);
    procedure FileOpen1Click(Sender: TObject);
    procedure Compress1Click(Sender: TObject);
  private
    { Private declarations }
    FFileName: WideString;
    procedure DisplayFileStructure;
    procedure RecurseStorage(ParentNode: TTreeNode; stg: IStorage);
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.FileExit1Click(Sender: TObject);
begin
  Close;
end;
procedure TForm1.RecurseStorage(ParentNode: TTreeNode; stg: IStorage);
var
  EnumStatStg: IEnumStatStg;
  StatStg: TStatStg;
  NodeName: string;
  ChildNode: TTreeNode;
```

```
  stgChild: IStorage;
begin
  OleCheck(stg.EnumElements(0, nil, 0, EnumStatStg));
  while EnumStatStg.Next(1, StatStg, nil) = S_OK do begin
    NodeName := StatStg.pwcsName;
    if Ord(NodeName[1]) < 32 then
      NodeName := '#' + IntToStr(Ord(NodeName[1])) +
        Copy(NodeName, 2, Length(NodeName) - 1);
{$IFDEF VER100}
    ChildNode := tree.Items.AddChild(ParentNode, NodeName +
      ' (' + FloatToStr(StatStg.cbSize) + ' bytes)');
{$ELSE}
    ChildNode := tree.Items.AddChild(ParentNode, NodeName +
      ' (' + IntToStr(StatStg.cbSize) + ' bytes)');
{$ENDIF}
    if StatStg.dwType = STGTY_STORAGE then begin
      // See if we can open the child storage
      OleCheck(stg.OpenStorage(StatStg.pwcsName, nil,
        STGM_READ or STGM_SHARE_EXCLUSIVE, nil, 0, stgChild));
      // Process the sub-storage
      RecurseStorage(ChildNode, stgChild);
    end;
  end;
end;
procedure TForm1.DisplayFileStructure;
var
  stgRoot: IStorage;
  Node: TTreeNode;
begin
  // First, check to see if it's a structured storage file
  if StgIsStorageFile(PWideChar(FFileName))  S_OK then begin
    ShowMessage(FFileName + ' is not a structured storage file.');
    Exit;
  end;
  pnlHeader.Caption := FFileName;
  // Open the file
  OleCheck(StgOpenStorage(PWideChar(FFileName), nil,
    STGM_READ or STGM_SHARE_EXCLUSIVE, nil, 0, stgRoot));
  tree.Items.BeginUpdate;
  try
    tree.Items.Clear;
    Node := tree.Items.Add(nil, '<Root>');
    RecurseStorage(Node, stgRoot);
    Node.Expand(True);
  finally
    tree.Items.EndUpdate;
  end;
  Compress1.Enabled := True;
end;
procedure TForm1.FileOpen1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then begin
```

```
    FFileName := OpenDialog1.FileName;
    DisplayFileStructure;
  end;
end;
procedure TForm1.Compress1Click(Sender: TObject);
var
  stgOriginal: IStorage;
  stgTemp: IStorage;
  TempFileName: WideString;
begin
  // Step 1 - Open the original file
  OleCheck(StgOpenStorage(PWideChar(FFileName), nil,
    STGM_READ or STGM_SHARE_EXCLUSIVE, nil, 0, stgOriginal));
  // Step 2 - Create a new, temporary file
  TempFileName := ChangeFileExt(FFileName, '.$$$');
  OleCheck(StgCreateDocFile(PWideChar(TempFileName),
    STGM_WRITE or STGM_SHARE_EXCLUSIVE, 0, stgTemp));
  // Step 3 - Copy the old file to the new one
  stgOriginal.CopyTo(0, nil, nil, stgTemp);
  // Step 4 - Close the temporary file
  stgTemp := nil;
  // Step 5 - Close the original file
  stgOriginal := nil;
  // Step 6 - Delete the original file
  DeleteFile(FFileName);
  // Step 7 - Rename the temporary file
  RenameFile(TempFileName, FFileName);
end;
end.
```

El núcleo de *SSView* está contenido en el procedimiento *RecurseStorage*. Cuando selecciona *File | Open* del menú principal, *SSView* primero verifica si el fichero es un fichero de almacenamiento estructurado válido. Si lo es, abre el fichero mediante una llamada a *StgOpenStorage*, y seguidamente llama a *RecurseStorage*, pasándole el almacén raíz del fichero como parámetro.

RecurseStorage utiliza la técnica descrita en la sección "Iterando a través de un fichero de almacenamiento estructurado" para navegar por los sub-almacenes y flujos contenidos en el almacenamiento. Cuando un sub-almacén es encontrado, *RecurseStorage* se llama a sí misma recursivamente para procesar éste.

En el apartado "Almacenamiento estructurado simple", al comienzo de este capítulo, hemos examinado las características y limitaciones de un fichero de almacenamiento estructurado abierto con el modo simple. Es interesante usar *SSView* para detectar si un fichero de almacenamiento estructurado ha sido creado con el modo simple. Por ejemplo, si visualiza un documento Word pequeño en *SSView*, verá un flujo de exactamente 4096 bytes - una indicación segura de que Word salva sus documentos usando la opción *STGM_SIMPLE*.

Por otra parte, en la aplicación *CarDemo* no he especificado la opción *STGM_SIMPLE* al crear el fichero de almacenamiento estructurado. Como consecuencia, si visualizamos el fichero creado con esta aplicación, veremos un flujo con menos de 4096 bytes.

La Figura 7.5 muestra la estructura de un documento Word en *SSView*.

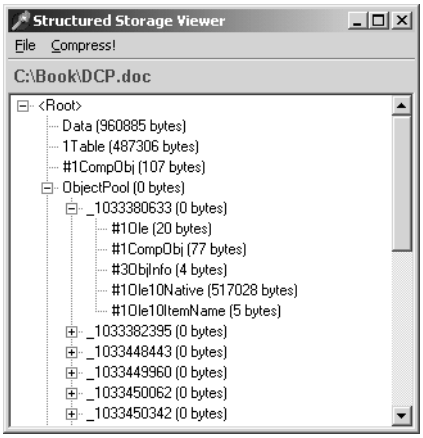


Figura 7.5: Visualizando la estructura de un documento Word.

Resumen

En este capítulo hemos examinado el almacenamiento estructurado: qué es, por qué es útil y cómo aprovechar la ventajas que ofrece en nuestras aplicaciones.

Como ha visto, realmente no es más difícil tratar con ficheros de almacenamiento estructurado que con ficheros estándar en las aplicaciones Delphi. Después de ocuparse de las líneas de código adicional necesarias para abrir el fichero y los flujos necesarios, puede simplemente utilizar *TOleStream* para leer y escribir en el fichero.

Le sugiero que trate de familiarizarse con el almacenamiento estructurado tanto como le sea posible, y considere utilizarlo en sus nuevos proyectos. Microsoft ha indicado que en una futura versión de Windows el almacenamiento estructurado podría convertirse en un formato estándar de ficheros. Incorporarlo a sus aplicaciones de hoy puede darle ventaja para cuando llegue ese día.

En el próximo capítulo examinaré la relación entre el almacenamiento estructurado y OLE. En particular, cubriré los contenedores OLE y los documentos compuestos, y explicaré qué son los conjuntos de propiedades OLE.

Por otra parte, en la aplicación *CarDemo* no he especificado la opción *STGM_SIMPLE* al crear el fichero de almacenamiento estructurado. Como consecuencia, si visualizamos el fichero creado con esta aplicación, veremos un flujo con menos de 4096 bytes.

La Figura 7.5 muestra la estructura de un documento Word en *SSView*.

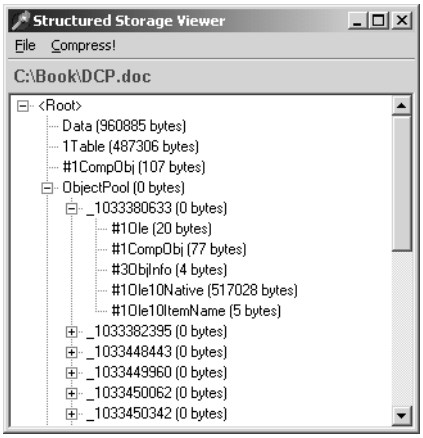


Figura 7.5: Visualizando la estructura de un documento Word.

Resumen

En este capítulo hemos examinado el almacenamiento estructurado: qué es, por qué es útil y cómo aprovechar la ventajas que ofrece en nuestras aplicaciones.

Como ha visto, realmente no es más difícil tratar con ficheros de almacenamiento estructurado que con ficheros estándar en las aplicaciones Delphi. Después de ocuparse de las líneas de código adicional necesarias para abrir el fichero y los flujos necesarios, puede simplemente utilizar *TOleStream* para leer y escribir en el fichero.

Le sugiero que trate de familiarizarse con el almacenamiento estructurado tanto como le sea posible, y considere utilizarlo en sus nuevos proyectos. Microsoft ha indicado que en una futura versión de Windows el almacenamiento estructurado podría convertirse en un formato estándar de ficheros. Incorporarlo a sus aplicaciones de hoy puede darle ventaja para cuando llegue ese día.

En el próximo capítulo examinaré la relación entre el almacenamiento estructurado y OLE. En particular, cubriré los contenedores OLE y los documentos compuestos, y explicaré qué son los conjuntos de propiedades OLE.

8

Almacenamiento Estructurado y OLE

En el Capítulo 7, "Almacenamiento Estructurado", hemos estudiado qué es el almacenamiento estructurado y cómo puede utilizarlo en sus aplicaciones. Aunque el almacenamiento estructurado es una tecnología muy utilizable por sí misma, OLE añade un par de capas más encima de él. Este capítulo examina esas dos capas: los conjuntos de propiedades OLE y los documentos compuestos/contenedores OLE.

Conjuntos de Propiedades OLE

De la misma forma que Microsoft ha definido y publicado un número de interfaces estándar COM, también ha definido un formato común de flujo para guardar información resumida sobre documentos en ficheros de almacenamiento estructurado. Si ha utilizado alguna vez Microsoft Word u otra aplicación de Microsoft Office, habrá visto que éste almacena información acerca del documento, como el nombre del autor, el tema, y el tiempo total de edición, dentro del propio documento. Este resumen puede ser visualizado no sólo dentro de Microsoft Word, sino también directamente en el Explorador de Windows. La Figura 8.1 muestra un resumen de la información almacenada en el documento Word en el que se almacena este libro.

Antes de la aparición de Windows 95, Microsoft Word utilizaba un formato de fichero propio para guardar sus documentos. Los documentos de Word incorporaban un resumen, pero la única forma de obtenerlo era abriendo el documento desde Word. Después de la salida de Windows 95, Microsoft decidió guardar el resumen de una forma estándar, de forma que cualquier programa que entendiera el método de almacenamiento pudiera leerlo.

El almacenamiento estructurado es el vehículo usado, y los conjuntos de propiedades OLE (*OLE Property Sets*) el método. Desafortunadamente, cuando los conjuntos de propiedades OLE fueron introducidos por primera vez, el formato fue publicado, pero Microsoft no hizo público el API para leer (y escribir) los conjuntos de propiedades, haciendo muy difícil a los programadores incorporarlos en sus aplicaciones.

Figura 8.1: Visualizando el resumen de información a través del Explorador de Windows.

Desde entonces, Microsoft publicó las interfaces *IPropertyStorageSet* e *IPropertyStorage*, y ha hecho mucho más simple la tarea de leer y escribir conjuntos de propiedades OLE.

Convenios de los conjuntos de propiedades OLE

Los conjuntos de propiedades OLE se almacenan en flujos. Los nombres de flujo que comienzan con el carácter *Chr(5)* están reservados para los conjuntos de propiedades. En este libro presentaré estos nombre de flujo como *#5SummaryInformation*, por ejemplo, donde *#5* se refiere al único carácter *Chr(5)*, y no a dos caracteres independientes '#' y '5'. El tamaño total de un flujo de conjuntos de propiedades está limitado a 256 KB.

Hasta el presente, Microsoft ha publicado tres nombres de conjuntos de propiedades estándar: *#5SummaryInformation*, *#5DocumentSummaryInformation* y *#5UserDefinedProperties*.

Las interfaces IPropertySetStorage e IPropertyStorage

Como he mencionado anteriormente, las interfaces *IPropertySetStorage* e *IPropertyStorage* proporcionan los medios necesarios para leer y escribir conjuntos de propiedades OLE.

Delphi 4 y las versiones posteriores incluyen la definición de estas interfaces en el fichero *ActiveX.pas*. Si está utilizando Delphi 3, refiérase al Listado 8.1 para ver la definición de las interfaces y estructuras de soporte necesarias para leer y escribir conjuntos de propiedades.

Listado 8.1: *PropSets.pas* - Definiciones de interfaces para Delphi 3

```
unit PropSets;
interface
uses
  Windows;
const
  // Property IDs for the SummaryInformation Property Set
  PIDSI_TITLE               = $00000002;  // VT_LPSTR
  PIDSI_SUBJECT             = $00000003;  // VT_LPSTR
  PIDSI_AUTHOR              = $00000004;  // VT_LPSTR
  PIDSI_KEYWORDS            = $00000005;  // VT_LPSTR
  PIDSI_COMMENTS            = $00000006;  // VT_LPSTR
  PIDSI_TEMPLATE            = $00000007;  // VT_LPSTR
  PIDSI_LASTAUTHOR          = $00000008;  // VT_LPSTR
  PIDSI_REVNUMBER           = $00000009;  // VT_LPSTR
  PIDSI_EDITTIME            = $0000000a;  // VT_FILETIME (UTC)
  PIDSI_LASTPRINTED         = $0000000b;  // VT_FILETIME (UTC)
  PIDSI_CREATE_DTM          = $0000000c;  // VT_FILETIME (UTC)
  PIDSI_LASTSAVE_DTM        = $0000000d;  // VT_FILETIME (UTC)
  PIDSI_PAGECOUNT           = $0000000e;  // VT_I4
  PIDSI_WORDCOUNT           = $0000000f;  // VT_I4
  PIDSI_CHARCOUNT           = $00000010;  // VT_I4
  PIDSI_THUMBNAIL           = $00000011;  // VT_CF
  PIDSI_APPNAME             = $00000012;  // VT_LPSTR
  PIDSI_DOC_SECURITY        = $00000013;  // VT_I4
  PRSPEC_INVALID            = $ffffffff;
  PRSPEC_LPWSTR             = 0;
  PRSPEC_PROPID             = 1;
type
  PShortInt = ^ShortInt;
  TOleDate = Double;
  POleDate = ^TOleDate;
  POleStr = PWideChar;
  PPOleStr = ^POleStr;
  PBStr = ^TBStr;
  TBStr = POleStr;
  TOleBool = WordBool;
  POleBool = ^TOleBool;
  PSCODE = ^Integer;
  SCODE = Integer;
  PLargeInteger = ^TLargeInteger;
  _LARGE_INTEGER = record
    case Integer of
    0: (
      LowPart: DWORD;
      HighPart: Longint);
    1: (
      QuadPart: LONGLONG);
  end;
  TLargeInteger = _LARGE_INTEGER;
  LARGE_INTEGER = _LARGE_INTEGER;
```

```
ULARGE_INTEGER = record
  case Integer of
  0: (
    LowPart: DWORD;
    HighPart: DWORD);
  1: (
    QuadPart: LONGLONG);
end;
PULargeInteger = ^TULargeInteger;
TULargeInteger = ULARGE_INTEGER;
PLPSTR = ^LPSTR;
PLPWSTR = ^LPWSTR;
PClipData = ^TClipData;
tagCLIPDATA = record
  cbSize: Longint;
  ulClipFmt: Longint;
  pClipData: Pointer;
end;
TClipData = tagCLIPDATA;
CLIPDATA = TClipData;
PCLSID = PGUID;
TCLSID = TGUID;
TVarType = Word;
PBlob = ^TBlob;
tagBLOB = record
  cbSize: Longint;
  pBlobData: Pointer;
end;
TBlob = tagBLOB;
BLOB = TBlob;
PROPID = ULONG;
PPropID = ^TPropID;
TPropID = PROPID;
FMTID = TGUID;
PFmtID = ^TFmtID;
TFmtID = TGUID;
{ IPropertyStorage / IPropertySetStorage }
PPropVariant = ^TPropVariant;
tagCAUB = packed record
  cElems: ULONG;
  pElems: PByte;
end;
CAUB = tagCAUB;
PCAUB = ^TCAUB;
TCAUB = tagCAUB;
tagCAI = packed record
  cElems: ULONG;
  pElems: PShortInt;
end;
CAI = tagCAI;
PCAI = ^TCAI;
TCAI = tagCAI;
```

```
tagCAUI = packed record
  cElems: ULONG;
  pElems: PWord;
end;
CAUI = tagCAUI;
PCAUI = ^TCAUI;
TCAUI = tagCAUI;
tagCAL = packed record
  cElems: ULONG;
  pElems: PLongint;
end;
CAL = tagCAL;
PCAL = ^TCAL;
TCAL = tagCAL;
tagCAUL = packed record
  cElems: ULONG;
  pElems: PULONG;
end;
CAUL = tagCAUL;
PCAUL = ^TCAUL;
TCAUL = tagCAUL;
tagCAFLT = packed record
  cElems: ULONG;
  pElems: PSingle;
end;
CAFLT = tagCAFLT;
PCAFLT = ^TCAFLT;
TCAFLT = tagCAFLT;
tagCADBL = packed record
  cElems: ULONG;
  pElems: PDouble;
end;
CADBL = tagCADBL;
PCADBL = ^TCADBL;
TCADBL = tagCADBL;
tagCACY = packed record
  cElems: ULONG;
  pElems: PCurrency;
end;
CACY = tagCACY;
PCACY = ^TCACY;
TCACY = tagCACY;
tagCADATE = packed record
  cElems: ULONG;
  pElems: POleDate;
end;
CADATE = tagCADATE;
PCADATE = ^TCADATE;
TCADATE = tagCADATE;
tagCABSTR = packed record
  cElems: ULONG;
  pElems: PBSTR;
```

```
end;
CABSTR = tagCABSTR;
PCABSTR = ^TCABSTR;
TCABSTR = tagCABSTR;
tagCABOOL = packed record
  cElems: ULONG;
  pElems: POleBool;
end;
CABOOL = tagCABOOL;
PCABOOL = ^TCABOOL;
TCABOOL = tagCABOOL;
tagCASCODE = packed record
  cElems: ULONG;
  pElems: PSCODE;
end;
CASCODE = tagCASCODE;
PCASCODE = ^TCASCODE;
TCASCODE = tagCASCODE;
tagCAPROPVARIANT = packed record
  cElems: ULONG;
  pElems: PPropVariant;
end;
CAPROPVARIANT = tagCAPROPVARIANT;
PCAPROPVARIANT = ^TCAPROPVARIANT;
TCAPROPVARIANT = tagCAPROPVARIANT;
tagCAH = packed record
  cElems: ULONG;
  pElems: PLargeInteger;
end;
CAH = tagCAH;
PCAH = ^TCAH;
TCAH = tagCAH;
tagCAUH = packed record
  cElems: ULONG;
  pElems: PULargeInteger;
end;
CAUH = tagCAUH;
PCAUH = ^TCAUH;
TCAUH = tagCAUH;
tagCALPSTR = packed record
  cElems: ULONG;
  pElems: PLPSTR;
end;
CALPSTR = tagCALPSTR;
PCALPSTR = ^TCALPSTR;
TCALPSTR = tagCALPSTR;
tagCALPWSTR = packed record
  cElems: ULONG;
  pElems: PLPWSTR;
end;
CALPWSTR = tagCALPWSTR;
PCALPWSTR = ^TCALPWSTR;
```

```
TCALPWSTR = tagCALPWSTR;
tagCAFILETIME = packed record
  cElems: ULONG;
  pElems: PFileTime;
end;
CAFILETIME = tagCAFILETIME;
PCAFILETIME = ^TCAFILETIME;
TCAFILETIME = tagCAFILETIME;
tagCACLIPDATA = packed record
  cElems: ULONG;
  pElems: PClipData;
end;
CACLIPDATA = tagCACLIPDATA;
PCACLIPDATA = ^TCACLIPDATA;
TCACLIPDATA = tagCACLIPDATA;
tagCACLSID = packed record
  cElems: ULONG;
  pElems: PCLSID;
end;
CACLSID = tagCACLSID;
PCACLSID = ^TCACLSID;
TCACLSID = tagCACLSID;
tagPROPVARIANT = packed record
  vt: TVarType;
  wReserved1: Word;
  wReserved2: Word;
  wReserved3: Word;
  case Integer of
    0: (bVal: Byte);
    1: (iVal: SmallInt);
    2: (uiVal: Word);
    3: (boolVal: TOleBool);
    4: (bool: TOleBool);
    5: (lVal: Longint);
    6: (ulVal: Cardinal);
    7: (fltVal: Single);
    8: (scode: SCODE);
    9: (hVal: LARGE_INTEGER);
    10: (uhVal: ULARGE_INTEGER);
    11: (dblVal: Double);
    12: (cyVal: Currency);
    13: (date: TOleDate);
    14: (filetime: TFileTime);
    15: (puuid: PGUID);
    16: (blob: TBlob);
    17: (pclipdata: PClipData);
    18: (pStream: Pointer{IStream});
    19: (pStorage: Pointer{IStorage});
    20: (bstrVal: TBStr);
    21: (pszVal: PAnsiChar);
    22: (pwszVal: PWideChar);
    23: (caub: TCAUB);
```

```
    24: (cai: TCAI);
    25: (caui: TCAUI);
    26: (cabool: TCABOOL);
    27: (cal: TCAL);
    28: (caul: TCAUL);
    29: (caflt: TCAFLT);
    30: (cascode: TCASCODE);
    31: (cah: TCAH);
    32: (cauh: TCAUH);
    33: (cadbl: TCADBL);
    34: (cacy: TCACY);
    35: (cadate: TCADATE);
    36: (cafiletime: TCAFILETIME);
    37: (cauuid: TCACLSID);
    38: (caclipdata: TCACLIPDATA);
    39: (cabstr: TCABSTR);
    40: (calpstr: TCALPSTR);
    41: (calpwstr: TCALPWSTR );
    42: (capropvar: TCAPROPVARIANT);
end;
PROPVARIANT = tagPROPVARIANT;
TPropVariant = tagPROPVARIANT;
tagPROPSPEC = packed record
  ulKind: ULONG;
  case Integer of
    0: (propid: TPropID);
    1: (lpwstr: POleStr);
end;
PROPSPEC = tagPROPSPEC;
PPropSpec = ^TPropSpec;
TPropSpec = tagPROPSPEC;
tagSTATPROPSTG = packed record
  lpwstrName: POleStr;
  propid: TPropID;
  vt: TVarType;
end;
STATPROPSTG = tagSTATPROPSTG;
PStatPropStg = ^TStatPropStg;
TStatPropStg = tagSTATPROPSTG;
tagSTATPROPSETSTG = packed record
  fmtid: TFmtID;
  clsid: TClsID;
  grfFlags: DWORD;
  mtime: TFileTime;
  ctime: TFileTime;
  atime: TFileTime;
  dwOSVersion: DWORD;
end;
STATPROPSETSTG = tagSTATPROPSETSTG;
PStatPropSetStg = ^TStatPropSetStg;
TStatPropSetStg = tagSTATPROPSETSTG;
IPropertySetStorage = interface;
```

```
  IPropertyStorage = interface;
  IEnumSTATPROPSTG = interface;
  IEnumSTATPROPSETSTG = interface;
  IPropertyStorage = interface(IUnknown)
    ['{00000138-0000-0000-C000-000000000046}']
    function ReadMultiple(cpspec: ULONG;
      rgpspec, rgpropvar: PPropSpec): HResult; stdcall;
    function WriteMultiple(cpspec: ULONG; rgpspec, rgpropvar: PPropSpec;
      propidNameFirst: TPropID): HResult; stdcall;
    function DeleteMultiple(cpspec: ULONG;
      rgpspec: PPropSpec): HResult; stdcall;
    function ReadPropertyNames(cpropid: ULONG; rgpropid: PPropID;
      rglpwstrName: PPOleStr): HResult; stdcall;
    function WritePropertyNames(cpropid: ULONG; rgpropid: PPropID;
      rglpwstrName: PPOleStr): HResult; stdcall;
    function DeletePropertyNames(cpropid: ULONG;
      rgpropid: PPropID): HResult; stdcall;
    function Commit(grfCommitFlags: DWORD): HResult; stdcall;
    function Revert: HResult; stdcall;
    function Enum(out ppenum: IEnumSTATPROPSTG): HResult; stdcall;
    function SetTimes(const pctime, patime,
      pmtime: TFileTime): HResult; stdcall;
    function SetClass(const clsid: TCLSID): HResult; stdcall;
    function Stat(pstatpsstg: PStatPropSetStg): HResult; stdcall;
  end;
  IPropertySetStorage = interface(IUnknown)
    ['{0000013A-0000-0000-C000-000000000046}']
    function Create(const rfmtid: TFmtID; const pclsid: TCLSID; grfFlags,
      grfMode: DWORD; out ppprstg: IPropertyStorage): HResult; stdcall;
    function Open(const rfmtid: TFmtID; grfMode: DWORD;
      out ppprstg: IPropertyStorage): HResult; stdcall;
    function Delete(const rfmtid: TFmtID): HResult; stdcall;
    function Enum(out ppenum: IEnumSTATPROPSETSTG): HResult; stdcall;
  end;
  IEnumSTATPROPSTG = interface(IUnknown)
    ['{00000139-0000-0000-C000-000000000046}']
    function Next(celt: ULONG; out rgelt;
      pceltFetched: PULONG): HResult; stdcall;
    function Skip(celt: ULONG): HResult; stdcall;
    function Reset: HResult; stdcall;
    function Clone(out ppenum: IEnumSTATPROPSTG): HResult; stdcall;
  end;
  IEnumSTATPROPSETSTG = interface(IUnknown)
    ['{0000013B-0000-0000-C000-000000000046}']
    function Next(celt: ULONG; out rgelt;
      pceltFetched: PULONG): HResult; stdcall;
    function Skip(celt: ULONG): HResult; stdcall;
    function Reset: HResult; stdcall;
    function Clone(out ppenum: IEnumSTATPROPSETSTG): HResult; stdcall;
  end;
implementation
end.
```

Leyendo propiedades

Para leer propiedades conocidas desde un flujo, se debe llamar a *IPropertyStorage.ReadMultiple*, como sigue:

