Francisco Charte

Manual de Introducción a Microsoft® Visual C#® 