```
const
  FMTID_SummaryInformation: TGUID
    = '{F29F85E0-4FF9-1068-AB91-08002B27B3D9}';
procedure TForm1.Button1Click(sender: TObject);
type
  TPropSpecArray = array [0 .. 1000] of TPropSpec;
  PPropSpecArray = ^TPropSpecArray;
  TpropVariantArray = array [0 .. 1000] of TPropVariant;
  PPropVariantArray =  ^TPropVariantArray;
var
  stgRoot: IStorage;
  stgPS: IPropertySetStorage;
  stgP: IPropertyStorage;
  ps: PPropSpecArray;
  pv: PPropVariantArray;
begin
  OleCheck(StgOpenStorage('C:\BOOK\DCP.DOC', nil,
    STGM_READ or STGM_SHARE_EXCLUSIVE, nil, 0, stgRoot));
  stgPS := stgRoot as IPropertySetStorage;
  OleCheck(stgPS.Open(FMTID_SummaryInformation,
    STGM_READ or STGM_SHARE_EXCLUSIVE, stgP));
  ps := nil;
  pv := nil;
  try
    GetMem(ps, sizeof(TPropSpec));
    GetMem(pv, sizeof(TPropVariant));
    ps[0].ulKind := PRSPEC_PROPID;
    ps[0].propid := PIDSI_TITLE;
    OleCheck(stgP.ReadMultiple(1, @ps[0], @pv[0]));
    // Hacer algo aqu con pv[0]...
  finally
    if ps  nil then FreeMem(ps);
    if pv  nil then FreeMem(pv);
  end;
end;
```

Este código abre el fichero de almacenamiento estructurado *C:\BOOK\CHAPTER2.DOC*, y entonces utiliza el operador **as** para obtener una interfaz *IPropertySetStorage* a partir de *IStorage*. Seguidamente se llama a *IPropertySetStorage.Open* para obtener una referencia a una interfaz *IPropertyStorage*. Observe que paso a este método el valor *FMTID_SummaryInformation*, que es el GUID asociado al flujo de conjunto de propiedades estándar *#5SummaryInformation*.

Seguidamente, utilizo *IPropertyStorage.ReadMultiple* para leer en este caso una sola propiedad. Se podría fácilmente haber leído diez propiedades de una sola vez, preparando con anterioridad un *array* de 10 registros de tipo *TPropSpec*. Después

que se ejecuta la llamada a *IPropertyStorage.ReadMultiple*, *pv[0]* pasa a contener el valor de la propiedad *Title* (el título del documento).

Escribiendo propiedades

Para grabar propiedades en un flujo, se debe llamar a *IPropertyStore.WriteMultiple*, como sigue:

```
const
  FMTID_SummaryInformation: TGUID
    = '{F29F85E0-4FF9-1068-AB91-08002B27B3D9}';
procedure TForm1.Button1Click(sender: TObject);
type
  TPropSpecArray = array [0 .. 1000] of TPropSpec;
  PPropSpecArray = ^TPropSpecArray;
  TPropVariantArray = array [0 .. 1000] of TPropVariant;
  PPropVariantArray =  ^TPropVariantArray;
var
  stgRoot: IStorage;
  stgPS: IPropertySetStorage;
  stgP: IPropertyStorage;
  ps: PPropSpecArray;
  pv: PPropVariantArray;
begin
  OleCheck(StgOpenStorage('C:\BOOK\DCP.DOC', nil,
    STGM_READ or STGM_SHARE_EXCLUSIVE, nil, 0, stgRoot));
  stgPS := stgRoot as IPropertySetStorage;
  OleCheck(stgPS.Open(FMTID_SummaryInformation,
    STGM_WRITE or STGM_SHARE_EXCLUSIVE, stgP));
  ps := nil;
  pv := nil;
  try
    GetMem(ps, sizeof(TPropSpec));
    GetMem(pv, sizeof(TPropVariant));

    ps[0].ulKind := PRSPEC_PROPID;
    ps[0].propid := PIDSI_TITLE;
    pv[0].vt := VT_LPSTR;
    pv[0].lpszstr := 'New title';
    OleCheck(stgP.WriteMultiple(1, @ps[0], @pv[0], 2));
  finally
    if ps  nil then FreeMem(ps);
    if pv  nil then FreeMem(pv);
  end;
end;
```

De manera similar al fragmento de código anterior, este fragmento abre el fichero de almacenamiento estructurado *C:\BOOK\DCP.DOC* y obtiene una referencia a una interfaz *IPropertyStorage*.

A continuación, utilizo *IPropertyStorage.WriteMultiple* para grabar una propiedad. Se puede grabar más de una propiedad de una sola vez preparando un *array* de más de un registro *TProcSpec*.

Eliminando propiedades

Para eliminar una o más propiedades de un flujo, se debe llamar a *IPropertyStorage.DeleteMultiple*, como sigue:

```
const
  FMTID_SummaryInformation: TGUID
    = '{F29F85E0-4FF9-1068-AB91-08002B27B3D9}';
procedure TForm1.Button1Click(sender: TObject);
type
  TPropSpecArray = array [0 .. 1000] of TPropSpec;
  PPropSpecArray = ^TPropSpecArray;
  TPropVariantArray = array [0 .. 1000] of TPropVariant;
  PPropVariantArray =  ^TPropVariantArray;
var
  stgRoot: IStorage;
  stgPS: IPropertySetStorage;
  stgP: IPropertyStorage;
  ps: PPropSpecArray;
begin
  OleCheck(StgOpenStorage('C:\BOOK\DCP.DOC', nil,
    STGM_WRITE or STGM_SHARE_EXCLUSIVE, nil, 0, stgRoot));
  stgPS := stgRoot as IPropertySetStorage;
  OleCheck(stgPS.Open(FMTID_SummaryInformation,
    STGM_WRITE or STGM_SHARE_EXCLUSIVE, stgP));
  GetMem(ps, sizeof(TPropSpec));
  try
    ps[0].ulKind := PRSPEC_PROPID;
    ps[0].propid := PIDSI_TITLE;
    OleCheck(stgP.DeleteMultiple(1, @ps[0]));
  finally
    FreeMem(ps);
  end;
end;
```

De manera similar al fragmento de código anterior, este fragmento abre el fichero de almacenamiento estructurado *C:\BOOK\DCP.DOC* y obtiene una referencia a una interfaz *IPropertyStorage*.

A continuación, utilizo *IPropertyStorage.DeleteMultiple* para eliminar una propiedad. Podemos eliminar más una propiedad de una sola vez, preparando un *array* de más de un registro *TPropSpec*.

Enumerando propiedades

ReadMultiple, *WriteMultiple* y *DeleteMultiple* requieren que se conozca el nombre o ID de la propiedad o las propiedades en cuestión. ¿Qué hacer si se desconoce qué

propiedades están presentes en el flujo? Además de los conjuntos de propiedades *#5SummaryInformation* y *#5DocumentSummaryInformation*, que contienen propiedades predefinidas, Microsoft ha definido el grupo de propiedades *#5UserDefinedProperties*. Lo interesante del conjunto de propiedades *#5UserDefinedProperties* es que cualquier usuario de Windows puede añadir propiedades a ese conjunto de propiedades utilizando el cuadro de diálogo estándar de Windows que se muestra en la Figura 8.2.

Figura 8.2: Windows permite asignar propiedades al conjunto de propiedades *#5UserDefinedProperties*.

En la figura puede verse cómo he añadido una propiedad '*Editorial*', que no forma parte de las que ofrece Word de forma predefinida.

Dado que el flujo *#5UserDefinedProperties* no contiene nombres de propiedades estándar, se necesita un modo de obtener los nombres de las propiedades almacenadas en el flujo. Aquí es donde entra en escena *IPropertyStorage.Enum*. Del mismo modo que se enumeran los almacenes y flujos en un fichero de almacenamiento estructurado, podemos enumerar las propiedades de un conjunto de propiedades. El siguiente fragmento de código muestra cómo hacerlo:

```
const
  FMTID_UserDefinedProperties: TGUID
    = '{D5CDD505-2E9C-101B-9397-08002B2CF9AE}';
procedure TForm1.Button1Click(sender: TObject);
type
  TPropSpecArray = array [0 .. 1000] of TPropSpec;
  PPropSpecArray = ^TPropSpecArray;
  TPropVariantArray = array [0 .. 1000] of TPropVariant;
```

```
  PPropVariantArray =  ^TPropVariantArray;
  TStatPropStgArray = array [0 .. 1000] of TStatPropStg;
  PStatPropStgArray =  ^TStatPropStgArray;
var
  stgRoot: IStorage;
  stgPS: IPropertySetStorage;
  stgP: IPropertyStorage;
  ps: PPropSpecArray;
  pv: PPropVariantArray;
  Enum: IEnumStatPropStg;
  sps: PStatPropStgArray;
  fetched: Longint;
begin
  OleCheck(StgOpenStorage('C:\BOOK\DCP.DOC', nil,
    STGM_READ or STGM_SHARE_EXCLUSIVE, nil, 0, stgRoot));
  stgPS := stgRoot as IPropertySetStorage;
  OleCheck(stgPS.Open(FMTID_UserDefinedProperties,
    STGM_READ or STGM_SHARE_EXCLUSIVE, stgP));
  ps := nil;
  pv := nil;
  sps := nil;
  try
    GetMem(ps, sizeof(TPropSpec));
    GetMem(pv, sizeof(TPropVariant));
    GetMem(sps, sizeof(TStatPropStg));
    OleCheck(stgP.Enum(Enum));
    while Enum.Next(1, sps[0], @fetched) = S_OK do begin
      ps[0].ulKind := PRSPEC_PROPID;
      ps[0].propid := sps[0].propid;
      OleCheck(stgP.ReadMultiple(NumIDs, @ps[0], @pv[0]));
      // Hacer algo con pv aqu...
    end;
  finally
    if ps  nil then FreeMem(ps);
    if pv  nil then FreeMem(pv);
    if sps  nil then FreeMem(sps);
  end;
end;
```

Este fragmento de código es en gran medida similar al utilizado para leer las propiedades almacenadas en el flujo *#5SummaryInformation*, con la diferencia de que utilizo *IPropertyStorage* para enumerar las propiedades, leyéndolas una por una del flujo.

Un componente para leer y escribir conjuntos de propiedades

Como puede ver, no es muy difícil leer y escribir propiedades usando las interfaces *IPropertySetStorage* e *IPropertyStorage*. Sin embargo, sería práctico disponer de un componente para hacer las cosas aún más fáciles.

El Listado 8.2 muestra el código fuente de un componente que puede leer o escribir a cualquier flujo de propiedades.

Listado 8.2: El componente *TPropertySet*

```
unit PropertySet;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,Dialogs,
{$IFDEF VER100}
  PropSets,  // For Delphi 3
{$ENDIF}
  ComObj, ActiveX;
const
  FMTID_SummaryInformation: TGUID
    = '{F29F85E0-4FF9-1068-AB91-08002B27B3D9}';
  FMTID_DocumentSummaryInformation: TGUID
    = '{D5CDD502-2E9C-101B-9397-08002B2CF9AE}';
  FMTID_UserDefinedProperties: TGUID
    = '{D5CDD505-2E9C-101B-9397-08002B2CF9AE}';
type
  TEnumPropertiesEvent= procedure(Sender:TObject; PropertyName:WideString;
    PropertyID: Integer; PropertyVariant: TPropVariant) of object;
  TPropertySet = class(TComponent)
  private
    FFileName: WideString;
    FActive: Boolean;
    FStorage: IStorage;
    FPropertySetStorage: IPropertySetStorage;
    FPropertyStorage: IPropertyStorage;
    FStreamGUID: TGUID;
    FOnEnumProperties: TEnumPropertiesEvent;
    procedure SetFileName(const Value: WideString);
    procedure SetActive(const Value: Boolean);
    procedure SetStreamGUID(const Value: TGUID);
    { Private declarations }
  protected
    { Protected declarations }
    procedure InternalOpen; dynamic;
    procedure InternalClose; dynamic;
    procedure DoEnumProperty(PropertyName:WideString; PropertyID:Integer;
      PropertyVariant: TPropVariant); dynamic;
  public
    { Public declarations }
    destructor Destroy; override;
    procedure Open;
    procedure Close;
    procedure Enumerate;
    function GetPropertyByName(APropertyName: WideString): TPropVariant;
    function GetPropertyByID(APropertyID: Integer): TPropVariant;
    procedure SetProperty(APropertyName: WideString; AValue: WideString);
    procedure DeleteProperty(APropertyName: WideString);
  published
    { Published declarations }
    property Active: Boolean read FActive write SetActive;
```

```
    property FileName: WideString read FFileName write SetFileName;
    property OnEnumProperties: TEnumPropertiesEvent
      read FOnEnumProperties write FOnEnumProperties;
    property StreamGUID: TGUID read FStreamGUID write SetStreamGUID;
  end;
procedure Register;
implementation
type
  TPropSpecArray = array [0 .. 1000] of TPropSpec;
  PPropSpecArray = ^TPropSpecArray;
  TPropVariantArray = array [0 .. 1000] of TPropVariant;
  PPropVariantArray = ^TPropVariantArray;
  TStatPropStgArray = array [0 .. 1000] of TStatPropStg;
  PStatPropStgArray = ^TStatPropStgArray;
procedure Register;
begin
  RegisterComponents('DCP', [TPropertySet]);
end;
{ TPropertySet }
procedure TPropertySet.Close;
begin
  Active := False;
end;
procedure TPropertySet.DeleteProperty(APropertyName: WideString);
var
  ps: PPropSpecArray;
begin
  GetMem(ps, sizeof(TPropSpec));
  try
    ps[0].ulKind := PRSPEC_LPWSTR;
    ps[0].lpwstr := PWideChar(APropertyName);
    OleCheck(FPropertyStorage.DeleteMultiple(1, @ps[0]));
  finally
    FreeMem(ps);
  end;
end;
destructor TPropertySet.Destroy;
begin
  Close;
end;
procedure TPropertySet.DoEnumProperty(PropertyName: WideString;
  PropertyID: Integer; PropertyVariant: TPropVariant);
begin
  if Assigned(FOnEnumProperties) then
    FOnEnumProperties(self, PropertyName, PropertyID, PropertyVariant);
end;
procedure TPropertySet.Enumerate;
var
  ps: PPropSpecArray;
  pv: PPropVariantArray;
  sps: PStatPropStgArray;
  Enum: IEnumStatPropStg;
```

```
  Fetched: LongInt;
  Prop: TPropVariant;
begin
  ps := nil;
  pv := nil;
  sps := nil;
  try
    GetMem(ps, sizeof(TPropSpec));
    GetMem(pv, sizeof(TPropVariant));
    GetMem(sps, sizeof(TStatPropStg));
    OleCheck(FPropertyStorage.Enum(Enum));
    while Enum.Next(1, sps[0], @Fetched) = S_OK do begin
      Prop := GetPropertyByID(sps[0].propid);
      DoEnumProperty(sps[0].lpwstrName, sps[0].propid, Prop);
    end;
  finally
    if ps  nil then
      FreeMem(ps);
    if pv  nil then
      FreeMem(pv);
    if sps  nil then
      FreeMem(sps);
  end;
end;
function TPropertySet.GetPropertyByID(APropertyID: Integer): TPropVariant;
var
  ps: PPropSpecArray;
  pv: PPropVariantArray;
begin
  ps := nil;
  pv := nil;
  try
    GetMem(ps, sizeof(TPropSpec));
    GetMem(pv, sizeof(TPropVariant));
    ps[0].ulKind := PRSPEC_PROPID;
    ps[0].propid := APropertyID;
    OleCheck(FPropertyStorage.ReadMultiple(1, @ps[0], @pv[0]));
    Result := pv[0];
  finally
    if ps  nil then
      FreeMem(ps);
    if pv  nil then
      FreeMem(pv);
  end;
end;
function TPropertySet.GetPropertyByName(
  APropertyName: WideString): TPropVariant;
var
  ps: PPropSpecArray;
  pv: PPropVariantArray;
begin
  ps := nil;
```

```
  pv := nil;
  try
    GetMem(ps, sizeof(TPropSpec));
    GetMem(pv, sizeof(TPropVariant));
    ps[0].ulKind := PRSPEC_LPWSTR;
    ps[0].lpwstr := PWideChar(APropertyName);
    OleCheck(FPropertyStorage.ReadMultiple(1, @ps[0], @pv[0]));
    Result := pv[0];
  finally
    if ps  nil then
      FreeMem(ps);
    if pv  nil then
      FreeMem(pv);
  end;
end;
procedure TPropertySet.InternalClose;
begin
  FPropertyStorage := nil;
  FPropertySetStorage := nil;
  FStorage := nil;
end;
procedure TPropertySet.InternalOpen;
begin
  FStorage := nil;
  if FFileName = '' then
    raise Exception.Create('File name must be set.');
  if StgIsStorageFile(PWideChar(FFileName))  S_OK then
    raise Exception.Create('File is not a structured storage file.');
  OleCheck(StgOpenStorage(PWChar(FFileName), nil,
    STGM_READWRITE or STGM_SHARE_EXCLUSIVE, nil, 0, FStorage));
  FPropertySetStorage := FStorage as IPropertySetStorage;
  OleCheck(FPropertySetStorage.Open(FStreamGUID,
    STGM_READWRITE or STGM_SHARE_EXCLUSIVE, FPropertyStorage));
end;
procedure TPropertySet.Open;
begin
  Active := True;
end;
procedure TPropertySet.SetActive(const Value: Boolean);
begin
  if FActive  Value then
    if Value then
      InternalOpen
    else
      InternalClose;
end;
procedure TPropertySet.SetFileName(const Value: WideString);
begin
  FFileName := Value;
end;
procedure TPropertySet.SetProperty(APropertyName: WideString;
  AValue: WideString);
```

```
var
  ps: PPropSpecArray;
  pv: PPropVariantArray;
begin
  ps := nil;
  pv := nil;
  try
    GetMem(ps, sizeof(TPropSpec));
    GetMem(pv, sizeof(TPropVariant));
    ps[0].ulKind := PRSPEC_LPWSTR;
    ps[0].lpwstr := PWideChar(APropertyName);
    pv[0].vt := VT_LPSTR;
    pv[0].pszval := PChar(AValue);
    OleCheck(FPropertyStorage.WriteMultiple(1, @ps[0], @pv[0], 2));
  finally
    if ps  nil then
      FreeMem(ps);
    if pv  nil then
      FreeMem(pv);
  end;
end;
procedure TPropertySet.SetStreamGUID(const Value: TGUID);
begin
  FStreamGUID := Value;
end;
end.
```

El componente *TPropertySet* es realmente muy sencillo. Simplemente reúne en un solo sitio todos los fragmentos de código mostrados para leer, escribir, eliminar y enumerar las propiedades almacenadas en un flujo. Tal cual se muestra, este componente sólo puede trabajar con una propiedad a la vez. Por supuesto, podrá mejorarlo de modo que pueda tratar con todo un *array* de propiedades a la vez.

Ejemplo: Accediendo a propiedades definidas por el usuario

Los Listados 8.3 y 8.4 muestran el código fuente completo para un programa que puede leer, escribir y eliminar propiedades al flujo *#5UserDefinedProperties*.

Listado 8.3: PropDemo - *PropDemo.dpr*

```
program PropDemo;
uses
  Forms,
  MainForm in 'MainForm.pas' {frmMain};
{$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(TfrmMain, frmMain);
  Application.Run;
end.
```

Listado 8.4: PropDemo - *MainForm.pas*

```
unit MainForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,Dialogs,
  StdCtrls, ComCtrls, ExtCtrls,
{$IFDEF VER100}
  PropSets,  // For Delphi 3
{$ENDIF}
  PropertySet, ActiveX;
type
  TfrmMain = class(TForm)
    pnlBottom: TPanel;
    pnlClient: TPanel;
    Label1: TLabel;
    listProperties: TListView;
    btnClose: TButton;
    Label2: TLabel;
    Label3: TLabel;
    ecPropertyName: TEdit;
    ecValue: TEdit;
    btnDelete: TButton;
    btnSet: TButton;
    btnOpen: TButton;
    OpenDialog1: TOpenDialog;
    PropertySet1: TPropertySet;
    procedure btnOpenClick(Sender: TObject);
    procedure btnCloseClick(Sender: TObject);
    procedure btnSetClick(Sender: TObject);
    procedure btnDeleteClick(Sender: TObject);
    procedure PropertySet1EnumProperties(Sender: TObject;
      PropertyName: string; PropertyID: Integer;
      PropertyVariant: tagPROPVARIANT);
    procedure listPropertiesClick(Sender: TObject);
  private
    { Private declarations }
    procedure LoadPropertyList;
  public
    { Public declarations }
  end;
var
  frmMain: TfrmMain;
implementation
{$R *.DFM}
procedure TfrmMain.LoadPropertyList;
begin
  listProperties.Items.Clear;
  PropertySet1.Enumerate;
end;
procedure TfrmMain.btnOpenClick(Sender: TObject);
begin
```

```
  if OpenDialog1.Execute then begin
    PropertySet1.Close;
    PropertySet1.FileName := OpenDialog1.FileName;
    PropertySet1.StreamGUID := FMTID_UserDefinedProperties;
    PropertySet1.Open;
    listProperties.Enabled := True;
    ecPropertyName.Enabled := True;
    ecValue.Enabled := True;
    btnSet.Enabled := True;
    btnDelete.Enabled := True;
    LoadPropertyList;
  end;
end;
procedure TfrmMain.btnCloseClick(Sender: TObject);
begin
  Close;
end;
procedure TfrmMain.listPropertiesClick(Sender: TObject);
var
  LI: TListItem;
begin
  LI := listProperties.Selected;
  if LI  nil then begin
    ecPropertyName.Text := LI.Caption;
    ecValue.Text := LI.SubItems[0];
  end;
end;
procedure TfrmMain.btnSetClick(Sender: TObject);
begin
  PropertySet1.SetProperty(ecPropertyName.Text, ecValue.Text);
  LoadPropertyList;
end;
procedure TfrmMain.btnDeleteClick(Sender: TObject);
begin
  PropertySet1.DeleteProperty(ecPropertyName.Text);
  LoadPropertyList;
end;
procedure TfrmMain.PropertySet1EnumProperties(Sender: TObject;
  PropertyName: string; PropertyID: Integer;
  PropertyVariant: tagPROPVARIANT);
var
  LI: TListItem;
begin
  if PropertyVariant.vt = VT_LPSTR then begin
    LI := listProperties.Items.Add;
    LI.Caption := PropertyName;
    LI.SubItems.Add(PropertyVariant.pszVal);
  end;
end;
end.
```

La Figura 8.3 muestra a *PropDemo* en tiempo de ejecución.

Figura 8.3: *PropDemo* gestiona propiedades definidas por el usuario.

Cuando se inicia la aplicación, no hay ningún fichero de datos abierto, razón por la que los controles de edición y los botones no están activos. Cuando el usuario abre un fichero de almacenamiento estructurado, el control de lista es rellenado con las propiedades del flujo *#5UserDefinedProperties*. Para este ejemplo, sólo he listado las propiedades de tipo *VT_LPSTR*, pero podrá fácilmente extender el ejemplo para manejar otros tipos de propiedades.

Si se selecciona una propiedad en el control de lista, los dos controles de edición serán actualizados para reflejar la propiedad seleccionada. Esto facilita la modificación del valor de la propiedad, introduciendo el nuevo valor y pulsando el botón *Set*, o la eliminación de la propiedad pulsando el botón *Delete*.

Como puede ver en el listado, el código de este ejemplo es muy simple. El trabajo de los botones *Set* y *Delete* se lleva a cabo mediante simples llamadas a los métodos apropiados del componente *TPropertySet*.

Ejemplo: Encontrando ficheros por sus propiedades

He aquí otro programa de ejemplo que ilustra otra característica muy útil de los conjuntos de propiedades: la posibilidad de ejecutar búsquedas de ficheros basados en las propiedades de éstos. Este ejemplo le permitirá buscar ficheros en su disco duro en busca de aquellos que tengan una propiedad específica.

La Figura 8.4 muestra la interfaz de usuario del programa de ejemplo *FindProp*.

Figura 8.4: *FindProp* permite que los usuarios encuentren ficheros por sus propiedades.

Introduzca la ruta de acceso a ficheros en el control de edición *File path*, y seleccione un flujo de propiedades en el cuadro de combinación *Stream*. Si selecciona el flujo *SummaryInformation* o bien *DocumentSummaryInformation*, deberá también seleccionar una propiedad predefinida en el cuadro de combinación *Property*. Si selecciona el flujo *UserDefinedProperties*, podrá introducir un nombre cualquiera en el cuadro *Property*.

Finalmente, introduzca el valor de la propiedad y pulse el botón *Find*. El programa recorrerá todos los ficheros que cumplan con la máscara de fichero especificada, mostrando los nombres de aquellos que presenten la propiedad y el valor indicados en el cuadro de lista inferior.

Observe que por limitaciones de espacio, encontrará a este ejemplo algunas limitaciones:

- La búsqueda no se extiende a subdirectorios.
- Sólo funciona para propiedades de los tipos *VT_LPSTR* y *VT_I4*.
- Las coincidencias deben ser exactas, con la excepción de que se ejecutan comparaciones de cadenas insensibles a diferencias entre mayúsculas y sus correspondientes minúsculas.

Estas limitaciones pueden ser resueltas fácilmente, y se proponen al lector como ejercicio.

Los Listados 8.5 y 8.6 contienen el código fuente del ejemplo *FindProp*.

Listado 8.5: FindProp - *FindProp.dpr*

```
program FindProp;
uses
  Forms,
  MainForm in 'MainForm.pas' {frmMain};
{$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(TfrmMain, frmMain);
  Application.Run;
end.
```

Listado 8.6: FindProp - *MainForm.pas*

```
unit MainForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,Dialogs,
  StdCtrls, ExtCtrls,
{$IFDEF VER100}
  PropSets,  // for Delphi 3
{$ENDIF}
  PropertySet, ActiveX;
type
  TfrmMain = class(TForm)
    pnlBottom: TPanel;
    pnlClient: TPanel;
    Label1: TLabel;
    cbStream: TComboBox;
    Label2: TLabel;
    cbPropertyName: TComboBox;
    Label3: TLabel;
    ecValue: TEdit;
    btnFind: TButton;
    btnClose: TButton;
    Label4: TLabel;
    lbFiles: TListBox;
    PropertySet1: TPropertySet;
    Label5: TLabel;
    ecFilePath: TEdit;
    procedure btnFindClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure btnCloseClick(Sender: TObject);
    procedure cbStreamClick(Sender: TObject);
  private
    function Matches(P: TPropVariant): Boolean;
    { Private declarations }
  public
    { Public declarations }
  end;
var
  frmMain: TfrmMain;
```

```
implementation
{$R *.DFM}
const
  StreamGUIDs: array[0 .. 2] of TGUID = (
    '{F29F85E0-4FF9-1068-AB91-08002B27B3D9}', // SummaryInformation
    '{D5CDD502-2E9C-101B-9397-08002B2CF9AE}', //
DocumentSummaryInformation
    '{D5CDD505-2E9C-101B-9397-08002B2CF9AE}'  // UserDefinedProperties
  );
procedure TfrmMain.FormCreate(Sender: TObject);
begin
  cbStream.ItemIndex := 0;
  cbStreamClick(Sender);
end;
procedure TfrmMain.cbStreamClick(Sender: TObject);
begin
  cbPropertyName.Items.Clear;
  case cbStream.ItemIndex of
    0: begin
      cbPropertyName.Style := csDropDownList;
      cbPropertyName.Items.AddObject('Title', TObject(PIDSI_TITLE));
      cbPropertyName.Items.AddObject('Subject', TObject(PIDSI_SUBJECT));
      cbPropertyName.Items.AddObject('Author', TObject(PIDSI_AUTHOR));
      cbPropertyName.Items.AddObject('Keywords', TObject(PIDSI_KEYWORDS));
      cbPropertyName.Items.AddObject('Comments', TObject(PIDSI_COMMENTS));
      cbPropertyName.Items.AddObject('Template', TObject(PIDSI_TEMPLATE));
      cbPropertyName.Items.AddObject('Last Author',
        TObject(PIDSI_LASTAUTHOR));
      cbPropertyName.Items.AddObject('Revision Number',
        TObject(PIDSI_REVNUMBER));
      cbPropertyName.Items.AddObject('Page Count',
TObject(PIDSI_PAGECOUNT));
      cbPropertyName.Items.AddObject('Word Count',
TObject(PIDSI_WORDCOUNT));
      cbPropertyName.Items.AddObject('Character Count',
        TObject(PIDSI_CHARCOUNT));
      cbPropertyName.Items.AddObject('Application Name',
        TObject(PIDSI_APPNAME));
      cbPropertyName.Items.AddObject('Document Security',
        TObject(PIDSI_DOC_SECURITY));
      cbPropertyName.ItemIndex := 0;
    end;
    1: begin
      cbPropertyName.Style := csDropDownList;
    end;
    2: begin
      cbPropertyName.Style := csDropDown;
    end;
  end;
end;
function TfrmMain.Matches(P: TPropVariant): Boolean;
begin
```

```
  case P.vt of
    VT_LPSTR:
      Result := UpperCase(P.pszVal) = UpperCase(ecValue.Text);
    VT_I4:
      Result := P.lVal = StrToInt(ecValue.Text);
    // Gestionar otros tipos de propiedades...
    else
      Result := False;
  end;
end;
procedure TfrmMain.btnFindClick(Sender: TObject);
var
  SR: TSearchRec;
  Res: Integer;
  P: TPropVariant;
  ID: Integer;
  FileName: WideString;
begin
  lbFiles.Items.Clear;
  Res := FindFirst(ecFilePath.Text, faReadOnly or faArchive, SR);
  if Res = 0 then
    try
      while Res = 0 do begin
        FileName := ExtractFilePath(ecFilePath.Text) + SR.Name;
        if StgIsStorageFile(PWideChar(FileName)) = S_OK then begin
          PropertySet1.FileName := FileName;
          PropertySet1.StreamGUID := StreamGuids[cbStream.ItemIndex];
          PropertySet1.Open;
          try
            if cbStream.ItemIndex = 2 then begin
              P := PropertySet1.GetPropertyByName(cbPropertyName.Text);
            end else begin
              ID := Integer(cbPropertyName.Items.Objects
                            [cbPropertyName.ItemIndex]);
              P := PropertySet1.GetPropertyByID(ID);
            end;
            if Matches(P) then
              lbFiles.Items.Add(FileName);
          finally
            PropertySet1.Close;
          end;
        end;
        Res := FindNext(SR);
      end;
    finally
      FindClose(SR);
    end;
  if lbFiles.Items.Count = 0 then
    ShowMessage('No matching files found.');
end;
procedure TfrmMain.btnCloseClick(Sender: TObject);
begin
```

```
    Close;
  end;
  end.
```

Ahora ya conoce cómo leer y escribir conjuntos de propiedades OLE en sus aplicaciones Delphi. Ha visto cómo las aplicaciones de Microsoft Office (al igual que muchas otras) utilizan conjuntos de propiedades para almacenar un resumen de los documentos. También he mostrado un ejemplo ingenioso de búsqueda de ficheros en base a los valores almacenados en conjuntos de propiedades.

En la siguiente sección examinaremos otro uso de común de los ficheros de almacenamiento estructurado, los contenedores OLE.

Documentos compuestos y contenedores OLE

En esta sección examinaremos documentos compuestos y los contenedores OLE. Un contenedor OLE es un programa que permite embeber uno o más servidores COM visuales dentro de sí. Por ejemplo, si ha utilizado alguna vez Microsoft Word, y ha integrado una imagen de Paint dentro de un documento de Word, entonces ha utilizado un contenedor OLE (vea Figura 8.5). Los documentos compuestos contienen datos en diferentes formatos. Los diferentes formatos de datos se almacenan tradicionalmente en diferentes flujos dentro de un mismo fichero (en otras palabras, ficheros de almacenamiento estructurado).

Figura 8.5: Word actuando como un contenedor OLE.

La creación de un contenedor OLE con Delphi es simple. Cree una nueva aplicación y en el formulario principal un componente *TOleContainer*. El componente *TOleContainer* puede encontrarse en la página *System* de la Paleta de Componentes.

En tiempo de diseño, puede hacer clic sobre el componente con el botón derecho del ratón, y seleccionar "*Insert Object*" en el menú de contexto. Se presentará el cuadro de diálogo *Insert Object*, según se muestra en la Figura 8.6.

Figura 8.6: Cuadro de diálogo *Insert Object*.

Si selección 'Imagen de Paintbrush' en la lista de tipos de objetos y pulsa OK, Windows arrancará una instancia de Paint en una ventana separada.

Nota

Delphi no soporta activación la *in-situ* en tiempo de diseño. Sin embargo, en tiempo de ejecución, Paint aparecerá en el lugar, como cabría esperar.

Si dibuja algo en Paint, y seguidamente selecciona *Archivo | Salir y volver a OleContainer1* en el menú de Paint, verá que la imagen aparece dentro del componente *TOleContainer* sobre el formulario principal (Figura 8.7).

Figura 8.7: Un contenedor OLE de Delphi con una imagen de Paint.

Activando y desactivando servidores OLE

El objetivo aquí es brindar un mecanismo mediante el cual el usuario pueda embeber objetos en tiempo de ejecución y no en tiempo de diseño. El siguiente fragmento de código muestra cómo hacerlo:

```
if OleContainer1.InsertObjectDialog then
  OleContainer1.DoVerb(OleContainer1.PrimaryVerb);
```

InsertObjectDialog presenta el cuadro de diálogo de Windows *Insert Object*. Si el usuario selecciona un tipo de objeto a insertar, la llamada devuelve *True*. Normalmente, cuando el usuario inserta un objeto desea inmediatamente comenzar a actuar sobre el objeto, por lo que en ese caso hago una llamada a *DoVerb* para ejecutar la acción por defecto sobre el objeto (normalmente, la acción por defecto consiste en activar el objeto).

La desactivación del servidor, lo cual devuelve el control a la aplicación contenedora, se produce automáticamente cuando el objeto pierde el foco. Si desea cerrar manualmente una aplicación servidora, puede ejecutar:

```
OleContainer1.Close;
```

Guardando y cargando objetos en ficheros y flujos

Aunque la aplicación servidora es responsable de operar sobre el objeto embebido, no es responsable de guardar el objeto en disco. Esto es responsabilidad de su aplicación. *TOleContainer* proporciona la capacidad de grabar objetos a un flujo o a un fichero. En este ejemplo guardaré el objeto en un fichero, pero en una aplicación compleja podría tener diversos objetos embebidos, y preferirá entonces grabarlos en un flujo.

No debemos dejar de mencionar el hecho de que, como los ficheros de almacenamiento estructurado soportan múltiples flujos, una manera obvia de guardar varios objetos en un único fichero consiste en utilizar almacenamiento estructurado, asignando un flujo separado a cada objeto.

Guardar un objeto OLE a disco es tan simple como escribir:

```
OleContainer1.SaveToFile('C:\MyFile.Doc');
```

Cargar el objeto es también muy sencillo:

```
OleContainer1.LoadFromFile('C:\MyFile.Doc');
```

Mezcla de menús

Como podrá haber notado si ha utilizado Microsoft Word, por ejemplo, como contenedor OLE, cuando se activa la aplicación servidora ésta pasa a controlar el menú principal del contenedor. Más exactamente, el servidor *mezcla* su menú con el menú del contenedor. Delphi soporta esta funcionalidad a través del uso de *TMenu.GroupIndex*.

Al asignar valores a la propiedad *GroupIndex* del menú principal, recuerde que los elementos de menú con valores pares de *GroupIndex* son conservados, mientras los elementos de menú con valores impares de *GroupIndex* son reemplazados por los elementos del menú correspondiente del servidor. Por ejemplo, en la aplicación de ejemplo *OLECont*, el menú *File* tiene un índice de grupo 0 (que se considera par), así que el menú *File* es mantenido. El menú *Edit* tiene un índice de grupo 1, así que éste es reemplazado por el menú *Edit* de Paint. Todos los restantes elementos de menú de la aplicación Paint se mezclan a continuación de la opción *Edit*.

Usando el Portapapeles

Además de cargar y guardar objetos en el disco, puede además copiar y pegar en el Portapapeles. Para copiar un objeto al Portapapeles, de forma que éste pueda ser pegado por ejemplo en otro contenedor OLE, puede efectuar la llamada:

```
OleContainer1.Copy;
```

Si desea implementar la operación de 'Cortar', debe destruir el objeto después de copiarlo, como sigue:

```
OleContainer1.Copy;
OleContainer1.DestroyObject;
```

Pegar desde el Portapapeles es igualmente sencillo. Sin embargo, debe verificar que el Portapapeles contiene algo que puede ser pegado. Las siguientes dos líneas de código muestran cómo se puede controlar esto:

```
if OleContainer1.CanPaste then
  OleContainer1.PasteSpecialDialog;
```

PasteSpecialDialog presenta un cuadro de diálogo como el que se muestra en la Figura 8.8, que permite al usuario determinar cómo pegar el objeto.

Figura 8.8: Pegando un objeto dentro de un contenedor OLE.

Ejemplo: Un contenedor OLE sencillo

En este ejemplo uniremos todas las piezas utilizadas en la discusión precedente para crear una aplicación contenedora OLE sencilla. Este ejemplo contiene unas pocas líneas de código, mas soporta la creación de un nuevo objeto, cargar y guardar objetos a disco, así como cortar, copiar y pegar al Portapapeles.

Los Listados 8.7 y 8.8 contienen el código fuente completo de la aplicación *OLECont*.

Listado 8.7: OLECont - *OLECont.dpr*

```
program OLECont;
uses
  Forms,
  MainForm in 'MainForm.pas' {Form1};
{$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
```

```
end.
```

Listado 8.8: OLECont - *MainForm.pas*

```
unit MainForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  Menus, OleCtnrs, StdCtrls, ExtCtrls;
type
  TForm1 = class(TForm)
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    Exit1: TMenuItem;
    New1: TMenuItem;
    Open1: TMenuItem;
    Save1: TMenuItem;
    N1: TMenuItem;
    OpenDialog1: TOpenDialog;
    Panel1: TPanel;
    OleContainer1: TOleContainer;
    Edit1: TMenuItem;
    Cut1: TMenuItem;
    Copy1: TMenuItem;
    Paste1: TMenuItem;
    InserTObject1: TMenuItem;
    N2: TMenuItem;
    N3: TMenuItem;
    Return1: TMenuItem;
    SaveDialog1: TSaveDialog;
    procedure New1Click(Sender: TObject);
    procedure Open1Click(Sender: TObject);
    procedure Save1Click(Sender: TObject);
    procedure Return1Click(Sender: TObject);
    procedure Exit1Click(Sender: TObject);
    procedure InsertObject1Click(Sender: TObject);
    procedure Cut1Click(Sender: TObject);
    procedure Copy1Click(Sender: TObject);
    procedure Paste1Click(Sender: TObject);
    procedure OleContainer1Activate(Sender: TObject);
    procedure OleContainer1Deactivate(Sender: TObject);
  private
    { Private declarations }
    FFileName: string;
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
```

```
procedure TForm1.Exit1Click(Sender: TObject);
begin
  Close;
end;
procedure TForm1.New1Click(Sender: TObject);
begin
  OleContainer1.DestroyObject;
end;
procedure TForm1.Open1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then begin
    FFileName := OpenDialog1.FileName;
    OleContainer1.LoadFromFile(FFileName);
  end;
end;
procedure TForm1.Save1Click(Sender: TObject);
begin
  if FFileName = '' then begin
    if SaveDialog1.Execute then
      FFileName := SaveDialog1.FileName
    else
      exit;
  end;
  OleContainer1.SaveToFile(FFileName);
end;
procedure TForm1.Return1Click(Sender: TObject);
begin
  OleContainer1.Close;
end;
procedure TForm1.InserTObject1Click(Sender: TObject);
begin
  if OleContainer1.InserTObjectDialog then
    OleContainer1.DoVerb(OleContainer1.PrimaryVerb);
end;
procedure TForm1.Cut1Click(Sender: TObject);
begin
  OleContainer1.Copy;
  OleContainer1.DestroyObject;
end;
procedure TForm1.Copy1Click(Sender: TObject);
begin
  OleContainer1.Copy;
end;
procedure TForm1.Paste1Click(Sender: TObject);
begin
  if OleContainer1.CanPaste then
    OleContainer1.PasteSpecialDialog;
end;
procedure TForm1.OleContainer1Activate(Sender: TObject);
begin
  Return1.Enabled := True;
end;
```

```
procedure TForm1.OleContainer1Deactivate(Sender: TObject);
begin
  Return1.Enabled := False;
end;
end.
```

La Figura 8.9 muestra a *OLECont* en tiempo de ejecución:

Figura 8.9: Un contenedor OLE escrito en Delphi en ejecución.

Resumen

He empleado una cantidad de tiempo considerable en este capítulo discutiendo los conjuntos de propiedades. Los conjuntos de propiedades son una característica sumamente útil y poderosa, que no ha ganado todavía el aprecio que merece fuera del entorno de las aplicaciones de Microsoft Office. Ahora debe tener toda la "munición" necesaria para almacenar conjuntos de propiedades dentro de sus propios ficheros. Por supuesto, para ello deberá hacer uso de ficheros de almacenamiento estructurado.

En la segunda parte del capítulo, he presentado los contenedores OLE. Como ha visto, es tremendamente sencillo crear un contenedor OLE en Delphi, gracias al componente *TOleContainer*.

9

Programación del *shell* de Windows

El capítulo final centra su atención en el desarrollo de extensiones del *shell* de Windows. El shell de Windows puede ser extendido mediante extensiones COM que implementen interfaces COM predefinidas.

En el Capítulo 2, "Interfaces y COM", hemos examinado los servidores COM internos al proceso. En este capítulo verá usos reales y concretos de servidores COM internos al proceso, incluyendo:

- Manejadores de menús contextuales
- Manejadores de "ganchos" (*hooks*) de copia
- Enlaces del *shell*
- Iconos de la bandeja (*tray icons*)
- Manejadores de Páginas de Propiedades

Además de reforzar los conceptos discutidos en los primeros capítulos del libro (en particular, todo lo relacionado con los servidores COM internos al proceso y el conteo de referencias a interfaces), este capítulo ofrece ejemplos y resultados prácticos que pueden resultar muy útiles tanto a Ud. como a sus usuarios.

Manejadores de menús contextuales

Los manejadores de menús contextuales son una de las extensiones del *shell* más simples que podemos desarrollar. En esta sección mostraremos cómo crear un manejador de menú contextual que añada el elemento de menú "*View Type Library*" (*Ver Biblioteca de Tipos*) al menú contextual que presenta Windows cuando el usuario pulsa el botón derecho sobre un fichero con una extensión .EXE, .DLL, .TLB o .OLB.

Nota

La mayoría de las bibliotecas de tipos están incluidas en ficheros .EXE, .DLL, o .TLB. Microsoft Office utiliza ficheros .OLB para sus bibliotecas de tipos.

La primera labor que deberá realizar cuando cree un manejador de menú contextual es crear el servidor COM. Para ello, seleccione *File* | *New* en el menú principal de Delphi. Pulse en la pestaña *ActiveX*, seleccione el icono *ActiveX Library*, y pulse OK.

A continuación, seleccione *File* | *New* en el menú de Delphi, y seleccione *COM Object* en la página *ActiveX*. Delphi presentará el Asistente para Objetos COM. Complételo según se muestra en la Figura 9.1.

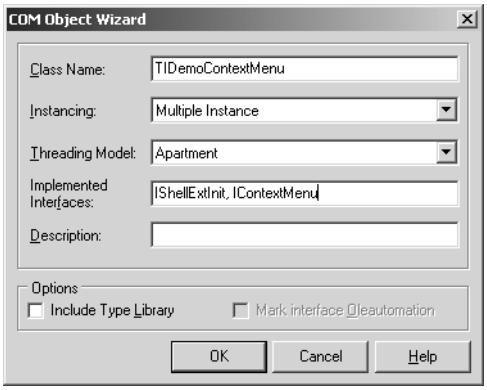


Figura 9.1: El Asistente de Objetos COM durante la generación de un manejador de menú contextual.

Cuando pulse OK, Delphi generará el código que se presenta en el Listado 9.1.

Nota

Recuerde que Delphi 3 no ofrece un Asistente para Objetos COM. Si está utilizando Delphi 3, deberá teclear manualmente el código que se muestra en el Listado 9.1.

Listado 9.1: Plantilla de código para un manejador de menú contextual

```
unit Unit1;
interface
uses
  Windows, ActiveX, ComObj;
type
 TTIDemoContextMenu = class(TComObject, IShellExtInit, IContextMenu)
 protected
  {Declare IShellExtInit methods here}
  {Declare IContextMenu methods here}
 end;
const
  Class_TIDemoContextMenu: TGUID =
'{66DA0B20-098C-11D3-B3DC-0040F67455FE}';
implementation
uses Comserv;
```

```
initialization
  TComObjectFactory.Create(ComServer, TTIDemoContextMenu,
    Class_TIDemoContextMenu, 'TIDemoContextMenu', '',
    ciMultiInstance, tmApartment);
end;
```

Guarde este fichero como *TIViewImpl.pas*. Deberá darse cuenta inmediatamente de que este objeto COM implementa dos interfaces: *IShellExtInit* e *IContextMenu*. Estas interfaces son definidas por Windows. Por lo tanto, no tenemos que declarar estas interfaces en la unidad, ya que Windows ya sabe de quiénes son. Sólo debe asegurarse de incluir *ShlObj* en la sección **uses** de la unidad.

La interfaz *IShellExtInit*

IShellExtInit es la responsable de inicializar el manejador de menú contextual. La interfaz *IShellExtInit* se define como:

```
const
  SID_IShellExtInit = '{000214E8-0000-0000-C000-000000000046}';
type
  IShellExtInit = interface(IUnknown)
    [SID_IShellextInit]
    function Initialize(pidlFolder: PItemIDList;
      lpdobj: IDataObject; hKeyProgID: HKEY): HResult; stdcall;
  end;
```

Como puede ver, se trata de una interfaz simple, pues sólo define un único método, *Initialize*. Por desgracia, el nombre de este método coincide con el de un método de la propia clase *TComObject*. En el Capítulo 1, "Las interfaces en Delphi", presenté las cláusulas de resolución de métodos. Esta es precisamente la situación por la cual se crearon las cláusulas de resolución de métodos. A través de una cláusula de resolución, podemos mapear *IShellExtInit.Initialize* a una función llamada *InitShellExtension*.

IShellExtInit.Initialize tiene tres parámetros. El primer parámetro, *pidlFolder*, describe la carpeta o directorio que contiene el o los ficheros sobre los cuales el usuario invoca el menú contextual. *Lpdobj* se utiliza para referenciar los nombres de los ficheros. El parámetro final., *hKeyProgID*, contiene un manejador a la clave *HKEY* en el Registro de Windows que describe el tipo de fichero seleccionado por el usuario.

La siguiente función, extraída del Listado 9.4, es una implementación típica de *IShellExtInit.Initialize*:

```
function TTIDemoContextMenu.InitShellExtension(pidlFolder: PItemIDList;
  lpdobj: IDataObject; hKeyProgID: HKEY): HResult; stdcall;
var
  Medium: TStgMedium;
  Format: TFormatEtc;
```

```
begin
  // Fallar si no hay objeto de datos
  if lpdobj = nil then begin
    Result := E_FAIL;
    Exit;
  end;
  // Rellenar la estructura TFormatEtc...
  with Format do begin
    cfFormat := CF_HDROP;
    ptd := nil;
    dwAspect := DVASPECT_CONTENT;
    lIndex := -1;
    tymed := TYMED_HGLOBAL;
  end;
  // Fallar si no podemos acceder a los datos
  Result := lpdobj.GetData(Format, Medium);
  if Failed(Result) then
    Exit;
  try
    // Asegurarse de que s□lo se ha seleccionado UN fichero
    if DragQueryFile(Medium.hGlobal, $FFFFFFFF, nil, 0) = 1 then begin
      SetLength(FFileName, MAX_PATH);
      DragQueryFile(Medium.hGlobal, 0, PChar(FFileName), MAX_PATH);
      Result := NOERROR;
    end else
      Result := E_FAIL;
  finally
    ReleaseStgMedium(Medium);
  end;
end;
```

Primero, debemos estar seguros de que el parámetro *lpdobj* que la función recibe es válido. Seguidamente se hace una llamada a *lpdobj.GetData* para recuperar los datos de *lpdobj*.

La llamada a *GetData* asigna memoria para el parámetro *Medium*, por lo que he encerrado el resto de la función en un bloque **try**/**finally** que asegure que esa memoria sea devuelta. La primera llamada a *DragQueryFile*, con un valor de *$FFFFFFFF*, devuelve el número de ficheros seleccionados por el usuario antes de invocar al menú contextual. Si no se ha seleccionado exactamente un fichero, la función devuelve el valor *E_FAIL*.

Si el usuario seleccionó exactamente un fichero, se hace otra llamada a *DragQueryFile* (esta vez con un valor de 0) para recuperar el nombre del fichero en cuestión. El nombre del fichero es almacenado en la variable *FFileName* para su uso futuro.

La interfaz *IContextMenu*

La interfaz *IContextMenu* es responsable de instalar y responder al menú contextual. *IContextMenu* está declarada en *ShlObj.pas* como:

```
const
  SID_IContextMenu = '{000214E4-0000-0000-C000-000000000046}';
type
  IContextMenu = interface(IUnknown)
    [SID_IContextMenu]
    function QueryContextMenu(Menu: HMENU;
      indexMenu, idCmdFirst, idCmdLast, uFlags: UINT): HResult; stdcall;
    function InvokeCommand(var lpici: TCMInvokeCommandInfo): HResult;
stdcall;
    function GetCommandString(idCmd, uType: UINT; pwReserved: PUINT;
      pszName: LPSTR; cchMax: UINT): HResult; stdcall;
  end;
```

Después de llamar a *IShellExtInit.Initialize*, Windows llama a *IContextMenu.QueryContextMenu*. Este método le ofrece la posibilidad de añadir elementos al menú contextual. *QueryContextMenu* tiene cinco parámetros: un manejador del menú contextual, el índice del menú a partir del cual Ud. insertará elementos al menú, los límites inferior y superior de *ID*s que pueden ser asignados a los nuevos elementos de menú, y una combinación de opciones que responden a patrones de bits. Las opciones válidas se enumeran en la Tabla 9.1.

Debe asignar al valor de retorno que devuelve *QueryContextMenu* la cantidad de elementos de menú que haya añadido.

Tabla 9.1: Valores válidos del parámetro *uFlags* de *QueryContextMenu*

Valor	Descripción
CMF_CANRENAME	Asignado si la aplicación soporta el cambio de nombres. Los manejadores de menús contextuales deben ignorar este valor.
CMF_DEFAULTONLY	Asignado cuando el usuario ha activado la opción por defecto del menú haciendo doble clic. Los manejadores de menús contextuales no deben añadir elementos de menú si este valor está asignado.
CMF_EXPLORE	Asignado si la vista de árbol de vista es visible en el Explorador de Windows. Los manejadores de menús contextuales deben ignorar este valor.
CMF_INCLUDESTATIC	Asignado en caso de que se esté creando un menú estático. Esta opción es utilizada únicamente por el navegador.
CMF_NODEFAULT	Asignado si no debe haber un elemento por defecto en el menú. Los manejadores de menús contextuales deben ignorar este valor.
CMF_NORMAL	Asignado en caso de operación normal. Un manejador de menú contextual debe añadir cualquier elemento de menú que soporte si este valor está asignado.

Valor	Descripción
CMF_NOVERBS	Asignado para los elementos mostrados en el menú '*Enviar A*'. Los manejadores de menús contextuales deben ignorar este valor.
CMF_VERBSONLY	Asignado si el menú es para un atajo de teclado. Los manejadores de menús contextuales deben ignorar este valor.

Como puede ver de la Tabla 9.1, la mayoría de los identificadores no tienen relación directa con los manejadores de menús contextuales (la interfaz *IContextMenu* es usada también en otras situaciones).

He aquí una implementación de *QueryContextMenu* que añade un elemento de menú al menú contextual. El texto del elemento de menú difiere en dependencia del contexto en el que *QueryContextMenu* es llamado.

```
function TTIDemoContextMenu.QueryContextMenu(Menu: HMENU;
  indexMenu, idCmdFirst, idCmdLast, uFlags: UINT): HResult;
begin
  Result := NOERROR;
  FViewMenuItem := indexMenu;
  if (uFlags and $000F) = CMF_NORMAL then begin
    InsertMenu(Menu, FViewMenuItem, MF_STRING or MF_BYPOSITION,
idCmdFirst,
      'View TypeLib...');
    Result := 1;
  end else if (uFlags and CMF_EXPLORE)  0 then begin
    InsertMenu(Menu, FViewMenuItem, MF_STRING or MF_BYPOSITION,
idCmdFirst,
      'View TypeLib from Explorer...');
    Result := 1;
  end else if (uFlags and CMF_VERBSONLY)  0 then begin
    InsertMenu(Menu, FViewMenuItem, MF_STRING or MF_BYPOSITION,
idCmdFirst,
      'View TypeLib from Shortcut...');
    Result := 1;
  end;
end;
```

QueryContextMenu compara el resultado de *uFlags and $000F* con *CMF_NORMAL*. Al aplicar la operación **and** con *$000F*, todos los valores de bits son limpiados, excepto *CMF_DEFAULTONLY*, *CMF_VERBSONLY*, *CMF_EXPLORE* y *CMF_NOVERBS*. *CMF_NORMAL* está definida en *ShlObj.pas* como 0, por lo que el valor *CMF_NORMAL* está implicado por la ausencia de *CMF_DEFAULTONLY*, *CMF_VERBSONLY*, *CMF_EXPLORE* y *CMF_NOVERBS*.

InsertMenu es una función del API de Windows que podría no serle familiar, a menos que haya trabajado con el API de Windows en Delphi u otro lenguaje (una

información detallada de como trabajar con el API de Windows desde Delphi, puede encontrarla en la serie de libros editados por Danypress "Los Tomos de Delphi - Núcleo del API Win32" y "API Gráfico Win32"). Esta función está definida en *Windows.pas* como:

```
function InsertMenu(hMenu: HMENU; uPosition, uFlags, uIDNewItem: UINT;
  lpNewItem: PChar): BOOL; stdcall;
```

El primer parámetro, *hMenu*, es el manejador del menú. Aquí podemos simplemente pasar el primer parámetro de *QueryContextMenu*. *UPosition* es un índice de base 0 dentro del menú, que indica dónde insertar el nuevo elemento de menú. Aquí podemos pasar el valor del parámetro *indexMenu* de *QueryContextMenu*.

UFlags es un conjunto de indicadores que determina cómo será insertado el nuevo elemento de menú. Normalmente se asigna a este parámetro *MF_STRING* o *MF_BYPOSITION*. *MF_STRING* indica que el parámetro final de *InsertMenu* es realmente una cadena de caracteres. *MF_BYPOSITION* indica que el parámetro *uPosition* es un índice de base 0 dentro del menú. Para conocer más sobre los otros posibles valores de este parámetro, consulte la documentación de Windows para *InsertMenu*.

UIDNewItem es el identificador numérico a asignar al elemento de menú. Por último, el parámetro final, *lpNewItem*, especifica el texto del elemento de menú en sí. Aquí normalmente se pasa el texto del nuevo elemento de menú.

El próximo método a conocer es *GetCommandString*. *GetCommandString* suministra un texto de ayuda (indicación - *hint* - en la terminología de Delphi) para los elementos que se añadan al menú contextual.

```
function TTIDemoContextMenu.GetCommandString(idCmd, uType: UINT;
  pwReserved: PUINT; pszName: LPSTR; cchMax: UINT): HResult;
begin
  Result := S_OK;
  if (idCmd = FViewMenuItem) and
     ((uType and GCS_HELPTEXT)  0) then
    StrLCopy(pszName, 'View selected type library', cchMax)
  else
    Result := E_INVALIDARG;
end;
```

GetCommandString tiene 4 parámetros útiles (*pwReserved* está reservado por Microsoft para usos futuros). El primer parámetro, *idCmd*, indica el índice de base 0 del elemento de menú para el cual se proporciona el texto de ayuda. *UType* indica el tipo de información que se brinda. Los valores válidos para este parámetro se listan en la Tabla 9.2. *PszName* apunta a un *buffer* en el que se almacena la cadena. El parámetro final, *cchMax*, indica el tamaño máximo del *buffer* en bytes.

Tabla 9.2: Valores válidos del parámetro *uType* de *GetCommandString*

Valor	Descripción
GCS_HELPTEXT	Recupera el texto de ayuda para el elemento del menú contextual.
GCS_VALIDATE	Verifica que el elemento de menú contextual existe.
GCS_VERB	Recupera el nombre de comando del elemento del menú contextual.

Finalmente, cuando el usuario selecciona el elemento de menú contextual, Windows llama a *IContextMenu.InvokeCommand*. *InvokeCommand* tiene un solo parámetro, *lpici*, de tipo *TCMInvokeCommandInfo*. *TCMInvokeCommandInfo* está declarado en *ShlObj.pas* de la siguiente forma:

```
type
  PCMInvokeCommandInfo = ^TCMInvokeCommandInfo;
  {$EXTERNALSYM _CMINVOKECOMMANDINFO}
  _CMINVOKECOMMANDINFO = record
    cbSize: DWORD;       { debe ser sizeof(CMINVOKECOMMANDINFO) }
    fMask: DWORD;        { cualquier combinaci□n de CMIC_MASK_* }
    hwnd: HWND;          { puede ser NULL (indica que no hay propietario)}
    lpVerb: LPCSTR;      { o una cadena de MAKEINTRESOURCE(idOffset)}
    lpParameters: LPCSTR;{ puede ser NULL (indica que no hay par□metros)}
    lpDirectory: LPCSTR; { puede ser NULL (indica que no hay directorio)}
    nShow: Integer;      { uno de los valores de ShowWindow() }
    dwHotKey: DWORD;
    hIcon: THandle;
  end;
  TCMInvokeCommandInfo = _CMINVOKECOMMANDINFO;
```

El campo de esta estructura que nos interesa principalmente es *lpVerb*. Contrario a lo que podría esperar, *lpVerb* es realmente un valor entero cuya palabra baja representa el índice de base 0 del elemento de menú que fue seleccionado. *Hwnd* proporciona un manejador de la ventana, que puede utilizarse para mostrar mensajes de error, como hago en *InvokeCommand*.

Observe que en *InvokeCommand*, he asumido inocentemente que *TIDemo.exe* está situado en algún lugar de la ruta de Windows. En una aplicación comercial, podría desear almacenar la ruta de la aplicación en el registro de Windows, donde el manejador de menú contextual pueda localizarla.

```
function TTIDemoContextMenu.InvokeCommand(
  var lpici: TCMInvokeCommandInfo): HResult;
var
  MI: Integer;
begin
  Result := S_OK;
  MI := Integer(lpici.lpVerb);
  If HiWord(MI)  0 then begin
```

```
      // La llamada se ha realizado desde otra aplicación
      // no soportado en este ejemplo
      Result := E_FAIL;
      exit;
    end;
    if LoWord(MI) = FViewMenuItem then
      // Ejecutar TIDemo
      ShellExecute(0, nil, 'TIDemo.exe', PChar(FFileName), nil, SW_SHOW)
    else
      Result := E_INVALIDARG;
  end;
```

Registrando el manejador de menú contextual

Después de escribir un manejador de menú contextual, se le debe registrar ante Windows, de modo que Windows sepa que está disponible. Los manejadores de menús contextuales están localizados bajo la clave *HKEY_CLASSES_ROOT\<extensión>\ShellEx\ContextMenuHandlers* en el Registro. La forma más sencilla de registrar y desregistrar un manejador de menú contextual es derivando una fábrica de clases personalizada de *TComObjectFactory*. Sólo deberá redefinir el procedimiento *UpdateRegistry* para registrar y desregistrar el manejador de menú contextual.

```
procedure TTIDemoFactory.UpdateRegistry(Register: Boolean);
begin
  inherited UpdateRegistry(Register);
  if Register then begin
    CreateRegKey('.tlb', '', 'TypeLibrary');
    CreateRegKey('TypeLibrary\ShellEx\ContextMenuHandlers\' +
      ClassName, '', GUIDToString(ClassID));
    CreateRegKey('ExeFiles\ShellEx\ContextMenuHandlers\' +
      ClassName, '', GUIDToString(ClassID));
  end
  else begin
    DeleteRegKey('.tlb');
    DeleteRegKey('TypeLibrary\ShellEx\ContextMenuHandlers\' +
      ClassName);
    DeleteRegKey('ExeFiles\ShellEx\ContextMenuHandlers\' +
      ClassName);
  end;
end;
```

La primera tarea que deberá realizar en este método es llamar al método heredado, para que el servidor COM sea registrado o desregistrado correctamente. Seguidamente, se debe añadir o eliminar la entrada del Registro específica del manejador de menú contextual.

Los Listados 9.2 al 9.4 muestran una implementación completa de un servidor COM para un manejador de menú contextual, junto con la fábrica de clase asociada.

Listado 9.2: CtxDemo - *CtxDemo.dpr*

```
library CtxDemo;
uses
  ComServ,
  TIViewImpl in 'TIViewImpl.pas',
  CtxDemo_TLB in 'CtxDemo_TLB.pas';
exports
  DllGetClassObject,
  DllCanUnloadNow,
  DllRegisterServer,
  DllUnregisterServer;
{$R *.TLB}
{$R *.RES}
begin
end.
```

Listado 9.3: CtxDemo - *CtxDemo_TLB.pas*

```
unit CtxDemo_TLB;
// ************************************************************************ //
// WARNING                                                                  //
// The types declared in this file were generated from data read from a //
// Type Library. If this type library is explicitly or indirectly (via  //
// another type library referring to this type library) re-imported, or //
// the 'Refresh' command of the Type Library Editor activated while     //
// editing the Type Library, the contents of this file will be          //
// regenerated and all manual modifications will be lost.               //
// ************************************************************************ //
// PASTLWTR : $Revision:   1.11.1.75  $
// File generated on 5/15/01 8:00:37 PM from Type Library described below.
// ************************************************************************ //
// Type Lib: C:\Book\samples\Chap9\CtxDemo\CtxDemo.tlb
// IID\LCID: {3577B511-0A4F-11D3-B3DE-0040F67455FE}\0
// Helpfile:
// HelpString: CtxDemo Library
// Version:    1.0
// ************************************************************************ //
interface
uses Windows, ActiveX, Classes, Graphics, OleCtrls, StdVCL;
// ************************************************************************//
// GUIDS declared in the TypeLibrary. Following prefixes are used:      //
//   Type Libraries     : LIBID_xxxx                                    //
//   CoClasses          : CLASS_xxxx                                    //
//   DISPInterfaces     : DIID_xxxx                                     //
//   Non-DISP interfaces: IID_xxxx                                      //
// ************************************************************************//
const
  LIBID_CtxDemo: TGUID = '{3577B511-0A4F-11D3-B3DE-0040F67455FE}';
implementation
uses ComObj;
end.
```

Listado 9.4: CtxDemo - *TIViewImpl.pas*

```
unit TIViewImpl;
interface
uses
  Windows, ActiveX, ComObj, ShlObj;
type
  TTIDemoContextMenu = class(TComObject, IShellExtInit, IContextMenu)
  private
    FFileName: string;
    FViewMenuItem: UINT;
    function IShellExtInit.Initialize = InitShellExtension;
  protected
    {Declare IShellExtInit methods here}
    function InitShellExtension(pidlFolder: PItemIDList;
      lpdobj: IDataObject; hKeyProgID: HKEY): HResult; stdcall;
    {Declare IContextMenu methods here}
    function QueryContextMenu(Menu: HMENU;
      indexMenu, idCmdFirst, idCmdLast, uFlags: UINT): HResult; stdcall;
    function GetCommandString(idCmd, uType: UINT; pwReserved: PUINT;
      pszName: LPSTR; cchMax: UINT): HResult; stdcall;
    function InvokeCommand(var lpici: TCMInvokeCommandInfo): HResult;
stdcall;
  end;
  TTIDemoFactory = class(TComObjectFactory)
  public
    procedure UpdateRegistry(Register: Boolean); override;
  end;
const
  Class_TIDemoContextMenu: TGUID =
'{66DA0B20-098C-11D3-B3DC-0040F67455FE}';
implementation
uses ComServ, ShellAPI, SysUtils;

function TTIDemoContextMenu.GetCommandString(idCmd, uType: UINT;
  pwReserved: PUINT; pszName: LPSTR; cchMax: UINT): HResult;
begin
  Result := S_OK;
  if (idCmd = FViewMenuItem) and
     ((uType and GCS_HELPTEXT)  0) then
    StrLCopy(pszName, 'View selected type library', cchMax)
  else
    Result := E_INVALIDARG;
end;

function TTIDemoContextMenu.InitShellExtension(pidlFolder: PItemIDList;
  lpdobj: IDataObject; hKeyProgID: HKEY): HResult; stdcall;
var
  Medium: TStgMedium;
  Format: TFormatEtc;
begin
  // Fallar si no hay objeto de datos
  if lpdobj = nil then begin
```

```
    Result := E_FAIL;
    Exit;
  end;
  // Preparar la estructura TFormatEtc...
  with Format do begin
    cfFormat := CF_HDROP;
    ptd := nil;
    dwAspect := DVASPECT_CONTENT;
    lIndex := -1;
    tymed := TYMED_HGLOBAL;
  end;
  // Fallar si no podemos acceder a los datos
  Result := lpdobj.GetData(Format, Medium);
  if Failed(Result) then
    Exit;
  try
    // Comprobar que un solo fichero ha sido seleccionado
    if DragQueryFile(Medium.hGlobal, $FFFFFFFF, nil, 0) = 1 then begin
      SetLength(FFileName, MAX_PATH);
      DragQueryFile(Medium.hGlobal, 0, PChar(FFileName), MAX_PATH);
      Result := NOERROR;
    end else
      Result := E_FAIL;
  finally
    ReleaseStgMedium(Medium);
  end;
end;

function TTIDemoContextMenu.InvokeCommand(
  var lpici: TCMInvokeCommandInfo): HResult;
var
  MI: Integer;
begin
  Result := S_OK;
  MI := Integer(lpici.lpVerb);
  if HiWord(MI)  0 then begin
    // Hemos sido llamados por otra aplicaci□n
    // No soportado en este ejemplo.
    Result := E_FAIL;
    Exit;
  end;
  if LoWord(MI) = FViewMenuItem then
    // Ejecutar TIDemo
    ShellExecute(0, nil, 'TIDemo.exe', PChar(FFileName), nil, SW_SHOW)
  else
    Result := E_INVALIDARG;
end;

function TTIDemoContextMenu.QueryContextMenu(Menu: HMENU; indexMenu,
  idCmdFirst, idCmdLast, uFlags: UINT): HResult;
begin
  Result := NOERROR;
```

```
  FViewMenuItem := indexMenu;
  if (uFlags and $000F) = CMF_NORMAL then begin
    InsertMenu(Menu, FViewMenuItem, MF_STRING or MF_BYPOSITION,
idCmdFirst,
      'View TypeLib...');
    Result := 1;
  end else if (uFlags and CMF_EXPLORE)  0 then begin
    InsertMenu(Menu, FViewMenuItem, MF_STRING or MF_BYPOSITION,
idCmdFirst,
      'View TypeLib from Explorer...');
    Result := 1;
  end else if (uFlags and CMF_VERBSONLY)  0 then begin
    InsertMenu(Menu, FViewMenuItem, MF_STRING or MF_BYPOSITION,
idCmdFirst,
      'View TypeLib from Shortcut...');
    Result := 1;
  end;
end;
{ TTIDemoFactory }

procedure TTIDemoFactory.UpdateRegistry(Register: Boolean);
begin
  inherited UpdateRegistry(Register);
  if Register then begin
    CreateRegKey('.tlb', '', 'TypeLibrary');
    CreateRegKey('TypeLibrary\ShellEx\ContextMenuHandlers\' +
      ClassName, '', GUIDToString(ClassID));
  end else begin
    DeleteRegKey('.tlb');
    DeleteRegKey('TypeLibrary\ShellEx\ContextMenuHandlers\' +
      ClassName);
  end;
end;
initialization
{$IFDEF VER100}
  TTIDemoFactory.Create(ComServer, TTIDemoContextMenu,
    Class_TIDemoContextMenu, 'TIDemoContextMenu', '', ciMultiInstance);
{$ELSE}
  TTIDemoFactory.Create(ComServer, TTIDemoContextMenu,
    Class_TIDemoContextMenu, 'TIDemoContextMenu', '', ciMultiInstance,
    tmApartment);
{$ENDIF}
end.
```

Las Figuras 9.2 a la 9.4 muestran tres escenarios diferentes en los que un menú contextual puede ser invocado. La Figura 9.2 muestra un menú contextual invocado desde el Explorador de Windows.

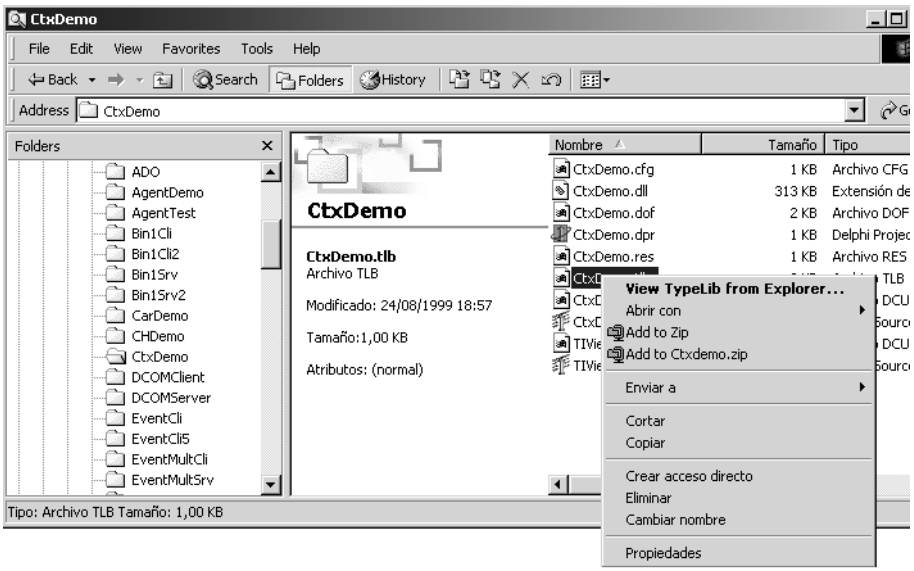


Figura 9.2: Menú contextual desde el Explorador de Windows.

La Figura 9.3 muestra un menú contextual invocado desde la ventana de diálogo *Archivo* | *Abrir* de Microsoft Word.

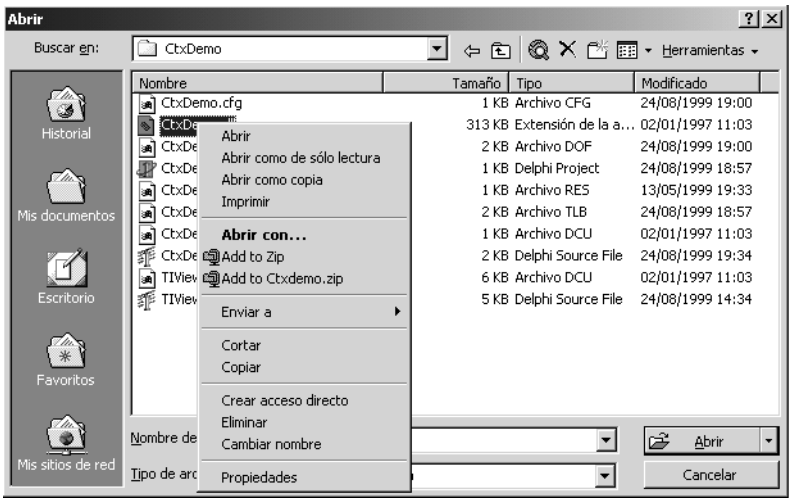


Figura 9.3: Menú contextual desde el diálogo *Archivo* | *Abrir*.

La Figura 9.4 muestra un menú contextual invocado desde el escritorio de Windows:

Figura 9.4: Menú contextual desde el Escritorio de Windows.

Ha visto cuán sencillo es crear un menú contextual en Delphi. En la próxima sección, le mostraré cómo crear manejadores de *ganchos* de copia.

Manejadores de *ganchos* de copia

De modo similar a los manejadores de menús contextuales, los manejadores de ganchos de copia (*copy hooks*) son servidores COM internos al proceso. Un *gancho* de copia puede enviarle una notificación cada vez que un usuario copie, renombre, mueva o elimine una carpeta. Sin embargo, no produce notificaciones en caso de operaciones sobre ficheros individuales. El manejador del gancho le ofrece la opción de denegar la operación, si lo cree conveniente.

Al igual que en el caso de los manejadores de menús contextuales, el primer paso en la construcción de un manejador de gancho de copia es crear un servidor COM. Para ello, seleccione *File | New* del menú principal de Delphi, pulse la pestaña *ActiveX*, en ella seleccione el icono *ActiveX Library* y pulse OK. Luego seleccione *File | New* del menú principal, y seleccione *COM Object* en la pestaña *ActiveX*. Delphi mostrará el Asistente para Objetos COM. Rellénelo como muestra la Figura 9.5.

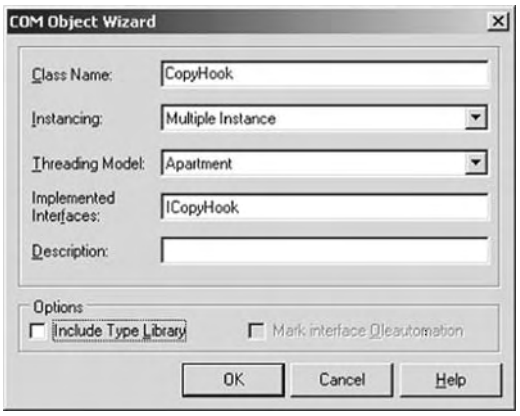


Figura 9.5: Asistente para Objetos COM para un manejador de gancho de copia.

Cuando pulse OK, Delphi generará el código que muestra el Listado 9.5 (si está utilizando Delphi3, tendrá que escribir el código).

Listado 9.5: Plantilla de código para el manejador de gancho de copia

```
unit Unit1;
interface
uses
  Windows, ActiveX, ComObj;
type
  TCopyHook = class(TComObject, ICopyHook)
  protected
    {Declare los m□todos I_CopyHook aqu□}
  end;
const
  Class_CopyHook: TGUID = '{EBF21F1D-653D-4EAE-A559-436AA3B7BABD}';
implementation
uses ComServ;
initialization
  TComObjectFactory.Create(ComServer, TCopyHook, Class_CopyHook,
    'CopyHook', '', ciMultiInstance, tmApartment);
end.
```

Guarde el fichero como *CopyHookImpl.pas*. Los manejadores de ganchos de copia implementan una interfaz simple: *ICopyHook*.

La interfaz *ICopyHook*

ICopyHook está definida en *ShlObj.pas* como:

```
type
  ICopyHookA = interface (IUnknown)
    [SID_IShellCopyHookA]
    function CopyCallback(Wnd: HWND; wFunc, wFlags: UINT;
```

```
        pszSrcFile: PAnsiChar; dwSrcAttribs: DWORD; pszDestFile: PAnsiChar;
        dwDestAttribs: DWORD): UINT; stdcall;
    end;
```

Como puede observar, la interfaz incluye un solo método. *CopyCallback* es el método que será llamado cuando el usuario intente realizar alguna de las operaciones descritas previamente.

CopyCallback tiene siete parámetros. El primer parámetro, *Wnd*, proporciona un manejador de ventana para mostrar mensajes de error. *WFunc* indica la operación que el usuario intenta realizar. Los valores válidos para *wFunc* se listan en la Tabla 9.3.

Nota

Los lectores familiarizados con la función del API *SHFileOperation* reconocerán los valores listados en las Tablas 9.3 y 9.4.

Tabla 9.3: Valores válidos para el parámetro *wFunc* de *CopyCallback*

Valor	Descripción
FO_COPY	Copia ficheros de la carpeta referenciada por *pszSrcFile* a la carpeta referenciada por *pszDestFile*.
FO_DELETE	Elimina ficheros de la carpeta referenciada por *pszSrcFile*.
FO_MOVE	Mueve ficheros de la carpeta referenciada por *pszSrcFile* a la carpeta referenciada por *pszDestFile*.
FO_RENAME	Renombra ficheros en la carpeta referenciada por *pszSrcFile*.

WFlags contiene una lista de identificadores asociados a patrones de bits, relacionados con la operación sobre ficheros. Los valores válidos para *wFlags* se listan en la Tabla 9.4.

Tabla 9.4: Valores válidos del parámetro *wFlags* de *CopyCallback*

Valor	Descripción
FOF_ALLOWUNDO	Guarda la información necesaria para deshacer la operación.
FOF_CONFIRMMOUSE	Todavía no implementada por Microsoft.
FOF_FILESONLY	Ejecuta la operación seleccionada sólo si el nombre de fichero especificado es *.*
FOF_MULTIDESTFILES	El parámetro *pszDestFile* especifica múltiples destinos. En lugar de una sola carpeta, este parámetro contiene un fichero de destino para cada fichero fuente.

Valor	Descripción
FOF_NOCONFIRMATION	No confirmar las operaciones sobre ficheros. Asume que el usuario seleccionó 'Sí a todo'.
FOF_NOCONFIRMMKDIR	No confirmar la creación de un nuevo directorio, para operaciones sobre ficheros que requieran la creación de un nuevo directorio.
FOF_NOCOPYSECURITYATTRIBS	No copiar los atributos de seguridad asociados a los fuentes (sólo Windows NT/2000).
FOF_NOERRORUI	No mostrar mensaje de error al usuario si ocurre un error.
FOF_RENAMEONCOLLISION	Si una operación para copiar, mover, o renombrar ficheros produce una duplicación de nombres, automáticamente se le dará al fichero un nuevo nombre.
FOF_SILENT	No mostrar un cuadro de diálogo de progreso durante las operaciones sobre ficheros.
FOF_SIMPLEPROGRESS	Mostrar un cuadro de diálogo de progreso, pero no mostrar los nombres de los ficheros individuales.

PszSrcFile contiene el nombre de la carpeta que va a ser manipulada. El parámetro siguiente, *dwSrcAttribs*, contiene los atributos de fichero de la carpeta en cuestión.

PszDestFile es el nombre de la carpeta de destino, si hay alguna. El parámetro final, *dwDestAttribs*, contiene los atributos de fichero de la carpeta de destino.

El valor devuelto debe ser *IDYES* para permitir la operación; *IDNO* para anular la operación, pero continuar procesando otras operaciones si las hubiera; o *IDCANCEL* para anular la operación y cancelar cualquier otra operación.

El Listado 9.6 muestra el código fuente de un manejador de gancho de copia funcional.

Listado 9.6: Manejador de gancho de copia

```
unit CopyHookImpl;
interface
uses
  Windows, ComObj, ShlObj, ShellApi;
type
  TCopyHook = class(TComObject, ICopyHook)
```

```
  protected
    {Declare ICopyHook methods here}
    function CopyCallback(Wnd: HWND; wFunc, wFlags: UINT;
      pszSrcFile: PAnsiChar; dwSrcAttribs: DWORD;
      pszDestFile: PAnsiChar; dwDestAttribs: DWORD): UINT; stdcall;
  end;
  TCHClassFactory = class(TComObjectFactory)
  public
    procedure UpdateRegistry(Register: Boolean); override;
  end;
const
  Class_CopyHook: TGUID = '{9090D650-0ACE-11D3-B3DF-00400F67455FE}';
implementation
uses ComServ;
{ TCopyHook }
function TCopyHook.CopyCallback(Wnd: HWND; wFunc, wFlags: UINT;
  pszSrcFile: PAnsiChar; dwSrcAttribs: DWORD; pszDestFile: PAnsiChar;
  dwDestAttribs: DWORD): UINT;
var
  OpString: string;
begin
  case wFunc of
    FO_MOVE: OpString := ' move ';
    FO_COPY: OpString := ' copy ';
    FO_DELETE: OpString := ' delete ';
    FO_RENAME: OpString := ' rename ';
  end;
  Result := MessageBox(Wnd, PChar('Are you sure you want to ' + OpString +
    pszSrcFile + '?'), 'CopyHookDemo', MB_YESNOCANCEL);
end;
{ TCHClassFactory }
procedure TCHClassFactory.UpdateRegistry(Register: Boolean);
begin
  inherited UpdateRegistry(Register);
  if Register then begin
    CreateRegKey('Directory\ShellEx\CopyHookHandlers\' + ClassName,
      '', GUIDToString(ClassID));
  end else begin
    DeleteRegKey('Directory\ShellEx\CopyHookHandlers\' + ClassName);
  end;
end;
initialization
{$IFDEF VER100}
  TCHClassFactory.Create(ComServer, TCopyHook, Class_CopyHook,
    'CopyHook', '', ciMultiInstance);
{$ELSE}
  TCHClassFactory.Create(ComServer, TCopyHook, Class_CopyHook,
    'CopyHook', '', ciMultiInstance, tmApartment);
{$ENDIF}
end.
```

Como puede ver, no es mucho el código involucrado. Esta implementación de *CopyCallback* simplemente verifica que el usuario sabe lo que está haciendo cuando mueve, copia, borra o renombra ficheros. Podría utilizar un manejador de gancho de copia, por ejemplo, para prevenir que los usuarios borren, muevan, o renombren carpetas ocultas.

Registrando un manejador de gancho de copia

De modo similar a como se registra un manejador de menú contextual, registrar un manejador de gancho de copia requiere la adición de entradas al Registro de Windows.

La forma más sencilla para registrar y desregistrar un manejador de gancho de copia es derivando una fábrica de clase personalizada de *TComObjectFactory*. Los Listados 9.7 al 9.9 muestran el código fuente completo del manejador de gancho de copia, junto con la clase *TCHClassFactory*, usada para registrar éste ante Windows.

Listado 9.7: CopyHookDemo - *CopyHookDemo.dpr*

```
library CopyHookDemo;
uses
  ComServ,
  CopyHookImpl in 'CopyHookImpl.pas',
  CopyHookDemo_TLB in 'CopyHookDemo_TLB.pas';
exports
  DllGetClassObject,
  DllCanUnloadNow,
  DllRegisterServer,
  DllUnregisterServer;
{$R *.TLB}
{$R *.RES}
begin
end.
```

Listado 9.8: CopyHookDemo - *CopyHookDemo_TLB.pas*

```
unit CopyHookDemo_TLB;
// ************************************************************************ //
// WARNING
// □□□-
// The types declared in this file were generated from data read from a
// Type Library. If this type library is explicitly or indirectly (via
// another type library referring to this type library) re-imported, or
// the 'Refresh' command of the Type Library Editor activated while
// editing the Type Library, the contents of this file will be
// regenerated and all manual modifications will be lost.
// ************************************************************************ //
// PASTLWTR : $Revision:   1.88  $
// File generated on 5/15/2001 7:51:50 PM from Type Library described
below.
// ************************************************************************//
```

```
// NOTE:
// Items guarded by $IFDEF_LIVE_SERVER_AT_DESIGN_TIME are used by
// properties which return objects that may need to be explicitly created
// via a function call prior to any access via the property. These items
// have been disabled in order to prevent accidental use from within the
// object inspector. You may enable them by defining
// LIVE_SERVER_AT_DESIGN_TIME or by selectively removing them from the
// $IFDEF blocks. However, such items must still be programmatically
// created via a method of the appropriate CoClass before they can be
// used.
// ************************************************************************ //
// Type Lib: C:\Book\samples\Chap9\CHDemo\CopyHookDemo.tlb (1)
// IID\LCID: {EE025D8F0-5A7E-11D3-B86C-00400F67455FE}\0
// Helpfile:
// DepndLst:
//   (1) v2.0 stdole, (C:\WINNT\System32\STDOLE2.TLB)
//   (2) v4.0 StdVCL, (C:\WINNT\System32\STDVCL40.DLL)
// ************************************************************************ //
{$TYPEDADDRESS OFF} // Unit must be compiled without type-checked
pointers.
interface
uses Windows, ActiveX, Classes, Graphics, OleServer, OleCtrls, StdVCL;
// ************************************************************************//
// GUIDS declared in the TypeLibrary. Following prefixes are used:
//   Type Libraries     : LIBID_xxxx
//   CoClasses          : CLASS_xxxx
//   DISPInterfaces     : DIID_xxxx
//   Non-DISP interfaces: IID_xxxx
// ************************************************************************//
const
  // TypeLibrary Major and minor versions
  CopyHookDemoMajorVersion = 1;
  CopyHookDemoMinorVersion = 0;
  LIBID_CopyHookDemo: TGUID = '{EE025D8F0-5A7E-11D3-B86C-00400F67455FE}';
implementation
uses ComObj;
end.
```

Listado 9.9: CopyHookDemo - *CopyHookImpl.pas*

```
unit CopyHookImpl;
interface
uses
  Windows, ComObj, ShlObj, ShellApi;
type
  TCopyHook = class(TComObject, ICopyHook)
  protected
    {Declare ICopyHook methods here}
    function CopyCallback(Wnd: HWND; wFunc, wFlags: UINT;
      pszSrcFile: PAnsiChar; dwSrcAttribs: DWORD;
      pszDestFile: PAnsiChar; dwDestAttribs: DWORD): UINT; stdcall;
  end;
```

```
  TCHClassFactory = class(TComObjectFactory)
  public
    procedure UpdateRegistry(Register: Boolean); override;
  end;
const
  Class_CopyHook: TGUID = '{9090D650-0ACE-11D3-B3DF-0040F67455FE}';
implementation
uses ComServ;
{ TCopyHook }
function TCopyHook.CopyCallback(Wnd: HWND; wFunc, wFlags: UINT;
  pszSrcFile: PAnsiChar; dwSrcAttribs: DWORD; pszDestFile: PAnsiChar;
  dwDestAttribs: DWORD): UINT;
var
  OpString: string;
begin
  case wFunc of
    FO_MOVE: OpString := ' move ';
    FO_COPY: OpString := ' copy ';
    FO_DELETE: OpString := ' delete ';
    FO_RENAME: OpString := ' rename ';
  end;
  Result := MessageBox(Wnd, PChar('Are you sure you want to ' + OpString +
    pszSrcFile + '?'), 'CopyHookDemo', MB_YESNOCANCEL);
end;
{ TCHClassFactory }
procedure TCHClassFactory.UpdateRegistry(Register: Boolean);
begin
  inherited UpdateRegistry(Register);
  if Register then begin
    CreateRegKey('Directory\ShellEx\CopyHookHandlers\' + ClassName,
      '', GUIDToString(ClassID));
  end else begin
    DeleteRegKey('Directory\ShellEx\CopyHookHandlers\' + ClassName);
  end;
end;
initialization
{$IFDEF VER100}
  TCHClassFactory.Create(ComServer, TCopyHook, Class_CopyHook,
    'CopyHook', '', ciMultiInstance);
{$ELSE}
  TCHClassFactory.Create(ComServer, TCopyHook, Class_CopyHook,
    'CopyHook', '', ciMultiInstance, tmApartment);
{$ENDIF}
end.
```

TCHClassFactory simplemente crea o elimina una entrada en el Registro de Windows. La Figura 9.6 muestra el manejador de gancho en acción.

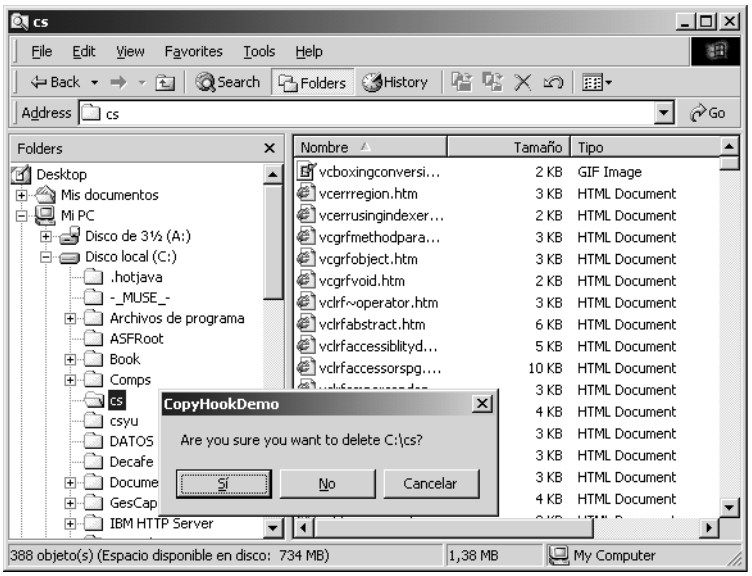


Figura 9.6: Intentando borrar el directorio *C:\cs*.

En esta sección hemos abordado la creación de manejadores de ganchos de copia. En las primeras dos secciones de este capítulo he creado servidores COM. En la siguiente, mostraré un programa que utiliza algunas de las interfaces predefinidas de Windows. En otras palabras, las interfaces que se describen en la siguiente sección ya han sido implementadas por Windows. Todo lo que se necesita hacer es realizar llamadas a las mismas para lograr el efecto deseado.

Accesos directos del shell

Si alguna vez ha instalado algún paquete de *software* para Windows, habrá visto los efectos de un acceso directo (enlace) del *shell*. Los accesos directos se utilizan al crear entradas en el menú Inicio, y también para crear los iconos de acceso a la aplicación en el escritorio.

Físicamente, un acceso directo es simplemente un fichero con extensión .LNK, almacenado en uno de los directorios estándar de Windows. Por ejemplo, para crear un acceso directo en el escritorio para el Administrador de un sistema Windows NT/2000, bastará con colocar un fichero .LNK en el directorio *C:\WINNT\PROFILES\Administrator\DESKTOP* (este directorio es diferente para un ordenador equipado con Windows 9x/ME). Sería trabajoso determinar manualmente la localización adecuada para un acceso directo. Windows proporciona la función *SHGetSpecialFolderLocation* para ofrecer esa información.

SHGetSpecialFolderLocation está definida en *ShlObj.pas* de la siguiente forma:

```
function SHGetSpecialFolderLocation(hwndOwner: HWND; nFolder: Integer;
  var ppidl: PItemIDList): HResult; stdcall;
```

El primer parámetro, *hwndOwner*, contiene un manejador de ventana que puede ser usado si Windows necesita mostrar un mensaje de error. *NFolder* especifica la carpeta que se desea localizar. Los valores válidos para este parámetro se listan en la Tabla 9.5. El último parámetro, *ppidl*, es un puntero a una lista de identificadores que especifican la localización de la carpeta relativa al escritorio. Después de obtener este valor, se debe utilizar *SHGetPathFromIDList* para convertirlo en una ruta absoluta, como se muestra en los siguientes ejemplos.

Tabla 9.5: Valores válidos del parámetro *nFolder* de *SHGetSpecialFolderLocation*

Valor	Descripción
CSIDL_ALTSTARTUP	Directorio que corresponde al grupo de programas de inicio no localizado del usuario.
CSIDL_APPDATA	Directorio que sirve como almacén común para datos específicos de las aplicaciones.
CSIDL_BITBUCKET	Directorio que almacena los ficheros de la Papelera de Reciclaje del usuario. La localización de este directorio no está en el Registro. Este directorio se marca con los atributos 'oculto' y 'de sistema' para evitar que el usuario lo mueva o borre.
CSIDL_COMMON_ALTSTARTUP	Directorio que corresponde al grupo de programas de inicio no localizado para todos los usuarios,
CSIDL_COMMON_DESKTOP DIRECTORY	Directorio que contiene los ficheros y carpetas que aparecen en el escritorio para todos los usuarios.
CSIDL_COMMON_FAVORITES	Directorio que sirve como almacén común para los elementos favoritos de todos los usuarios.
CSIDL_COMMON_PROGRAMS	Directorio que contiene los directorios de los grupos de programas comunes que aparecen en el menú Inicio de todos los usuarios.
CSIDL_COMMON_STARTMENU	Directorio que contiene los programas y carpetas que aparecen en el menú Inicio de todos los usuarios.
CSIDL_COMMON_STARTUP	Directorio que contiene los programas que aparecen en la carpeta Inicio de todos los usuarios.

Valor	Descripción
CSIDL_CONTROLS	Carpeta virtual que contiene los iconos de las aplicaciones de Panel de Control.
CSIDL_COOKIES	Directorio que se utiliza como almacén común de las *cookies* de Internet.
CSIDL_DESKTOP	El Escritorio de Windows - carpeta virtual que sirve como raíz del espacio de nombres.
CSIDL_DESKTOPDIRECTORY	Directorio que se utiliza para almacenar los ficheros que se muestran en el escritorio (no confundir con la carpeta del escritorio en sí).
CSIDL_DRIVES	Mi PC - carpeta virtual que contiene todos los elementos del ordenador local: unidades de disco, impresoras y el Panel de Control. La carpeta puede contener también unidades de red mapeadas.
CSIDL_FAVORITES	Directorio que se utiliza como almacén común para los elementos favoritos del usuario.
CSIDL_FONTS	Carpeta virtual que almacena las fuentes.
CSIDL_HISTORY	Directorio que se utiliza como almacén común para el historial de navegación por Internet.
CSIDL_INTERNET	Carpeta virtual que representa a Internet.
CSIDL_INTERNET_CACHE	Directorio que se utiliza como almacén común para los ficheros temporales de Internet.
CSIDL_NETHOOD	Directorio que almacena los objetos que aparecen en Entorno de Red.
CSIDL_NETWORK	Entorno de Red - carpeta virtual que representa el nivel superior de la jerarquía de red.
CSIDL__PERSONAL	Directorio que se utiliza como almacén común de documentos.
CSIDL_PRINTERS	Carpeta virtual que contiene las impresoras instaladas.
CSIDL_PRINTHOOD	Directorio que se utiliza como almacén común para los enlaces a las impresoras.
CSIDL_PROGRAMS	Directorio que contiene los grupos de programas del usuario (que a su vez son directorios).
CSIDL_RECENT	Directorio que almacena los documentos utilizados más recientemente por el usuario.

Valor	Descripción
CSIDL_SENDTO	Directorio que contiene los elementos del menú 'Enviar a...'.
CSIDL_STARTMENU	Directorio que almacena los elementos del menú Inicio.
CSIDL_STARTUP	Directorio que corresponde al grupo de programas de Inicio del usuario. El sistema ejecuta esos programas cada vez que el usuario inicia sesión en Windows NT/2000 o arranca Windows 9x/ME.
CSIDL_TEMPLATES	Directorio que se utiliza como almacén común para plantillas de documentos.

Interfaces de los enlaces del shell

Deberá hacer uso de dos interfaces COM para crear enlaces del *shell*: *IShellLink* e *IPersistFile*. *IShellLink* está definida en *ShlObj.pas* de la siguiente forma:

```
const
  SID_IShellLinkA = '{000214EE-0000-0000-C000-000000000046}';
type
  IShellLinkA = interface(IUnknown)
    [SID_IShellLinkA]
    function GetPath(pszFile: PAnsiChar; cchMaxPath: Integer;
      var pfd: TWin32FindData; fFlags: DWORD): HResult; stdcall;
    function GetIDList(var ppidl: PItemIDList): HResult; stdcall;
    function SetIDList(pidl: PItemIDList): HResult; stdcall;
    function GetDescription(pszName: PAnsiChar;
      cchMaxName: Integer): HResult; stdcall;
    function SetDescription(pszName: PAnsiChar): HResult; stdcall;
    function GetWorkingDirectory(pszDir: PAnsiChar;
      cchMaxPath: Integer): HResult; stdcall;
    function SetWorkingDirectory(pszDir: PAnsiChar): HResult; stdcall;
    function GetArguments(pszArgs: PAnsiChar;
      cchMaxPath: Integer): HResult; stdcall;
    function SetArguments(pszArgs: PAnsiChar): HResult; stdcall;
    function GetHotkey(var pwHotkey: Word): HResult; stdcall;
    function SetHotkey(wHotkey: Word): HResult; stdcall;
    function GetShowCmd(out piShowCmd: Integer): HResult; stdcall;
    function SetShowCmd(iShowCmd: Integer): HResult; stdcall;
    function GetIconLocation(pszIconPath: PAnsiChar; cchIconPath: Integer;
      out piIcon: Integer): HResult; stdcall;
    function SetIconLocation(pszIconPath: PAnsiChar;
      iIcon: Integer): HResult; stdcall;
    function SetRelativePath(pszPathRel: PAnsiChar;
      dwReserved: DWORD): HResult; stdcall;
    function Resolve(Wnd: HWND; fFlags: DWORD): HResult; stdcall;
    function SetPath(pszFile: PAnsiChar): HResult; stdcall;
  end;
```

IPersistFile está definida en *ActiveX.pas* como:

```
type
  IPersistFile = interface(IPersist)
    ['{0000010B-0000-0000-C000-000000000046}']
    function IsDirty: HResult; stdcall;
    function Load(pszFileName: POleStr; dwMode: Longint): HResult;
stdcall;
    function Save(pszFileName: POleStr; fRemember: BOOL): HResult;
stdcall;
    function SaveCompleted(pszFileName: POleStr): HResult; stdcall;
    function GetCurFile(out pszFileName: POleStr): HResult; stdcall;
  end;
```

Creando un enlace

Para crear un enlace a Microsoft Paint en la carpeta Inicio, deberá escribir el siguiente código:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  ShellLink: IShellLink;
  PersistFile: IPersistFile;
  ItemIDList : PItemIDList;
  Dir: array[0 .. MAX_PATH] of char;
begin
  ShellLink := CreateComObject(CLSID_ShellLink) as IShellLink;
  PersistFile := ShellLink as IPersistFile;
  if GetSystemDirectory(Dir, MAX_PATH) = 0 then
    Exit;
  OleCheck(ShellLink.SetPath(PChar(Dir + '\MsPaint.exe')));
  // Encontrar carpeta Inicio
  OleCheck(SHGetSpecialFolderLocation(0, CSIDL_STARTUP, ItemIDList));
  if SHGetPathFromIDList(ItemIDList, Dir) then
    OleCheck(PersistFile.Save(
     PWideChar(WideString(string(Dir) + '\MsPaint.lnk')),
     False));
end;
```

Analicemos por un momento qué hace procedimiento. Inicialmente se llama a *CreateComObject* para obtener una referencia a la interfaz *IShellLink*. Seguidamente, se "convierte" la interfaz *IShellLink* en una interfaz *IPersistFile*. Cualquier implementación de *IShellLink* debe también implementar *IPersistFile*, así que con total seguridad podemos obtener una interfaz *IPersistFile* a partir de una interfaz *IShellLink*.

GetSystemDirectory es una función del API que recupera el directorio de sistema de Windows (normalmente *C:\WINDOWS\SYSTEM* para Windows 9x/ME, y *C:\WINNT\SYSTEM32* para Windows NT/2000). Este es el directorio en el que se encuentra *MsPaint.exe*, asumiendo que fue situado en su lugar predeterminado

cuando se instaló Windows. *IShellLink.SetPath* le comunica al objeto COM dónde reside el fichero de destino.

A continuación se llama a *SHGetSpecialFolderLocation* para recuperar la carpeta de Inicio de Windows. Después de hacerlo, añado el nombre de fichero de enlace (*MsPaint.lnk*) a la carpeta. *IPersistFile.Save* recibe un parámetro de tipo *WideString*, por lo que convierto a *WideString*, y luego a *PWideChar* al llamar a *IPersistFile.Save*.

Existe una variedad de opciones que pueden configurarse cuando se crea un acceso directo, tales como el directorio de trabajo, el atajo de teclado, la descripción y los argumentos. En el ejemplo que se presenta en los Listados 9.10 al 9.12 muestra cómo utilizar *IShellLink* para establecer los atributos del enlace.

Aún cuando el código no es conceptualmente difícil, en ocasiones es difícil recordar la secuencia exacta de código necesaria para manipular accesos directos. En el ejemplo que sigue, crearé un conjunto de rutinas que podrá utilizar más adelante para crear, cargar y guardar un acceso directo. Una mejor alternativa sería crear un componente Delphi que encapsule ese trabajo. Existen varias bibliotecas de terceras partes que proporcionan componentes para crear accesos directos, añadir iconos a la bandeja de Windows, etc. De esas bibliotecas, la que me es más familiar es *SysTools*, distribuida por Danysoft *http://www.danyshop.com*.

Listado 9.10: LinkDemo - *LinkDemo.dpr*

```
program LinkDemo;
uses
  Forms,
  MainForm in 'MainForm.pas' {frmMain},
  NewLinkForm in 'NewLinkForm.pas' {frmNewShortcut};
{$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(TfrmMain, frmMain);
  Application.Run;
end.
```

Listado 9.11: LinkDemo - *MainForm.pas*

```
unit MainForm;
interface
uses
  SysUtils, Windows, Registry, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ActiveX, ShlObj, ExtCtrls, Menus,
  ShellAPI, ComCtrls, CommCtrl;
type
  TfrmMain = class(TForm)
    Panel1: TPanel;
    Panel2: TPanel;
```

```
    btnNew: TButton;
    btnOpen: TButton;
    btnSave: TButton;
    Label1: TLabel;
    Label2: TLabel;
    ecTarget: TEdit;
    btnFindTarget: TButton;
    Label3: TLabel;
    ecStartIn: TEdit;
    Label4: TLabel;
    Label5: TLabel;
    btnChangeIcon: TButton;
    cbRun: TComboBox;
    OpenTarget: TOpenDialog;
    Label6: TLabel;
    Label7: TLabel;
    ecDescription: TEdit;
    ecArguments: TEdit;
    OpenIcon: TOpenDialog;
    hkShortcut: THotKey;
    ecLinkName: TEdit;
    GroupBox1: TGroupBox;
    imgIcon: TImage;
    UpDown1: TUpDown;
    procedure btnNewClick(Sender: TObject);
    procedure btnFindTargetClick(Sender: TObject);
    procedure btnChangeIconClick(Sender: TObject);
    procedure UpDown1Click(Sender: TObject; Button: TUDBtnType);
    procedure btnSaveClick(Sender: TObject);
    procedure btnOpenClick(Sender: TObject);
    procedure ecTargetChange(Sender: TObject);
  private
    { Private declarations }
    FLinkFile: string;
    FLocation: Integer;
    FTarget: string;
    FDescription: string;
    FArguments: string;
    FIconFile: string;
    FIconIndex: Integer;
    FStartIn: string;
    FShortcut: Word;
    FRun: Integer;
    FLoading: Boolean;
    procedure SetFields;
    procedure CreateShortcut(const ALinkFile: WideString; AFolder:
Integer);
    procedure LoadShortcut(const ALinkFile: WideString);
    procedure SaveShortcut(const ALinkFile: WideString);
    procedure ShowIcon;
  public
    { Public declarations }
```

```
  end;
var
  frmMain: TfrmMain;
implementation
uses
  ComObj, NewLinkForm;
{$R *.DFM}
const
  RunCommands: array[0 .. 2] of Integer =
    (SW_SHOWNORMAL, SW_SHOWMINIMIZED, SW_SHOWMAXIMIZED);
procedure TfrmMain.CreateShortcut(const ALinkFile: WideString;
  AFolder: Integer);
var
  ShellLink: IShellLink;
  PersistFile: IPersistFile;
  ItemIDList : PItemIDList;
  Dir: array[0 .. MAX_PATH] of char;
begin
  ShellLink := CreateComObject(CLSID_ShellLink) as IShellLink;
  PersistFile := ShellLink as IPersistFile;
  // Encontrar carpeta Inicio
  OleCheck(SHGetSpecialFolderLocation(0, AFolder, ItemIDList));
  if SHGetPathFromIDList(ItemIDList, Dir) then begin
    FLinkFile := string(Dir) + '\' + ALinkFile + '.lnk';
    OleCheck(PersistFile.Save(PWideChar(WideString(FLinkFile)), True));
  end;
end;
procedure TfrmMain.LoadShortcut(const ALinkFile: WideString);
var
  ShellLink: IShellLink;
  PersistFile: IPersistFile;
  AStr: array[0 .. MAX_PATH] of char;
  FindData: TWin32FindData;
  ShowCmd: Integer;
begin
  FLoading := True;
  try
    ShellLink := CreateComObject(CLSID_ShellLink) as IShellLink;
    PersistFile := ShellLink as IPersistFile;
    OleCheck(PersistFile.Load(PWideChar(ALinkFile), STGM_READ));
    // Extraer informaci□n del acceso directo
    FLinkFile := ALinkFile;
    OleCheck(ShellLink.GetPath(AStr, MAX_PATH, FindData, SLGP_SHORTPATH));
    FTarget := AStr;
    OleCheck(ShellLink.GetArguments(AStr, MAX_PATH));
    FArguments := AStr;
    OleCheck(ShellLink.GetDescription(AStr, MAX_PATH));
    FDescription := AStr;
    OleCheck(ShellLink.GetWorkingDirectory(AStr, MAX_PATH));
    FStartIn := AStr;
    OleCheck(ShellLink.GetIconLocation(AStr, MAX_PATH, FIconIndex));
    FIconFile := AStr;
```

```
    if FIconFile = '' then
      FIconFile := FTarget;
    OleCheck(ShellLink.GetShowCmd(ShowCmd));
    FRun := ShowCmd - 1;
    OleCheck(ShellLink.GetHotKey(FShortcut));
    // Mostrar la informaci□n
    SetFields;
  finally
    FLoading := False;
  end;
end;
procedure TfrmMain.SaveShortcut(const ALinkFile: WideString);
var
  ShellLink: IShellLink;
  PersistFile: IPersistFile;
  HotKey: Word;
  HKMod: Byte;
begin
  ShellLink := CreateComObject(CLSID_ShellLink) as IShellLink;
  PersistFile := ShellLink as IPersistFile;
  OleCheck(PersistFile.Load(PWideChar(ALinkFile), STGM_SHARE_DENY_WRITE));
  // Asignar atajo de teclado
  HotKey := hkShortcut.HotKey;
  HKMod := 0;
  if hkCtrl in hkShortcut.Modifiers then
    HKMod := HKMod or HOTKEYF_CONTROL;
  if hkShift in hkShortcut.Modifiers then
    HKMod := HKMod or HOTKEYF_SHIFT;
  if hkAlt in hkShortcut.Modifiers then
    HKMod := HKMod or HOTKEYF_ALT;
  if hkExt in hkShortcut.Modifiers then
    HKMod := HKMod or HOTKEYF_EXT;
  HotKey := (HKMod shl 8) + HotKey;
  // Asignar par□metros
  OleCheck(ShellLink.SetPath(PChar(ecTarget.Text)));
  OleCheck(ShellLink.SetIconLocation(PChar(FIconFile), FIconIndex));
  OleCheck(ShellLink.SetDescription(PChar(ecDescription.Text)));
  OleCheck(ShellLink.SetWorkingDirectory(PChar(ecStartIn.Text)));
  OleCheck(ShellLink.SetArguments(PChar(ecArguments.Text)));
  OleCheck(ShellLink.SetHotkey(HotKey));
  OleCheck(ShellLink.SetShowCmd(RunCommands[cbRun.ItemIndex]));
  // Guardar acceso directo
  OleCheck(PersistFile.Save(PWideChar(ALinkFile), True));
end;
procedure TfrmMain.ShowIcon;
begin
  UpDown1.Max := ExtractIcon(hInstance, PChar(FIconFile),
    UINT(-1)) - 1;

  imgIcon.Picture.Icon.Handle := ExtractIcon(hInstance,
    PChar(FIconFile), FIconIndex);
end;
```

```
procedure TfrmMain.btnNewClick(Sender: TObject);
var
  frmNewShortcut: TfrmNewShortcut;
begin
  frmNewShortcut := TfrmNewShortcut.Create(nil);
  try
    if frmNewShortcut.ShowModal = mrOk then begin
      FLocation := frmNewShortcut.Folder;
      FLinkFile := frmNewShortcut.ecName.Text;
      CreateShortcut(FLinkFile, FLocation);
      // Activar campos y botones
      UpDown1.Enabled := True;
      ecTarget.Enabled := True;
      ecDescription.Enabled := True;
      ecArguments.Enabled := True;
      ecStartIn.Enabled := True;
      hkShortcut.Enabled := True;
      cbRun.Enabled := True;
      btnFindTarget.Enabled := True;
      btnChangeIcon.Enabled := True;
      btnSave.Enabled := True;
      // Limpiar campos
      FTarget := '';
      FDescription := '';
      FArguments := '';
      FIconFile := '';
      FIconIndex := 0;
      FStartIn := '';
      FShortcut := 0;
      FRun := 0;
      SetFields;
    end;
  finally
    frmNewShortcut.Free;
  end;
end;
procedure TfrmMain.SetFields;
var
  H: Byte;
  Modifier: THKModifiers;
begin
  ecLinkName.Text := FLinkFile;
  ecTarget.Text := FTarget;
  ecDescription.Text := FDescription;
  ecArguments.Text := FArguments;
  ecStartIn.Text := FStartIn;
  cbRun.ItemIndex := FRun;
  H := Hi(FShortcut);
  Modifier := [];
  if (H and HOTKEYF_ALT) = HOTKEYF_ALT then
    Include(Modifier, hkAlt);
  if (H and HOTKEYF_CONTROL) = HOTKEYF_CONTROL then
```

```
    Include(Modifier, hkCtrl);
  if (H and HOTKEYF_EXT) = HOTKEYF_EXT then
    Include(Modifier, hkExt);
  if (H and HOTKEYF_SHIFT) = HOTKEYF_SHIFT then
    Include(Modifier, hkShift);
  hkShortcut.HotKey := FShortcut;
  hkShortcut.Modifiers := Modifier;
  ShowIcon;
  ActiveControl := ecTarget;
end;
procedure TfrmMain.btnFindTargetClick(Sender: TObject);
begin
  if OpenTarget.Execute then begin
    ecTarget.Text := OpenTarget.FileName;
    FIconFile := ecTarget.Text;
    FIconIndex := 0;
    UpDown1.Max := ExtractIcon(hInstance, PChar(FIconFile),
      UINT(-1)) - 1;
    UpDown1.Position := 0;
    ShowIcon;
  end;
end;
procedure TfrmMain.btnChangeIconClick(Sender: TObject);
begin
  if OpenIcon.Execute then begin
    FIconFile := OpenIcon.FileName;
    FIconIndex := 0;
    UpDown1.Max := ExtractIcon(hInstance, PChar(FIconFile),
      UINT(-1)) - 1;
    UpDown1.Position := 0;
    ShowIcon;
  end;
end;
procedure TfrmMain.UpDown1Click(Sender: TObject; Button: TUDBtnType);
begin
  FIconIndex := UpDown1.Position;
  ShowIcon;
end;
procedure TfrmMain.btnSaveClick(Sender: TObject);
begin
  // Asegurar que se ha especificado un destino
  if ecTarget.Text = '' then begin
    ShowMessage('Please specify a target file before saving.');
    Exit;
  end;
  SaveShortcut(FLinkFile);
end;
procedure TfrmMain.btnOpenClick(Sender: TObject);
begin
  if OpenTarget.Execute then begin
    LoadShortcut(OpenTarget.FileName);
  end;
```

```
end;
procedure TfrmMain.ecTargetChange(Sender: TObject);
begin
  if not FLoading then begin
    FIconFile := ecTarget.Text;
    FIconIndex := 0;
    ShowIcon;
  end;
end;
end.
```

Listado 9.12: LinkDemo - *NewLinkForm.pas*

```
unit NewLinkForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ExtCtrls, ShlObj;
type
  TfrmNewShortcut = class(TForm)
    pnlBottom: TPanel;
    pnlClient: TPanel;
    btnOK: TButton;
    btnCancel: TButton;
    Label1: TLabel;
    cbLocation: TComboBox;
    Label2: TLabel;
    ecName: TEdit;
    procedure btnOKClick(Sender: TObject);
    procedure cbLocationClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    FFolder: Integer;
    { Private declarations }
  public
    { Public declarations }
    property Folder: Integer read FFolder;
  end;
implementation
type
  TFolderRec = record
    FolderName: string;
    FolderID: Integer;
  end;
const
{$IFDEF VER100}
  CSIDL_INTERNET = $0001;
  CSIDL_ALTSTARTUP = $001d;
  CSIDL_COMMON_ALTSTARTUP = $001e;
  CSIDL_COMMON_FAVORITES = $001f;
  CSIDL_INTERNET_CACHE = $0020;
  CSIDL_COOKIES = $0021;
```

```
  CSIDL_HISTORY = $0022;
{$ENDIF}
  MAX_FOLDERS = 30;
  Folders: array[0 .. MAX_FOLDERS - 1] of TFolderRec = (
    (FolderName: 'Alt Startup'; FolderID: CSIDL_ALTSTARTUP),
    (FolderName: 'App Data'; FolderID: CSIDL_APPDATA),
    (FolderName: 'Bit Bucket'; FolderID: CSIDL_BITBUCKET),
    (FolderName: 'Common Alt Startup'; FolderID: CSIDL_COMMON_ALTSTARTUP),
    (FolderName: 'Common Desktop Directory';
       FolderID: CSIDL_COMMON_DESKTOPDIRECTORY),
    (FolderName: 'Common Favorites'; FolderID: CSIDL_COMMON_FAVORITES),
    (FolderName: 'Common Programs'; FolderID: CSIDL_COMMON_PROGRAMS),
    (FolderName: 'Common Start Menu'; FolderID: CSIDL_COMMON_STARTMENU),
    (FolderName: 'Common Startup'; FolderID: CSIDL_COMMON_STARTUP),
    (FolderName: 'Controls'; FolderID: CSIDL_CONTROLS),
    (FolderName: 'Cookies'; FolderID: CSIDL_COOKIES),
    (FolderName: 'Desktop'; FolderID: CSIDL_DESKTOP),
    (FolderName: 'Desktop Directory'; FolderID: CSIDL_DESKTOPDIRECTORY),
    (FolderName: 'Drives'; FolderID: CSIDL_DRIVES),
    (FolderName: 'Favorites'; FolderID: CSIDL_FAVORITES),
    (FolderName: 'Fonts'; FolderID: CSIDL_FONTS),
    (FolderName: 'History'; FolderID: CSIDL_HISTORY),
    (FolderName: 'Internet'; FolderID: CSIDL_INTERNET),
    (FolderName: 'Internet Cache'; FolderID: CSIDL_INTERNET_CACHE),
    (FolderName: 'Network Neighborhood'; FolderID: CSIDL_NETHOOD),
    (FolderName: 'Network'; FolderID: CSIDL_NETWORK),
    (FolderName: 'Personal'; FolderID: CSIDL_PERSONAL),
    (FolderName: 'Printers'; FolderID: CSIDL_PRINTERS),
    (FolderName: 'Printer Neighborhood'; FolderID: CSIDL_PRINTHOOD),
    (FolderName: 'Programs'; FolderID: CSIDL_PROGRAMS),
    (FolderName: 'Recent Files'; FolderID: CSIDL_RECENT),
    (FolderName: 'Send To'; FolderID: CSIDL_SENDTO),
    (FolderName: 'Start Menu'; FolderID: CSIDL_STARTMENU),
    (FolderName: 'Startup'; FolderID: CSIDL_STARTUP),
    (FolderName: 'Templates'; FolderID: CSIDL_TEMPLATES)
  );
{$R *.DFM}
procedure TfrmNewShortcut.btnOKClick(Sender: TObject);
begin
  if ecName.Text = '' then begin
    ModalResult := mrNone;
    ShowMessage('Please enter a file name for the shortcut.');
  end;
end;
procedure TfrmNewShortcut.cbLocationClick(Sender: TObject);
begin
  FFolder := Folders[cbLocation.ItemIndex].FolderID;
end;
procedure TfrmNewShortcut.FormCreate(Sender: TObject);
var
  Index: Integer;
begin
```

```
    for Index := 0 to High(Folders) - 1 do
      cbLocation.Items.Add(Folders[Index].FolderName);
    cbLocation.ItemIndex := 0;
    FFolder := Folders[0].FolderID;
  end;
  end.
```

La Figura 9.7 muestra el programa *LinkDemo* en tiempo de ejecución:

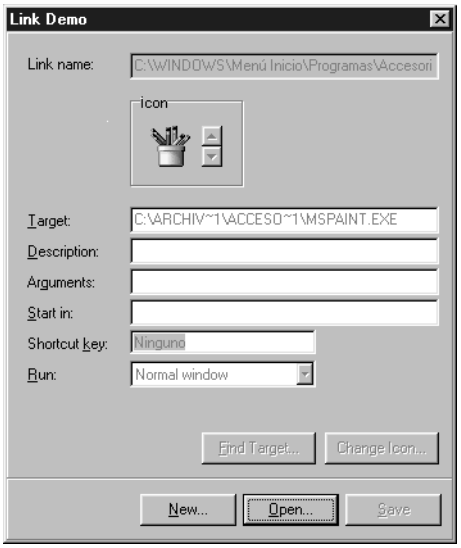


Figura 9.7: *LinkDemo* muestra las propiedades de un acceso directo situado en el Escritorio.

En esta sección he mostrado cómo utilizar el *shell* de Windows para crear accesos directos en el Escritorio, el menú Inicio u otras carpetas especiales del sistema. En la próxima sección veremos cómo trabajar los iconos de la bandeja.

Iconos de la bandeja

Al igual que la sección anterior, esta sección examina un fenómeno muy común del *shell* de Windows - los iconos de la bandeja (*tray icons*). Los iconos de la bandeja están situados por defecto en la parte inferior derecha del Escritorio de Windows. La Figura 9.8 muestra la bandeja de los iconos de mi ordenador.

Deberá conocer únicamente una función del API de Windows para trabajar con los iconos de la bandeja. Esta función es *Shell_NotifyIcon*, y está definida en *ShellApi.pas* de la siguiente forma:

```
function Shell_NotifyIcon(dwMessage: DWORD;
                  lpData: PNotifyIconData): BOOL; stdcall;
```

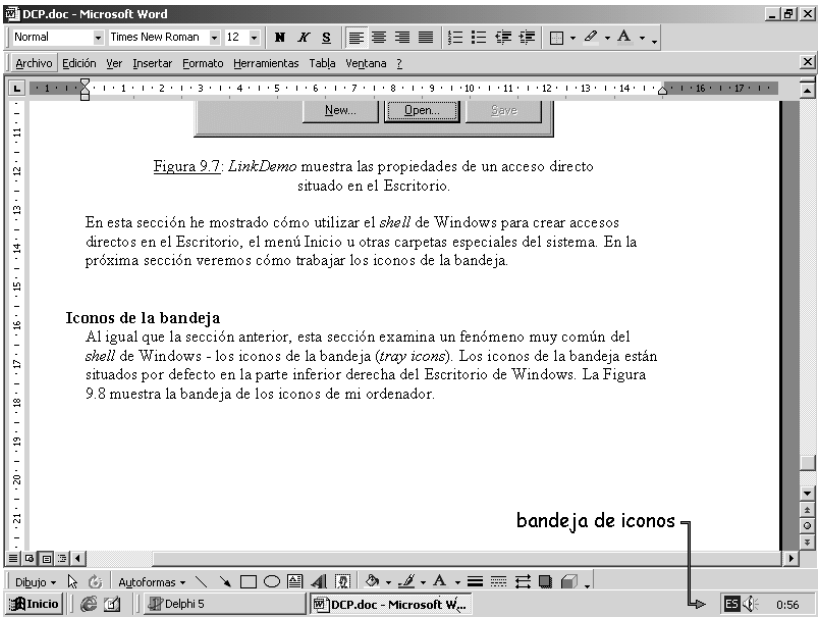


Figura 9.8: A la bandeja pueden añadirse nuevos iconos.

Shell_NotifyIcon tiene dos parámetros. El primero, *dwMessage*, indica la operación que se desea realizar. Los valores válidos para este parámetro se listan en la Tabla 9.6.

Tabla 9.6: Valores válidos del parámetro *dwMessage* de *Shell_NotifyIcon*

Valor	Descripción
NIM_ADD	Añade un icono a la bandeja.
NIM_DELETE	Elimina un icono de la bandeja.
NIM_MODIFY	Modifica un icono de la bandeja.

El otro parámetro, *lpData*, contiene la información de soporte necesaria para efectuar la operación. Es un puntero a un registro de tipo *TNotifyIconData*, definido en *ShellApi.pas* como:

```
type
  TNotifyIconData = record
    cbSize: DWORD;
    Wnd: HWND;
    uID: UINT;
    uFlags: UINT;
    uCallbackMessage: UINT;
    hIcon: HICON;
    szTip: array [0..63] of AnsiChar;
  end;
```

Como es usual en muchas de la nuevas APIs de Microsoft, *cbSize* indica el tamaño en bytes de la estructura *TNotifyIconData*. *Wnd* es el manejador de la ventana que recibirá mensajes de notificación relacionados con los iconos de bandeja (más información sobre este tema en esta misma página, en la sección "Mensajes de iconos de bandeja").

UID debe contener un identificador único que utilizará para referirse al icono de la barra de tareas. *UFlags* contendrá una combinación de identificadores que indicará cuáles de los otros campos del registro contienen datos válidos. Los valores primarios para el campo *uFlags* se listan en la Tabla 9.7.

Tabla 9.7: Valores válidos del campo *uFlags* de *TNotifyIconData*

Valor	Descripción
NIF_ICON	El campo *hIcon* es válido.
NIF_MESSAGE	El campo *uCallback* es válido.
NIF_TIP	El campo *szTip* es válido.

UCallbackMessage es un identificador de mensaje definido por la aplicación. Windows utilizará este identificador cuando envíe mensajes a la ventana especificada en *Wnd*. *HIcon* contiene un manejador del icono a añadir, eliminar o modificar. Por último, *szTip* contiene una cadena de caracteres que servirá como indicación textual (*tooltip*) del icono de bandeja.

Mensajes de iconos de bandeja

Como hemos mencionado hace un momento, los iconos de bandeja realmente no responden a ningún mensaje de Windows. Para que su aplicación detecte las acciones del usuario sobre los iconos de la bandeja, por ejemplo, deberá crear una ventana (oculta) cuyo solo propósito sea responder a mensajes en nombre de los iconos.

El acto de crear esta ventana de notificación no es particularmente difícil, pero es lo suficientemente engorroso como para no desear tener que recordar los detalles cada vez que utilice un icono de bandeja en sus aplicaciones. Por esa razón, en esta sección construiremos un componente Delphi que oculta todas esta complejidades.

El componente *TDCPTrayIcon*

El Listado 9.13 muestra el código fuente del componente *TDCPTrayIcon*. No explicaré el código línea por línea, pero comentaré varios puntos con más detalle.

En primer lugar, el método *Create* utiliza *AllocateHWnd* para crear el manejador de la ventana responsable de responder a los mensajes relacionados con el icono de bandeja.

El método *Loaded*, que es llamado una vez que el componente ha sido creado, asigna el icono de la aplicación al componente en caso de que no se haya asignado uno explícitamente.

NotificationWndProc es el procedimiento de ventana para la ventana creada por *AllocateHWnd*. Esta rutina intercepta los doble clics y los clics con el botón derecho sobre el icono de bandeja. Los doble clics son enviados a la aplicación a través del evento *OnDblClick*. Los clics con el botón derecho son utilizados para mostrar un menú contextual, en caso de que el usuario asigne uno a través de la propiedad *FPopupMenu* del componente.

SendTrayMessage es el método encargado de hacer llamadas a *Shell_NotifyIcon*. Este asigna automáticamente las opciones adecuadas para la llamada, en base a los valores de las propiedades del componente. Realizar todas las llamadas a *Shell_NotifyIcon* a través de un único método de acceso reduce las probabilidades de error. Si sólo hay un lugar en el que los parámetros pueden haber sido asignados incorrectamente, sólo habrá un lugar donde buscar en caso de que algo no funcione.

ShowInTaskbar y *RemoveFromTaskbar* muestran y eliminan, respectivamente, la aplicación de la barra de tareas. Cuando su aplicación esté minimizada (y visible en la bandeja), generalmente no deseará que el elemento asociado a la aplicación se muestre en la barra de tareas. *TDCPTrayIcon* realiza automáticamente llamadas a estos métodos para ocultar y mostrar la aplicación en la barra de tareas en los momentos apropiados.

ShowTrayIcon y *RemoveTrayIcon* muestran y eliminan, respectivamente, el icono de la bandeja. Estas funciones son llamadas automáticamente cuando el componente es creado y destruido, y también cuando la propiedad *Active* del componente cambia de valor.

Listado 9.13: *TrayIcon.pas*

```
unit TrayIcon;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, Menus, ShellAPI;
type
  TDCPTrayIcon = class(TComponent)
  private
    { Private declarations }
    FNotificationHandle: HWnd;
    FTrayIcon: TIcon;
    FShowHint: Boolean;
    FHint: string;
```

```
    FActive: Boolean;
    FCurrentlyActive: Boolean;
    FOnDblClick: TNotifyEvent;
    FPopupMenu: TPopupMenu;
    procedure NotificationWndProc(var Message : TMessage);
    procedure SetShowHint(const Value: Boolean);
    procedure SetHint(const Value: string);
    procedure SetTrayIcon(const Value: TIcon);
    procedure SetActive(const Value: Boolean);
    procedure SetPopupMenu(const Value: TPopupMenu);
  protected
    { Protected declarations }
    procedure DoDblClick;
    procedure Loaded; override;
    procedure OnAppMinimize(Sender: TObject);
    function SendTrayMessage(AMessage: DWORD): Boolean;
  public
    { Public declarations }
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    procedure RemoveTrayIcon;
    procedure RestoreApp;
    procedure ShowTrayIcon;
    procedure RemoveFromTaskbar;
    procedure ShowInTaskbar;
  published
    { Published declarations }
    property Active: Boolean read FActive write SetActive;
    property Hint: string read FHint write SetHint;
    property OnDblClick: TNotifyEvent read FOnDblClick write FOnDblClick;
    property PopupMenu: TPopupMenu read FPopupMenu write SetPopupMenu;
    property ShowHint: Boolean read FShowHint write SetShowHint;
    property TrayIcon: TIcon read FTrayIcon write SetTrayIcon;
  end;
procedure Register;
implementation
const
  ID_TRAYICON = 1;
  UWM_TRAYICON = WM_USER + 1;
procedure Register;
begin
  RegisterComponents('DCP', [TDCPTrayIcon]);
end;
{ TDCPTrayIcon }
constructor TDCPTrayIcon.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FNotificationHandle := AllocateHWnd(NotificationWndProc);
  FTrayIcon := TIcon.Create;
  Application.OnMinimize := OnAppMinimize;
end;
destructor TDCPTrayIcon.Destroy;
```

```
begin
  if FCurrentlyActive then
    RemoveTrayIcon;
  FTrayIcon.Free;
  if FNotificationHandle  0 then
    DeallocateHWnd(FNotificationHandle);
  inherited Destroy;
end;
procedure TDCPTrayIcon.DoDblClick;
begin
  if Assigned(OnDblClick) then
    OnDblClick(self);
end;
procedure TDCPTrayIcon.Loaded;
begin
  inherited Loaded;
  if not (csDesigning in ComponentState) then begin
    if FTrayIcon.Handle = 0 then
      FTrayIcon.Assign(Application.Icon);
  end;
  SetTrayIcon(FTrayIcon);
  if FActive then
    ShowTrayIcon;
end;
procedure TDCPTrayIcon.NotificationWndProc(var Message: TMessage);
var
  Pt: TPoint;
begin
  if Message.Msg = UWM_TRAYICON then begin
    case Message.lParam of
      WM_LBUTTONDBLCLK:
        DoDblClick;
      WM_RBUTTONDOWN:
        if Assigned(FPopupMenu) then begin
          SetForegroundWindow(FNotificationHandle);
          GetCursorPos(Pt);
          FPopupMenu.Popup(Pt.X, Pt.Y);
        end;
    end;
  end;
end;
procedure TDCPTrayIcon.OnAppMinimize(Sender: TObject);
begin
  if FCurrentlyActive then
    RemoveFromTaskbar;
end;
procedure TDCPTrayIcon.RemoveFromTaskbar;
begin
  if not (csDesigning in ComponentState) then begin
    ShowWindow(Application.Handle, SW_HIDE);
    ShowWindow(Application.MainForm.Handle, SW_HIDE);
  end;
```

```
end;
procedure TDCPTrayIcon.RemoveTrayIcon;
begin
  if FCurrentlyActive then
    if SendTrayMessage(NIM_DELETE) then begin
      FCurrentlyActive := False;
      ShowInTaskbar;
    end;
end;
procedure TDCPTrayIcon.RestoreApp;
begin
  ShowInTaskbar;
  SetForegroundWindow(Application.MainForm.Handle);
end;
function TDCPTrayIcon.SendTrayMessage(AMessage: DWORD): Boolean;
var
  Flags: Integer;
  NotifyIconData: TNotifyIconData;
begin
  Result := True;
  if csDesigning in ComponentState then
    Exit;
  Flags := NIF_MESSAGE or NIF_ICON;
  if FShowHint then
    Flags := Flags or NIF_TIP;
  FillChar(NotifyIconData, SizeOf(NotifyIconData), 0);
  with NotifyIconData do begin
    cbSize := SizeOf(TNotifyIconData);
    Wnd := FNotificationHandle;
    uID := ID_TRAYICON;
    uFlags := Flags;
    uCallBackMessage := UWM_TRAYICON;
    hIcon := FTrayIcon.Handle;
    StrLCopy(szTip, PChar(FHint), SizeOf(szTip));
  end;
  Result := Shell_NotifyIcon(AMessage, @NotifyIconData);
end;
procedure TDCPTrayIcon.SetActive(const Value: Boolean);
begin
  FActive := Value;
  if FActive then
    ShowTrayIcon
  else
    RemoveTrayIcon;
end;
procedure TDCPTrayIcon.SetHint(const Value: string);
begin
  FHint := Value;
  SendTrayMessage(NIM_MODIFY);
end;
procedure TDCPTrayIcon.SetPopupMenu(const Value: TPopupMenu);
begin
```

```
    FPopupMenu := Value;
  end;
  procedure TDCPTrayIcon.SetShowHint(const Value: Boolean);
  begin
    FShowHint := Value;
    SendTrayMessage(NIM_MODIFY);
  end;
  procedure TDCPTrayIcon.SetTrayIcon(const Value: TIcon);
  begin
    FTrayIcon.Assign(Value);
    SendTrayMessage(NIM_MODIFY);
  end;
  procedure TDCPTrayIcon.ShowInTaskbar;
  begin
    if not (csDesigning in ComponentState) then begin
      if Application.MainForm  nil then
        ShowWindow(Application.MainForm.Handle, SW_SHOWNORMAL);
      ShowWindow(Application.Handle, SW_SHOWNORMAL);
    end;
  end;
  procedure TDCPTrayIcon.ShowTrayIcon;
  begin
    if SendTrayMessage(NIM_ADD) then
      FCurrentlyActive := True;
  end;
  end.
```

Este componente cumple perfectamente su función, pero no va mucho más allá. Existen bibliotecas de terceros que incluyen componentes similares que ofrecen mucha más funcionalidad, como por ejemplo iconos de bandeja animados. Nuevamente, la biblioteca con la que estoy más familiarizado es *SysTools*, distribuida por Danysoft Internacional *http://www.danyshop.com* .

Un ejemplo de aplicación de bandeja

Ahora que ha visto cómo crear un componente para iconos de bandeja, le mostraré una aplicación sencilla que presenta un icono en la bandeja. Los Listados 9.14 y 9.15 muestran el código fuente de una aplicación que hace uso del componente *TDCPTrayIcon*.

Listado 9.14: TrayDemo - *TrayDemo.dpr*

```
program TrayDemo;
uses
  Forms,
  MainForm in 'MainForm.pas' {Form1};
{$R *.RES}
begin
```

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

Listado 9.15: TrayDemo - *MainForm.pas*

```
unit MainForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  TrayIcon, Menus, StdCtrls;
type
  TForm1 = class(TForm)
    DCPTrayIcon1: TDCPTrayIcon;
    PopupMenu1: TPopupMenu;
    Restore1: TMenuItem;
    Close1: TMenuItem;
    Label1: TLabel;
    procedure DCPTrayIcon1DblClick(Sender: TObject);
    procedure Close1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.DCPTrayIcon1DblClick(Sender: TObject);
begin
  DCPTrayIcon1.RestoreApp;
end;
procedure TForm1.Close1Click(Sender: TObject);
begin
  Close;
end;
end.
```

La Figura 9.9 muestra la aplicación *TrayDemo* en tiempo de ejecución.

Como ha visto en esta sección, escribir programas que hagan uso de la bandeja de iconos de Windows no es complicado. Con la creación de un componente Delphi, el código requerido por la aplicación cliente será sólo unas cuantas líneas.

En la próxima sección examinaremos una extensión del *shell* más complicada – los manejadores de páginas de propiedades.

Figura 9.9: *TDCPTrayIcon* puede heredar automáticamente el icono de la aplicación.

Manejadores de páginas de propiedades

Cuando hace clic con el botón derecho del ratón sobre un fichero cualquiera en el Explorador de Windows, verá siempre el elemento de menú '*Propiedades...*'. La selección de este elemento de menú permite ver las propiedades del fichero seleccionado. En muchos casos, todo lo que verá es la pestaña *General*, que muestra el nombre del fichero, su ubicación y atributos. La Figura 9.10 muestra el cuadro de diálogo de propiedades para un fichero que contiene una imagen de mapa de bits.

Para otros tipos de ficheros, tales como aplicaciones y DLLs, verá una página adicional que muestra la información de la versión almacenada en el fichero (Figura 9.11).

Otros tipos de ficheros podrán tener páginas de propiedades adicionales, mostrando otros tipos de información. ¿No sería estupendo si pudiera definir y mostrar páginas de propiedades personalizadas para los ficheros utilizados por su aplicación? Bueno, es posible. El proceso no es tan sencillo como en las aplicaciones del *shell* anteriores, pero si sigue cuidadosamente la guía mostrada, verá que no es tan difícil después de todo.

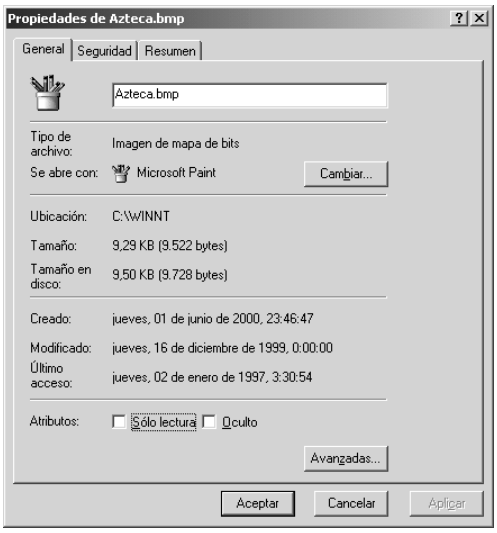


Figura 9.10: Cuadro de diálogo de propiedades estándar de Windows.

En el Capítulo 7, "Almacenamiento Estructurado", hemos creado una aplicación llamada *CarDemo* que almacena el nombre y modelo de un coche en un flujo independiente dentro del fichero de almacenamiento estructurado *CARS.CAR*. Mostraré cómo crear un manejador de página de propiedades para mostrar la información en una página separada del cuadro de diálogo de propiedades.

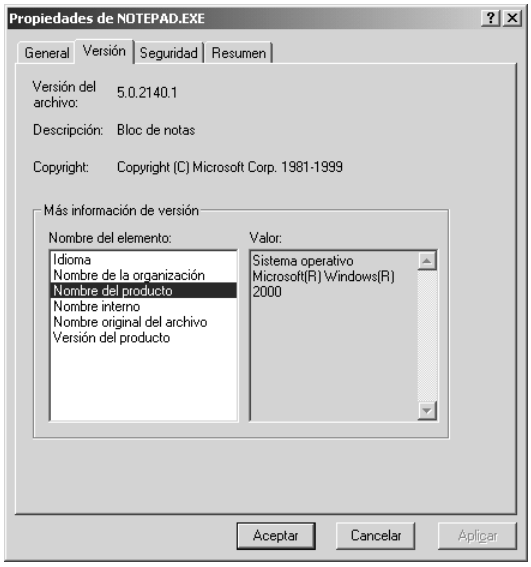


Figura 9.11: La información de versión se muestra en el diálogo de propiedades.

Creando una plantilla de diálogo

Lo primero que se debe hacer es crear una plantilla de diálogo para la nueva página de propiedades. Si le ha surgido alguna duda al leer esto, es que probablemente Delphi es su primer lenguaje de programación, y nunca ha visto una plantilla de diálogo. Si ha hecho otros trabajos con Visual C++ o Borland C++, seguramente conocerá qué es una plantilla de diálogo.

Delphi no trabaja con plantillas de diálogo. Esto hace la programación en Delphi más sencilla, pero desafortunadamente hace que este proyecto sea significativamente más complejo. Tiene tres opciones para crear una plantilla de página de propiedades:

- Puede usar otra herramienta de programación, por ejemplo Visual C++, para crear la plantilla de diálogo. Esto es lo que hemos hecho para este ejemplo.
- Si dispone de Delphi 5 o una versión posterior (u otra herramienta de Borland, como Delphi 1 RAD Pack o Borland C++), puede utilizar el *Resource Workshop* para crear la plantilla de diálogo.
- Si todo lo demás es imposible, puede utilizar cualquier editor de texto para crear un fichero .RC que contenga el código fuente de la plantilla de diálogo, y después usar la utilidad de línea de comandos *BRCC* para compilar el fichero .RC a una plantilla de diálogo. No recomiendo esta vía a menos que esté *realmente desesperado*, ya que es extremadamente difícil crear un cuadro de diálogo de un modo no visual. En el Listado 9.16 puede ver el código fuente del cuadro de diálogo del ejemplo.

Independientemente del método seleccionado, terminará obteniendo un fichero de extensión .RC que contiene el código fuente de una plantilla de diálogo. El Listado 9.16 muestra *PPage.RC*, que contiene una plantilla de diálogo creada con Visual C++.

Listado 9.16: *PPage.RC* - Plantilla de diálogo para *PropDemo.DLL*

```
#define IDD_CARPROPS                    106
#define IDC_VEHICLENAME                 1000
#define IDC_MODELYEAR                   1001
#define IDC_STATIC                      -1
IDD_CARPROPS DIALOG DISCARDABLE  0, 0, 210, 154
STYLE WS_CHILD | WS_VISIBLE | WS_CAPTION
CAPTION "Car Information"
FONT 8, "MS Sans Serif"
BEGIN
    LTEXT           "Vehicle name:",IDC_STATIC,7,9,52,8
    LTEXT           "Model year:",IDC_STATIC,7,24,52,8
    EDITTEXT        IDC_VEHICLENAME,75,7,128,12,ES_AUTOHSCROLL |
WS_DISABLED
    EDITTEXT        IDC_MODELYEAR,75,22,40,12,ES_AUTOHSCROLL | WS_DISABLED
END
```

Observe que el cuadro de diálogo se ha definido utilizando la combinación de estilos *WS_CHILD | WS_VISIBLE | WS_CAPTION*. El símbolo "|" es el operador **'or'** de manipulación de bits, como reconocerá si está familiarizado con C o C++.

Después de obtener el fichero .RC, puede crear un fichero .RES usando el siguiente comando:

```
BRCC32 PPage.rc
```

Esto producirá el fichero *PPage.RES*.

Interfaces de un manejador de página de propiedades

Los manejadores de páginas de propiedades deben implementar las interfaces *IShellExtInit* e *IShellPropSheetExt*. Como es usual, podemos usar el Asistente para Objetos COM de Delphi para crear el esqueleto de proyecto (los usuarios de Delphi 3 deberán introducir manualmente el código mostrado en el Listado 9.17). Desde el menú principal, seleccione *File | New*. En la página *ActiveX* del Depósito de Objetos, seleccione *ActiveX Library*. Seleccione nuevamente *File | New*, y sobre la página *ActiveX*, seleccione *COM Object*.

Rellene el Asistente para Objetos COM tal como muestra la Figura 9.12.

Cuando pulse OK, Delphi generará el código mostrado en el Listado 9.17. Guarde el fichero como *PropSheet.pas*.

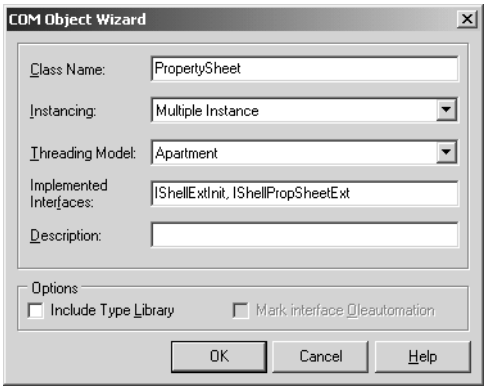


Figura 9.12: Asistente de Objetos COM para un manejador de páginas de propiedades.

Listado 9.17: Esqueleto generado por el Asistente de Objetos COM

```
unit PropSheet;
interface
uses
  Windows, ActiveX, Classes, ComObj;
type
```

```
  TPropertySheet = class(TComObject, IShellExtInit, IShellPropSheetExt)
  protected
    {Declare IShellExtInit methods here}
    {Declare IShellPropSheetExt methods here}
  end;
const
  Class_PropertySheet: TGUID = '{FCC2441E-A97F-430E-A7B8-AD1E66A0D4B0}';
implementation
uses ComServ;
initialization
  TComObjectFactory.Create(ComServer, TPropertySheet, Class_PropertySheet,
    'PropertySheet', '', ciMultiInstance, tmApartment);
end.
```

En este momento ya está familiarizado con la interfaz *IShellExtInit*. Por otra parte, *IShellPropSheetExt* está definida en *ShlObj.pas* como:

```
type
  IShellPropSheetExt = interface(IUnknown)
    [SID_IShellPropSheetExt]
    function AddPages(lpfnAddPage: TFNAddPropSheetPage;
      lParam: LPARAM): HResult; stdcall;
    function ReplacePage(uPageID: UINT; lpfnReplaceWith:
      TFNAddPropSheetPage; lParam: LPARAM): HResult; stdcall;
  end;
```

Esta interfaz define únicamente dos métodos: *AddPage* y *ReplacePage*. *AddPage* se utiliza para añadir una nueva página al diálogo de propiedades, mientras que *ReplacePage* se utiliza para reemplazar una página existente en el diálogo de propiedades. En este ejemplo no utilizaremos *ReplacePage* (aunque debemos proporcionar un manejador vacío, ya que todos los métodos de la interfaz deben ser implementados).

Los Listados 9.18 al 9.20 muestran el código fuente del manejador de página de propiedades después que las interfaces han sido implementadas. No examinaremos la implementación de la interfaz *IShellExtInit*, ya que la hemos analizado al comienzo del capítulo. Además, he implementado una clase personalizada descendiente de *TObjectFactory* para gestionar el registro y des-registro del servidor COM, como en proyectos anteriores.

Al inicio de la sección de implementación, verá las siguientes constantes:

```
const
  IDD_CARPROPS     = 106;
  IDC_VEHICLENAME  = 1000;
  IDC_MODELYEAR    = 1001;
```

Estos valores son copia de los definidos en la parte superior de *PPage.RC*, y se utilizan para identificar el cuadro de diálogo y los controles de edición situados en el mismo.

Debajo de las constantes, verá una línea que dice:

```
{$R PPAGE.RES}
```

Esto asegura que la plantilla de diálogo sea enlazada a la DLL generada por el proyecto.

Deteniéndonos en el método *AddPages*, he escrito un código bastante genérico para añadir la nueva página al diálogo de propiedades. Inicialmente, se rellena un registro de tipo *TPropSheetPage*. El campo *pszTemplate* especifica el recurso a partir del cual se creará la nueva página de propiedades. Note que a este campo le he asignado el valor *IDD_ORDPROPS*, el identificador asignado al diálogo en el fichero *PPage.RC*. El campo *pfnDlgProc* especifica el *procedimiento de diálogo* (muy similar a un procedimiento de ventana) que será utilizado para responder a los mensajes dirigidos al diálogo. Por su parte, *pfnCallback* apunta a una función de respuesta que será llamada cuando se produzcan ciertos eventos (por ejemplo, cuando la página de propiedades sea destruida). Note también que he asignado *self* a *psp.lParam*. Una vez que el control es transferido al procedimiento de diálogo, no tendré otra manera de referirme al objeto *TPropertySheet*, así que almaceno aquí un puntero a él.

Una vez que la página de propiedades ha sido creada gracias a las llamadas a *CreatePropertySheetPage* y *lpfnAddPage*, verá una llamada a *_AddRef*. En el Capítulo 1 he tratado el uso de *_AddRef* y *_Release*, y mencioné que rara vez necesitaría utilizar dichos métodos. Lo he hecho aquí porque sin ello el objeto *TPropertySheet* sería destruido antes del servidor COM (una vez que la página de propiedades ha sido creada, las únicas rutinas que se ejecutan son el procedimiento de diálogo y la función de respuesta, ninguna de las cuales es método de *TPropertySheet*). No obstante, he guardado el nombre del fichero seleccionado en el campo privado *FFileName* de *TPropertySheet*. La llamada a *_AddRef* garantiza que el objeto *TPropertySheet* no será destruido hasta que se haga una llamada a *_Release*).

Listado 9.18: PropDemo – *PropDemo.dpr*

```
library PropDemo;
uses
  Windows,
  ComServ,
  PropSheet in 'PropSheet.pas',
  PropDemo_TLB in 'PropDemo_TLB.pas';
exports
  DllGetClassObject,
  DllCanUnloadNow,
  DllRegisterServer,
  DllUnregisterServer;
{$R *.TLB}
{$R *.RES}
```

```
end.
```

Listado 9.19: PropDemo - *PropDemo_TLB.pas*

```
unit PropDemo_TLB;
// ************************************************************************ //
// WARNING                                                                  //
// □□□-                                                                     //
// The types declared in this file were generated from data read from a     //
// Type Library. If this type library is explicitly or indirectly (via      //
// another type library referring to this type library) re-imported, or     //
// the 'Refresh' command of the Type Library Editor activated while         //
// editing the Type Library, the contents of this file will be              //
// regenerated and all manual modifications will be lost.                   //
// ************************************************************************ //
// PASTLWTR : $Revision:   1.11.1.75  $
// File generated on 5/15/01 10:25:44 AM from Type Library described
below.
// ************************************************************************ //
// Type Lib: C:\Book\samples\Chap9\PropDemo\PropDemo.tlb
// IID\LCID: {D01C8F62-264B-11D3-B7FE-0040F67455FE}\0
// Helpfile:
// HelpString: PropDemo Library
// Version:    1.0
// ************************************************************************ //
interface
uses Windows, ActiveX, Classes, Graphics, OleCtrls, StdVCL;
// ************************************************************************//
// GUIDS declared in the TypeLibrary. Following prefixes are used:          //
//   Type Libraries     : LIBID_xxxx                                        //
//   CoClasses          : CLASS_xxxx                                        //
//   DISPInterfaces     : DIID_xxxx                                         //
//   Non-DISP interfaces: IID_xxxx                                          //
// ************************************************************************//
const
  LIBID_PropDemo: TGUID = '{D01C8F62-264B-11D3-B7FE-0040F67455FE}';
implementation
uses ComObj;
end.
```

Listado 9.20: PropDemo - *PropSheet.pas*

```
unit PropSheet;
interface
uses
  Windows, Messages, ActiveX, ComObj, CommCtrl, ShlObj;
type
  TPropertySheet = class(TComObject, IShellExtInit, IShellPropSheetExt)
  private
    FFileName: WideString;
    function IShellExtInit.Initialize = InitShellExtension;
  protected
```

```
    {Declare IShellExtInit methods here}
    function InitShellExtension(pidlFolder: PItemIDList;
      lpdobj: IDataObject; hKeyProgID: HKEY): HResult; stdcall;
    {Declare IShellPropSheetExt methods here}
    function AddPages(lpfnAddPage: TFNAddPropSheetPage;
      lParam: LPARAM): HResult; stdcall;
    function ReplacePage(uPageID: UINT; lpfnReplaceWith:
      TFNAddPropSheetPage; lParam: LPARAM): HResult; stdcall;
  end;
  TPropertySheetFactory = class(TComObjectFactory)
  public
    procedure UpdateRegistry(Register: Boolean); override;
  end;
const
  Class_PropertySheet: TGUID = '{DD1C8F61-264B-11D3-B7FE-0040F67455FE}';
implementation
uses Classes, SysUtils, ComServ, ShellAPI, AXCtrls;
const
  IDD_CARPROPS     = 106;
  IDC_VEHICLENAME  = 1000;
  IDC_MODELYEAR    = 1001;
{$R PPAGE.RES}
function PropertySheetDlgProc(hDlg: HWND; uMessage: UINT;
  wParam: WPARAM; lParam: LPARAM): Boolean; stdcall;
var
  psp: PPropSheetPage;
  ps: TPropertySheet;
  stgRoot: IStorage;
  stmInfo: IStream;
  Res: HResult;
  OS: TOleStream;
  Reader: TReader;
  VehicleName: string;
  ModelYear: Integer;
begin
  Result := True;
  case uMessage of
    WM_INITDIALOG: begin
      psp := PPropSheetPage(lParam);
      ps := TPropertySheet(psp.lParam);
      Res := StgOpenStorage(PWideChar(ps.FFileName), nil, STGM_READ or
        STGM_SHARE_EXCLUSIVE, nil, 0, stgRoot);
      if SUCCEEDED(Res) then begin
        Res := stgRoot.OpenStream('Info', nil, STGM_READ or
          STGM_SHARE_EXCLUSIVE, 0, stmInfo);
        if SUCCEEDED(Res) then begin
          OS := TOleStream.Create(stmInfo);
          try
            Reader := TReader.Create(OS, 1024);
            try
              VehicleName := Reader.ReadString;
              ModelYear := Reader.ReadInteger;
```

```
            finally
              Reader.Free;
            end;
          finally
            OS.Free;
          end;
          SetDlgItemText(hDlg, IDC_VEHICLENAME, PChar(VehicleName));
          SetDlgItemText(hDlg, IDC_MODELYEAR, PChar(IntToStr(ModelYear)));
        end;
      end;
    end;
    WM_COMMAND: begin
      // Si hay botones en el di□logo, deben gestionarse aqu□□
    end;
    WM_NOTIFY: begin
      case PNMHDR(lParam).code of
        PSN_APPLY: begin
          // El usuario ha pulsado OK o el bot□n 'Aplicar'.
        end;
      end;
    end;
    else
      Result := False;
  end;
end;
function PropertySheetCallback(hWnd: HWND; uMessage: UINT;
  var psp: TPropSheetPage): UINT; stdcall;
begin
  case uMessage of
    PSPCB_RELEASE:
      if psp.lParam  0 then
        TPropertySheet(psp.lParam)._Release;
  end;
  Result := 1;
end;
{ TPropertySheet }
function TPropertySheet.InitShellExtension(pidlFolder: PItemIDList;
  lpdobj: IDataObject; hKeyProgID: HKEY): HResult; stdcall;
var
  Medium: TStgMedium;
  Format: TFormatEtc;
  FileName: string;
begin
  // Fallar si no hay objeto de datos
  if lpdobj = nil then begin
    Result := E_FAIL;
    exit;
  end;
  // Preparar estructura TFormatEtc...
  with Format do begin
    cfFormat := CF_HDROP;
    ptd := nil;
```

```
    dwAspect := DVASPECT_CONTENT;
    lIndex := -1;
    tymed := TYMED_HGLOBAL;
  end;
  // Fallar si no podemos acceder a los datos
  Result := lpdobj.GetData(Format, Medium);
  if Failed(Result) then
    exit;
  try
    // Asegurarse de que un solo fichero est seleccionado
    if DragQueryFile(Medium.hGlobal, $FFFFFFFF, nil, 0) = 1 then begin
      SetLength(FileName, MAX_PATH);
      DragQueryFile(Medium.hGlobal, 0, PChar(FileName), MAX_PATH);
      FFileName := FileName;
      Result := NOERROR;
    end else
      Result := E_FAIL;
  finally
    ReleaseStgMedium(Medium);
  end;
end;
function TPropertySheet.AddPages(lpfnAddPage: TFNAddPropSheetPage;
  lParam: LPARAM): HResult;
var
  psp: TPropSheetPage;
  hPage: HPropSheetPage;
begin
  FillChar(psp, sizeof(psp), 0);
  psp.dwSize := sizeof(psp);
  psp.dwFlags := PSP_USETITLE or PSP_USECALLBACK;
  psp.hInstance := hInstance;
  psp.pszTemplate := MAKEINTRESOURCE(IDD_CARPROPS);
  psp.pszTitle := 'Car Information';
  psp.pfnDlgProc := @PropertySheetDlgProc;
  psp.pfnCallback := @PropertySheetCallback;
  psp.lParam := Integer(self);
  hPage := CreatePropertySheetPage(psp);
  if hPage  nil then begin
    if not lpfnAddPage(hPage, lParam) then
      DestroyPropertySheetPage(hPage);
  end;
  _AddRef;
  Result := NOERROR;
end;
function TPropertySheet.ReplacePage(uPageID: UINT;
  lpfnReplaceWith: TFNAddPropSheetPage; lParam: LPARAM): HResult;
begin
  Result := E_NOTIMPL;
end;
{ TPropertySheetFactory }
procedure TPropertySheetFactory.UpdateRegistry(Register: Boolean);
begin
```

```
    inherited UpdateRegistry(Register);
    if Register then begin
      CreateRegKey('.car', '', 'CarDemo');
      CreateRegKey('CarDemo\shellex\PropertySheetHandlers\' +
        ClassName, '', GUIDToString(ClassID));
    end else begin
      DeleteRegKey('CarDemo\shellex\PropertySheetHandlers\' +
        ClassName);
    end;
  end;
  initialization
  {$IFDEF VER100}
    TPropertySheetFactory.Create(ComServer, TPropertySheet,
  Class_PropertySheet,
      'PropertySheet', '', ciMultiInstance);
  {$ELSE}
    TPropertySheetFactory.Create(ComServer, TPropertySheet,
  Class_PropertySheet,
      'PropertySheet', '', ciMultiInstance, tmApartment);
  {$ENDIF}
  end.
```

Implementando el procedimiento de diálogo

No estamos trabajando con un formulario estándar de Delphi - en lugar de ello, estamos trabajando con ventanas de diálogo de Windows, así que tenemos que escribir algún código de bajo nivel para interactuar con la página de propiedades. Una vez que la página de propiedades es creada, Windows le envía un mensaje *WM_INITDIALOG*. En el ejemplo, éste es el único mensaje al que necesitamos responder. Sin embargo, hemos incluido código para mostrar dónde se puede responder a mensajes *WM_COMMAND* (que se envían, por ejemplo, cuando el usuario hace clic sobre un botón de la página de propiedades), y al mensaje *WM_NOTIFY* (que se envía cuando el usuario hace clic sobre los botones *Aplicar* u *OK* de la página de propiedades).

Como ya he mencionado, el único mensaje al que debemos responder en este ejemplo es *WM_INITDIALOG*. El parámetro *lParam* de *PropertySheetDlgProc* contendrá un puntero al registro *TPropSheetPage* que he inicializado en *TPropertySheet.AddPages*. Notará que recupero el campo *lParam* de la estructura para obtener una referencia al objeto *TPropertySheet*. Después de esto, ya puedo recuperar el nombre del fichero sobre el que el usuario ha pulsado, para abrirlo y obtener el nombre y modelo del vehículo.

El código para leer la información relacionada con el vehículo del fichero estructurado es el mismo que se utiliza en la aplicación *CarDemo*. Una vez obtenida la información, utilizo la función del API *SetDlgItemText* para mostrarla en los controles de edición de la página de propiedades.

SetDlgItemText realiza la acción equivalente a lo que ocurre entre bastidores cuando se escribe el siguiente código en Delphi:

```
Edit1.Text := 'Hello'
```

Implementando la función de respuesta

El otro punto de interés en este ejemplo es la función de respuesta *PropertySheetCallback*. Esta es llamada cuando la página de propiedades es destruida. El parámetro *psp* contiene el registro *TPropSheetPage* que es inicializado en *TPropertySheet.AddPages*, así que nuevamente convierto el campo *lParam* a un *TPropertySheet*. Después llamo a *_Release* para que Windows sepa que he terminado de utilizar la interfaz, y que puede destruirla cuando estime pertinente.

Nota

Cuando implemente procedimientos de diálogo y funciones de respuesta, asegúrese siempre de declararlas como stdcall. Si no lo hace, su página de propiedades no funcionará adecuadamente, y podrá observar comportamientos anómalos.

Después de compilar y registrar el servidor COM del ejemplo, podrá hacer clic con el botón derecho en el fichero CARS.CAR desde el Explorador de Windows. Al seleccionar 'Propiedades' en el menú contextual, verá una página adicional llamada '*Car Information*' en el diálogo de propiedades.

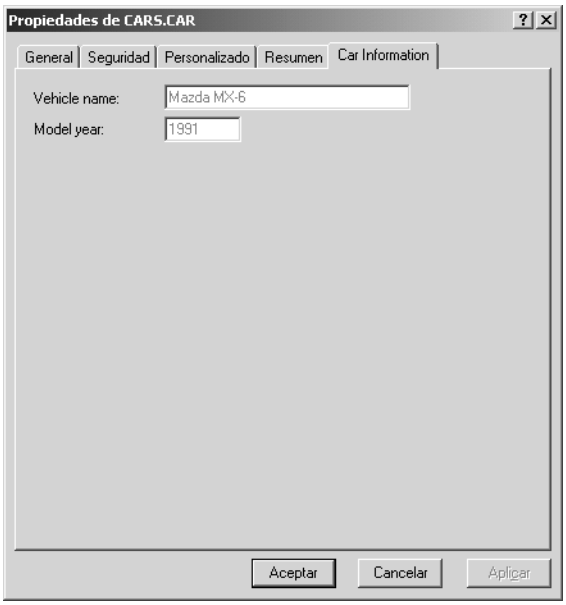


Figura 9.13: Una página de propiedades personalizada.

Dónde ir desde aquí

En este capítulo hemos examinado cinco de las interfaces COM más comunes relacionadas con el *shell* de Windows: los manejadores de menús contextuales, los manejadores de "ganchos" de copia, los enlaces del *shell*, los iconos de bandeja y los manejadores de páginas de propiedades.

A lo largo del libro, le he brindado la información necesaria para rápida y fácilmente crear clientes y servidores COM en Delphi. En algunas ocasiones he profundizado en los detalles de funcionamiento interno de Delphi y COM, pero en la mayor parte de los casos he intentado ofrecerle la información necesaria para desarrollar aplicaciones COM sin atormentarle con los detalles internos.

Para reafirmar los puntos tratados en el libro, hemos desarrollado ejemplos útiles en cada capítulo, mostrando que se puede aprender COM en pequeños incrementos, poniendo en práctica los nuevos conocimientos adquiridos casi inmediatamente.

Existen numerosas referencias adicionales que podrá utilizar para obtener mayor información sobre la programación en Delphi en general, y la programación COM en particular. Los libros específicos de COM no están dirigidos directamente a Delphi, pero pueden darnos una comprensión más profunda de las interioridades de COM, en caso de que esté interesado. El Apéndice A, "Lecturas y Recursos

Recomendados", ofrece un listado de referencias que personalmente he encontrado útiles en el pasado.

Recomendados", ofrece un listado de referencias que personalmente he encontrado útiles en el pasado.

Apéndice A

Lecturas y recursos recomendados

Si después de leer este libro desea profundizar en los conocimientos de Delphi o de la programación COM, este Apéndice le ofrece sugerencias de a dónde dirigirse.

Programación en Delphi

Los libros que se listan en esta sección tienen que ver con la programación en Delphi en general. La mayoría discuten elementos avanzados de programación (incluyendo la programación COM).

1. **Delphi 5 Developer's Guide**, de Xavier Pacheco y Steve Teixeira. Esta es una excelente referencia avanzada, escrita por dos ex-empleados de Borland. El libro está repleto de trucos y técnicas de programación en Delphi, e incluye un CD con numerosos capítulos adicionales que, por limitaciones de espacio, no están incluidos en el libro en sí. (Versión en castellano en *www.danyshop.com*).
2. **La Biblia de Delphi 5**, de Marco Cantú. Este libro es una referencia indispensable para cualquier programador de Delphi. Ofrece explicaciones detalladas de un amplio rango de posibilidades de Delphi, incluyendo la utilización de tecnologías como COM, ActiveX y CORBA. (Versión en castellano en *www.danyshop.com*).
3. **Delphi Component Design**, de Danny Thorpe. Este libro fue escrito para versiones anteriores de Delphi y está agotado, pero se lo recomiendo encarecidamente si puede conseguir una copia. Thorpe (uno de los diseñadores originales de Delphi) va más allá del resto de los autores y cuenta la historia interna de la VCL.
4. **Los Tomos de Delphi: El núcleo del API Win32**, y **Los Tomos de Delphi: El API Gráfico**, de varios autores, publicados en castellano por **Danysoft Internacional**. Referencia indispensable sobre cómo acceder al API de Windows desde aplicaciones Delphi. (+información en *www.danypress.com*).

Programación COM

Esta sección lista dos recursos adicionales que podrá consultar en relación con la programación COM. Están orientados a programadores C/C++, pero si desea un conocimiento más profundo de COM, encontrará útiles estas referencias.

1. **Essential COM**, de Don Box. El código de este libro está escrito en C++, pero es un libro valioso también para los programadores Delphi que deseen comprender completamente las interioridades del modelo COM.
2. **Microsoft Developer's Network** (MSDN). Disponible en CD en régimen de suscripción, o contactando con Danysoft *info@danysoft.com*. Esta es la referencia definitiva para las nuevas tecnologías de Microsoft, incluyendo la programación COM y las tecnologías basadas en COM, como ADO, SAPI (*Speech API*), etc.

Indice

Para facilitar la búqueda, hemos unificado todos los métodos, parámetros, listados, funciones, interfaces, etc. Bajo dichos epígrafes.

!

#5UserDefinedProperties (conjunto de propiedades): 355

A

ActiveX.pas: 64

ActiveX: 211

- controles: 211
  - Agente Microsoft, Ver Agente Microsoft
  - añadir página de propiedades: 247
  - control de licencia en tiempo de diseño: 230
  - convertir VCL a: 229
  - crear: 228
  - distribución: 248
  - información sobre la versión: 230
  - probar: 247
  - registrar: 247
  - Ventana de diálogo (About Box): 230
- formularios activos (ActiveForms): 211, 249
  - crear: 249
  - probando: 264
- Ver también: ADO (ActiveX Data Objects).

ADO (ActiveX Data Objects): 198
Arquitectura: 199
comandos: 204
conexiones: 201
errores: 205
instalar: 200
objeto Field: 205
obtener: 200
Recordset: 202
ADOTest - aplicación de ejemplo: 206
Agent, Microsoft: 211
animación: 221
Announce: 221
capacidades de voz: 219
applet del panel de control: 215
marcas: 219
comandos: 218
Ejemplo de aplicación: 222
eventos: 222
instalar: 212
obtener: 211
peticiones: 216
programación: 213
AgentDemo, Ejemplo de aplicación: 222
agregación COM: 69
Algoritmo
de burbuja: 33

NextFit: 72

Almacenamiento estructurado

Abriendo un fichero: 312

Aplicación CarDemo: 320

Comprimiendo un fichero: 318

Conjuntos de Propiedades (OLE), ver OLE - Conjuntos de Propiedades.318

Contenedores OLE, ver OLE - Contenedores OLE.

Creando almacenes adicionales: 316

Creando un fichero: 309

Documentos compuestos: 369

Enumerando: 316

Flujo (stream): 308

Conjuntos de propiedades OLE: 343

Trabajando con flujos: 313

SSView, ejemplo: 337

Uso de ficheros, ejemplo: 319

Añadir usuarios y grupos, cuadro de diálogo: 274

Aplicaciones

Automatización

Aplicación existente: 156

Aplicación MemoDemo: 156

Ejemplo de aplicación Microsoft Agent: 222

Arquitectura ADO: 199

Arrays

AreaFactor: 145

Arrays variantes: 99

Acceso: 103
Automatización: 140
Conjunto de datos remotos con DCOM: 281
Funciones, ver Funciones.
Mejorando el rendimiento: 104
VarArrayCreate: 102
VarDemo, ejemplo: 105
VarType: 100
Asignación directa de un puntero a una interfaz: 23
Asistente
de ActiveForms: 250
de Objetos COM: 76, 112
de Objetos de Automatización: 145
para Controles ActiveX, convertir VCLs: 229
Automatización: 139
Aplicaciones existentes: 156
Bibliotecas de tipos: 112
Conteo automático: 26
Disp-interfaces: 142
Cliente IntfCli: 196
Componente TEventSink: 173
Conexiones de Cliente: 187
Creando el Cliente: 194
Creando el Servidor: 185
Ejemplo de aplicación EventCli: 177
Ejemplo de Servidor de automatización EventSrv: 170
Instanciando el servidor: 187

Enlace tardío: 140
Enlace temprano: 140
Eventos: 168
Implementación de la interfaz IUnknown: 19
Interfaces: 140
Servidores de automatización externos al proceso: 154
Aplicaciones existentes: 155
Convenio de llamada safecall: 154
HResult, valores: 155
MemoCli, Aplicación: 162
MemoDemo, aplicación: 156
MemoSrv, aplicación: 158
Propiedades Color, Font y Text: 159
Servidores de automatización internos al proceso: 144
CreateOleObject, función: 148
Ejemplo de aplicación UnitCli: 149
Ejemplo de Servidor de conversión de unidades de medida: 144
Variantes: 140
Ver También: ADO (ActiveX Data Objects).
B
Bases de Datos - ADO, ver ADO (ActiveX Data Objects).42
BestFit.pas: 91
Bibliotecas de tipos: 111 - 112
Creando un Visor: 127, 129, 131, 133, 135
Co-clases: 137
Disp-interfaces: 137
Enumeraciones: 136

Extendiendo TIViewer: 138
Interfaces: 137
Extensiones: 377
IDL (Interface Definition Languaje): 115
Parámetros, ver Parámetros: in - out - var.
Propiedades de sólo lectura, sólo escritura y de lectura y escritura: 124
UnitCli, ejemplo de automatización: 151
Ver también: Editor de Bibliotecas de Tipos.
Ver también: Servidores COM externos al proceso.
BinBase.pas: 86
BinCli.dpr: 93
BinIntf.pas: 86
C
Cajas de diálogo, ver Ventanas de diálogo.422
Clases
clases abstractas: 9
derivar clases: 10
diferencias entre una interfaz y una clase: 11
instancia: 10
Coclass (Editor de Bibliotecas de Tipos): 119
interfaces
diferencias entre una interfaz y una clase: 11
múltiples interfaces: 43
Objeto TPropertySheet: 426
TBestFit: 85
TComObject: 61
TConn: 187

TConns: 187
TEmployee: 36 - 37
TEmployeeOrder: 37
TEventSink: 172
TFirstFit: 84
TFormattedHexInteger: 29, 33
TFormattedInteger: 29, 33
TFormattedNumber: 10, 17
TGUID: 15
TInterfacedObject, interfaz IUnknown: 19
TInventoryItem: 36, 39, 42
TIViewer: 127, 133
TListBox: 41
TListView: 43
TNextFit: 75, 80
TObject: 12, 23
TPageControl: 41
TPropertySet: 357
TTypedComObject: 112
TUnitAuto: 118
TWriter: 320
Class_DColorPropPage, controles ActiveX: 248
Class_DFontPropPage, controles ActiveX: 248
Class_DPicturePropPage, controles ActiveX: 248
Class_DStringPropPage, controles ActiveX: 248
Class_NextFit GUID: 75
Cláusulas

Cláusula uses: 111
resolución de métodos: 44, 62
clave ProgID: 60
Clientes
Automatización
Ver: Automatización.
cláusula uses: 111
DCOM
Ver: DCOM.
Distribución unidimensional
Ver: Distribución unidimensional.
llamada safecall: 155
CLSIDs, identificadores de clase: 59
Controles ActiveX: 248
Servidores COM internos al proceso: 68
Código fuente
Editor de Bibliotecas de Tipos: 125
TInterfacedObject: 19
VCL IUnknown: 17
COM (Component Object Model)
agregación: 69
Arrays variantes, ver Arrays - Arrays Variantes.
COM Distribuido, ver DCOM.
GUIDs y COM: 59, 61, 63
Objetos: 60
Asistente de objetos COM: 76, 112
Compartimentado & libre: 66

Compartimentado: 66

con otros lenguajes: 109

de un solo hilo: 66

DLLs, ver Servidores COM internos al proceso.

Ejecutables, ver Servidores COM externos al proceso.

Fábricas de Clases: 60

Multihilado libre: 66

Recursos: 436

Servidores COM externos al proceso: 97

IMarshal para nuestros datos: 99

Instanciación: 97

Servidores COM internos al proceso: 60, 377

Accesos directos del shell: 399

ActiveX Library: 73

Constructores: 68

Distribución unidimensional optima, ejemplo: 71

Instancias: 68

Manejadores de menús contextuales, ver Manejadores de menús contextuales.

Manejadores de páginas de propiedades, ver Manejadores de páginas de propiedades.

Soporte de múltiples hilos: 66

Comandos

adCmdFile, función Recordset: 204

adCmdStoredProc, función Recordset: 203

adCmdTable, función Recordset: 203

adCmdTableDirect, función Recordset: 204

adCmdText, función Recordset: 203
adCmdUnknown, función Recordset: 203
adCmdUnspecified, función Recordset: 203
adExecuteNoRecords, función Recordset: 204
ADO (ActiveX Data Objects): 205
Menú File
New: 72
Save All: 125
Save: 73
Menú Run
Register ActiveX Server: 67
Unregister ActiveX Server: 67
Menú View | Type Library: 118
Microsoft Agent: 218
Play: 220
Speak: 219
Recordset, función: 203
ComObj.pas (función CreateComObject): 68
Compartimentado & libre
Objetos COM: 66
Compatibilidad (modelos interfaz/objeto): 25
Componentes
EventSink: 7
PropertySet: 7
TDCPTrayIcon: 414
TPropertySet: 357
TrayIcon: 7

Comprimiendo un fichero de almacenamiento estructurado: 318

ComServ.pas: 67

conexión, funciones: 201

Conjunto de datos remotos, DCOM (COM distribuido):

Clientes administrativos: 293

Clientes punto de venta: 300

Servidores: 279

Constantes (Editor de Bibliotecas de Tipos): 121

Contactar con Danypress: 3

Contador interno de bloqueos: 64

Contenedores OLE, ver OLE - Contenedores OLE.369

Convertir un componente VCL a un control ActiveX: 229

Copiar objetos, (contenedores OLE): 372

CopyHookDemo: 396

CORBA, (código fuente de la biblioteca de tipos): 125

Cuadro de diálogo, ver Ventanas de diálogo.422

Cursores (función RecordSet): 203

D

DCOM (COM Distribuido): 267

Conjunto de datos remotos, ver Conjunto de datos remotos, DCOM (COM Distribuido).

Creando clientes DCOM: 276

Servidores

Administrador de Usuarios: 272

Conjuntos de datos remotos: 279

Creación: 268

Instalación: 269, 272, 275

Web: 267
DCOMClient: 276
Declaraciones
Clase TComObject: 61
Defragmentar (Ficheros de almacenamiento estructurado): 318
Delegación de interfaces: 43
Derivar clases: 10
Desactivando servidores OLE: 371
dirección MAC: 14
Disp-interfaces
Bibliotecas de tipos: 137
Editor de Bibliotecas de tipos: 120
Eventos de automatización: 142, 168
IUnitAutoDisp: 143
Ver también: Automatización - Disp-interfaces.
Distribución unidimensional óptima: 71
Bin1Srv.dpr: 76
BinIntf.pas: 77
Creación: 73
NextFit.pas: 75
DLLs (dynamic link libraries)
Accesos directos al Shell, ver Shell de Windows - Accesos directos.377
ActiveX Library: 73
Automatización: 144
CreateOleObject, función: 148
Ejemplo de aplicación UnitCli: 149
Ejemplo de Servidor de conversión de unidades de medida: 144

Sencillo de crear: 147

Constructores: 63

Ejemplo: Distribución unidimensional óptima: 71

Fábricas de Clase: 60

Iconos de la bandeja, ver Iconos de la bandeja.

Instancias: 63

Manejador de gancho de copia, ver Manejador de gancho de copia.

Manejadores de menús contextuales, ver Manejadores de menús contextuales.

Manejadores de páginas de propiedades, ver Manejadores de páginas de propiedades.63

Multihilos: 66

Registrando: 67

DocFiles (almacenamiento estructurado): 307

Ver también: Almacenamiento estructurado.

Documentos compuestos (almacenamiento estructurado): 369

E

Editor de Bibliotecas de Tipos: 117

Alias: 121

Ayuda F1: 117

Barra de Herramientas: 119

CoClass: 120

Código fuente: 125

Constantes: 120

Dispinterfaces: 120

Enumeraciones: 120

Interfaces: 120, 137

IUnitAuto: 119
Lista de objetos: 118
Métodos: 121, 137
Módulos: 121
Nodos: 119
Propiedades: 124
Registros: 121
Servidores COM externos al proceso: 126
Uniones: 121
Ejecutables
Biblioteca de tipos: 127
Fábricas de Clases: 60
Instancias: 97
Marshaling: 98
Ver: Automatización - Servidores de automatización externos al proceso.
Enlace (automatización)
Tardío: 140
Temprano: 140
Enumeradores
Bibliotecas de tipos: 136
EventCli, ejemplo de aplicación: 177
EventMultSrv, ejemplo de aplicación: 183
Eventos
Automatización, ver Automatización:
Microsoft Agent: 216
OnRequestComplete: 217
EventSrv, servidor de automatización: 169

F

Fábricas de Clases: 60, 63

Ficheros

PwcsName: 309

TLB: 111

FindProp, ejemplo: 365

FRefCount, variable: 19

Fuentes

Marshaling: 154

Servidores COM externos al proceso: 158

Funciones

_AddRef: 22

clase TComObject: 62

operador as: 25

redefinir: 27

_Release: 22

clase TComObject:62

redefinir: 27

AddRef, ver _AddRef.

CoCreateInstance: 148

Command: 204

Connection: 202

Convert: 145

CreateComObject: 68

CreateInstance: 64

CreateOleObject: 148

CreateRemote (DCOM): 277

DllCanUnloadNow: 65
DllGetClassObject: 65
DllRegisterServer: 65
DllUnregisterServer: 65
GetActiveOleObject: 148
GetFuncDesc: 134
GetInterface: 23
GetTypeInfoCount: 135
GrfMode: 309
IShellExtInit:
    Parámetros: 379
Llamando: 22
LoadTypeLibrary: 134
    Co-Clases: 137
    Enumeraciones: 136 - 137
LockServer: 64
QueryInterface: 17
Recordset: 202
    cursores: 203
    Tipos de bloqueo: 203
Release, ver _Release.
Shell_NotifyIcon: 412
SHGetSpecialFolderLocation: 399
StgCreateDocFile: 309
StgOpen: 309
TComObject: 61
VarArrayCreate: 102

VarArrayOf: 103
VarType: 100
G
grfMode, valores válidos: 310
GrphDemo, demostración gráfica: 50
GUIDs (Globally Unique Identifiers): 13
Class_NextFit: 75
CreateOleObject: 141
GUIDDemo: 14
Interfaces: 14
Núcleo de interfaces COM: 64
Programación COM: 59
Tarjetas de red: 14
H
Herencia (múltiples interfaces): 44
Heurística: 71
Hilado
En servidores COM internos al proceso: 66
Objetos COM, ver COM - Objetos.
HKEY_CLASSES_ROOT: 59
HResult, ver valores - HResult: 383
I
Iconos de la bandeja: 412
Identificadores
Función Shell_NotifyIcon: 413
Funciones de conexión: 202
Método CopyCallback: 393

Instalar
ADO, ver ADO (ActiveX Data Objects).
Agent Microsoft, ver Agent Microsoft - Instalar.
DCOM, ver DCOM (distributed COM).
Servidores COM internos al proceso: 67
Instanciación múltiple (servidor COM externo al proceso): 98
instancias
clases abstractas: 9
Interfaces: 12
Objetos COM: 63
Servidores COM externos al proceso: 98
Servidores COM internos al proceso: 68
Interfaces
Automatización: 140
Cambiar: 12
COM (Component Object Model): 59
comparación con las clases abstractas: 11
Compatibilidad con el modelo de objetos: 25
declarar: 13
Delegación: 45
directivas de acceso: 12
Duales: 144
Editor de Bibliotecas de Tipos, ver Editor de Bibliotecas de Tipos.
Funciones: 12
GrphDemo: 50
GUIDs (Globally Unique Identifiers): 13
IAreaUnitConverter: 147

IClassFactory: 63
ICompare: 34
IContextMenu: 380
ICopyHook: 392
IDispatch: 141
IFormattedNumber: 11
IFormattedNumber2: 13
IIntfCallbackEvents: 186
Implementando: 15
Instancias: 12
IntfDemo, ejemplo: 28
IPersistFile: 403
IPropertySetStorage: 344
IPropertyStorage: 344
    Métodos: 354
IProvideClassInfo: 114
IShellExtInit: 379
IShellLink: 402
IShellPropSheetExt: 424
IStorage: 309
    Métodos: 312 - 317
ITest: 115
ITestEvents: 195
ITypeInfo: 114, 133
IUnitAuto: 143
IUnknown: 12, 16
    _AddRef: 17

_Release: 17
Implementación automática: 19
Implementación manual: 17
QueryInterface: 17
TComObject: 62
TFormattedNumber: 17
Las referencias se cuentan: 17
manejadores de páginas de propiedades: 424
Múltiples interfaces: 43
obtener:
Asignación directa: 23
GetInterface: 23
operador as: 24
Propiedades: 50
Punteros: 23
ventajas sobre las clases: 20
Interfaces de respuesta: 185
Creando el cliente: 194
Creando el servidor: 185
Instanciando el servidor: 187
IntfSrv, servidor de automatización: 188
Manejando las conexiones de cliente: 187
IntfSrv, servidor de automatización: 188
Iterando, (ficheros de almacenamiento estructurado): 316
L
Lernout & Hauspie TruVoice Text-To-Speech Engine: 212
LinkDemo: 404

Listados

About1.pas: 246
ADO Text - MainForm.pas: 206
AgentDemo - AgentDemo.dpr: 223
AgentDemo - MainForm.pas: 224
AgentTest - AgentTest.dpr: 213
AgentTest - MainForm.pas: 214
Bin1Cli - Bin1Cli.dpr: 82
Bin1Cli - MainForm.pas: 83
Bin1Srv - Bin1Srv.dpr: 76
Bin1Srv - Bin1Srv_TLB.pas: 81
Bin1Srv - BinIntf.pas: 77
Bin1Srv - NextFit.pas: 77
Bin1Srv2 - BestFit.pas: 91
Bin1Srv2 - Bin1Srv.dpr: 86
Bin1Srv2 - BinBase.pas: 86
Bin1Srv2 - BinIntf.pas: 86
Bin1Srv2 - FirstFit.pas: 90
Bin1Srv2 - NextFit.pas: 89
BinCli2 - BlinCli.dpr: 93
BinCli2 - MainForm.pas: 93
CarDemo - CarDemo.dpr: 321
CarDemo - CarInfoForm.pas: 324
CarDemo - GasForm.pas: 326
CarDemo - GasItemForm.pas: 330
CarDemo - MainForm.pas: 321
CarDemo - OilForm.pas: 331

CarDemo - OilItemForm.pas: 336

CheckListBoxImpl1.pas: 231

CLBX_TLB.pas: 238

Código generado por el Asistente de Objetos COM: 113

Código IDL para la interfaz ITest: 116

Componente EventSink: 173

CopyHookDemo - CopyHookDemo.dpr: 396

CopyHookDemo - CopyHookDemo_TLB.pas: 396

CopyHookDemo - CopyHookImpl.pas: 397

CtxDemo - CtxDemo.dpr: 386

CtxDemo - CtxDemo_TLB.pas: 386

CtxDemo - TIViewImpl.pas: 387

DCOMClient - DCOMClient.dpr: 276

DCOMClient - MainForm.pas: 276

DCOMServer - DCOMServer.dpr: 269

DCOMServer - DCOMServer_TLB.pas: 270

DCOMServer - MainForm.pas: 271

DCOMServer - SimpleServer.pas: 269

Esqueleto generado por el Asistente de Objetos COM: 424

EventCli - MainForm.pas: 177

EventCli5 - MainForm.pas: 180

EventMulSrv - EventIntf.pas: 183

EventSrv - EventIntf.pas: 169

EventSrv - EventSrv_TLB.pas: 170

Fichero de proyecto del control ActiveX: 231

FindProp - FindProp.dpr: 366

FindProp - MainForm.pas: 366

Formulario principal de la aplicación GUIDDemo: 14
FriendX - AboutForm.pas: 251
FriendX - FriendFormImpl.pas: 258
FriendX - FriendX.dpr: 250
FriendX - FriendX_TLB.pas: 251
GrphDemo - código fuente de IntfForm.pas: 51
GrphDemo - grphdemo.dpr: 57
GrphDemo - MainForm.pas: 54
GrphDemo - Shapes.pas: 52
Implementación de IUnknown: 17
IntfCli - MainForm.pas: 196
IntfDemo - IntfDemo.dpr: 32
IntfDemo - IntfUnit.pas: 29
IntfDemo - MainForm.pas: 31
IntfSrv - IntfSrv_TLB.pas: 192
IntfSrv - IntfUnit.pas: 188
IntfSrv - Mainform.pas: 192
LinkDemo - LinkDemo.dpr: 404
LinkDemo - MainForm.pas: 404
LinkDemo - NewLinkForm.pas: 410
Manejador de gancho de copia: 394
MemoCli - mainForm.pas: 165
MemoCli - MemoSrv_TLB.pas: 162
MemoDemo - MainForm.pas: 156
MemoSrv - MainForm.pas: 158
MemoSrv - MemoIntf.pas: 160
NextFit.pas: 75

OLECont - MainForm.pas: 373

PartAdmin - MainForm.pas: 294

PartAdmin - PartAdmin.dpr: 293

PartAdmin - PartForm.pas: 298

PartSrv - DataModule.pas: 286

PartSrv - MainForm.pas: 292

PartSrv - Parts.pas: 285

PartSrv - PartServerImpl: 289

PartSrv - PartSrv.dpr: 282

PartSrv - PartSrv_TLB.pas: 282

PartStock - MainForm.pas: 300

PartStock - PartStock.dpr: 300

Plantilla de código para el manejador de gancho de copia: 392

Plantilla de código para un manejador de menú contextual: 378

PPage.RC - Plantilla de diálogo para PropDemo.DLL: 423

PropDemo - MainForm.pas: 362

PropDemo - PropDemo_TLB.pas: 427

PropSets.pas - Definiciones de interfaces para Delphi 3: 345

SortDemo - EmpUnit.pas: 36

SortDemo - IntfUnit.pas: 35

SortDemo - InvUnit.pas: 37

SortDemo - MainForm.pas: 39

SSView - MainForm.pas: 338

SSView - SSView.dpr: 337

TInterfacedObject: 19

TIViewer - MainForm.pas: 128

TIViewer - TIViewer.dpr: 133

TrayDemo - MainForm.pas: 420
TrayDemo - TrayDemo.dpr: 419
TrayIcon.pas: 415
VarDemo - MainForm.pas: 105
VarDemo - VarDemo.dpr: 105
lpVerb: 384
M
MainForm.pas, ver listados.31
Manejadores
de gancho de copia: 394
CopyHookDemo: 396
ICopyHook, interface: 392
Registrando: 396
de menús contextuales: 377
Ejemplo CtxDemo: 386
IContexMenu, interface: 379
IShellExtInit, interface: 379
Registrando: 385
de páginas de propiedades: 421, 423, 425, 427, 429, 431
esqueleto del proyecto: 424
Interfaces: 424
Plantillas de diálogo: 423
PropDemo, ver PropDemo.
MemoCli: 161
MemoDemo: 157
MemoSrv: 161
Métodos

AddPage: 425

AddPages: 426

AppShow: 141

con el mismo nombre: 44

Convert: 121

CopyCallback: 393

CreateParameter: 204

CreateStorage: 316

CreateStream: 312

CustomSort: 42

DeleteMultiple de IPropertyStorage: 354

EnumElements: 316

Find (Aplicación ADOTest): 210

FormattedString: 9

FormCreate: 41

GetCommandString: 383

GetData (parámetro lpdobj): 380

GetIDsOfNames, interfaz IDispatch: 142

GetTypeInfo, interfaz IDispatch: 142

GetTypeInfoCount, interfaz IDispatch: 142

Initialize (interfaz IShellExtInit): 379

InvokeCommand, interfaz IContextMenu: 384

New: 146

Open (función Connection): 202

OpenStream: 314

QueryContextMenu: 381

Read: 315

ReadMultiple del interfaz IPropertyStorage: 352
ReplacePage: 425
SendText (servidor de automatización EventSrv): 170
Ver también: Editor de Bibliotecas de Tipos.
Write: 313
WriteMultiple del interfaz IPropertyStorage: 353
Microsoft
Ver: ADO (ActiveX Data Objects).
Ver: Agent Microsoft.
MIDL, código fuente:125
Modelo
de Interfaz: 25
de Objeto: 25
N
Nodos, ver Editor de Bibliotecas de Tipos: 119
O
Objetos
COM, ver COM - Objetos.
Contenedores OLE, ver OLE - Contenedores OLE.25
de respuesta: 195
Field: 205
Propiedades comunes: 205
Ver también: ADO - objeto Field.
TBinItem: 80
TFirstFit: 84
OLE (Object linking and embedding)

Conjuntos de Propiedades: 343, 345, 347, 349, 351, 353, 355, 357, 359, 361, 363, 365, 367

Eliminando propiedades: 354

Enumerando propiedades: 354

Escribiendo propiedades: 353

FindProp, ejemplo: 365

Flujos: 344

IPropertySetStorage & IPropertyStorage, ver interfaces.

Leyendo propiedades: 352

PropDemo, ejemplo: 361

TPropertySet, ver Componentes - TPropertySet.

Contenedores OLE: 369

Cargando objetos: 371

Cuadro de diálogo Insert Object: 370

Ejemplo OLECont: 373

Guardando objetos: 371

Mezcla de menús: 371

Portapapeles: 372

Recursos: 435

OleAutomation: 126

OleCheck: 62

Operador as: 25, 352

_AddRef, función: 24

en Servidores COM internos al proceso: 70

obtener interfaces: 24

Ordenando

ADOTest, aplicación de ejemplo: 210

Algoritmos: 33

Interfaces: 34

Override, directiva: 27

P

Parámetros

adParamInput, función Command: 204

adParamInputOutput, función Command: 204

adParamOutput, función Command: 204

adParamReturnValue, función Command: 204

adParamUnknown, función Command: 204

AOrderBy: 35

cbSize, función Shell_NotifyIcon: 414

cchMax, función GetCommandString: 383

clsid: 69

ComClass: 76

ComServer: 76

Contraseña de acceso: 201

DSN: 201

dwClsContext (servidores COM internos al proceso): 69

dwDestAttribs, método CopyCallback: 394

dwMessage, función Shell_NotifyIcon: 413

dwSrcAttribs, método CopyCallback: 394

hIcon, función Shell_NotifyIcon: 414

hKeyProgID, interfaz IShellExtInit: 379

hMenu de QueryContextMenu: 383

hwndOwner, función SHGetSpecialFolderLocation: 400

ID de usuario: 201

idCmd, método GetCommandString: 383
iid, servidores COM internos al proceso: 68
in: 123
indexMenu de QueryContextMenu: 383
lpData: 413
Lpdobj: 379
Medium: 380
nFolder de SHGetSpecialFolderLocation:
    Valores: 400 - 402
out (de salida): 122
pidlFolder, interfaz IShellExtInit: 379
PszDestFile, (método CopyCallback): 394
PszName, (método GetCommandString): 383
PszSrcFile, (método CopyCallback): 394
pv (servidores COM internos al proceso): 69
szTip, función ShellNotifyIcon: 414
uFlags de QueryContextMenu:
    Valores: 381
uID, función Shell_NotifyIcon: 414
uIDNewItem: 383
uPosition, (QueryContextMenu): 383
uType: 384
    Valores: 384
var (de entrada y salida): 122
wFlags de CopyCallback: 393
    Valores: 393
wFunc de CopyCallback: 393

Valores: 393
Wnd
CopyCallback, método: 393
Shell_NotifyIcon, función: 414
Portapapeles, (contenedor OLE): 372
Problemas NP-completos: 71
Procedimientos
Interfaces: 12
LoadCoClass: 137
LoadEnum: 136
LoadInterface: 137
OleCheck: 63
PropDemo
Función de respuesta: 432
PropDemo.DLL: 423
PropDemo.dpr: 361, 426
PropDemo_TLB.pas: 427
PropSheet.pas: 424
Ventanas de diálogo: 431
PropDemo, ejemplo: 426
Propiedades
ConnectKind: 180
Valores: 180
en las interfaces: 50
GroupIndex: 372
MemoSrv, aplicación: 159
objeto Field: 205

Punteros
a una interfaz: 23
Asignación directa: 23
GetInterface, función: 23
operador as: 24
FController (clase TComObject): 62
R
Recursos para el programador Delphi: 435
Registrando
Biblioteca de tipos: 125
el manejador de menú contextual: 385
Servidores COM internos al proceso: 67
un manejador de gancho de copia: 396
un control ActiveX: 247
Registro
DllRegisterServer: 65
DllUnregisterServer: 65
HKEY_CLASSES_ROOT: 59
S
Safecall, directiva: 154
Serializando (marshaling): 155
Bibliotecas de tipos: 122
Cadenas, fuentes e imágenes: 155
Conjuntos de datos remotos: 280
Servidores COM externos al proceso: 98
Servidores
Automatización, ver Automatización.

COM externos al proceso: 97

Bibliotecas de tipos: 126

Instanciación: 97

Serialización de datos (marchaling): 98

Ver: Automatización - Servidores de Automatización externos al proceso.

COM internos al proceso: 65

Ver: Automatización - Servidores de Automatización internos al proceso.

Ver: COM (Component Object Model) - Servidores COM internos al proceso.

DCOM (COM Distribuido), ver DCOM - Servidores.

Desactivando Servidores OLE: 371

Disp-interfaces, ver Disp-interfaces - Eventos de automatización.

Ver: Interfaces de respuesta.

Shell de Windows: 377

Accesos directos: 400

Extensiones

Ver: Manejador de gancho de copia.

Ver: Manejadores de menús contextuales.

Ver: Manejadores de páginas de propiedades.

Iconos de la bandeja: 412

SortDemo, ejemplo: 33

SysTools: 404

T

Tabla de métodos virtuales: 69

Tipo interface: 12

U

Usuarios

Interfaces: 11

Servidores DCOM: 272

Utilidades

DCOMCNFG: 278

RegSvr32: 67

V

Valores

ConnectKind, propiedad: 180

grfMode, función StgCreateDocFile: 310

HResult: 62

uFlags de QueryContextMenu: 381

Variables

Arrays variantes, ver Arrays - Arrays variantes.

Declarar: 12

FRefCount: 19

nil: 23

Variantes: 140

Variante V: 141

Ventanas de diálogo

About Box: 230

Añadir usuarios y grupos: 274

Import Type Library: 179

Insert Object: 370

Nuevos elementos:73

Propiedades: 422

SimpleDCOMServer Object: 273

Speech Properties: 215

Web Deployment Options: 264

Versiones

Controles ActiveX: 230

Interfaces: 12

Visual Basic, probando el control ActiveX: 247

VMT: Tabla de métodos virtuales: 69

W

Webs

Danypress: 3, 435

Danyshop: 435

Danysoft: 3

DCOM: 267

Microsoft Agent: 211

MSDN: 64

Word, código de automatización: 140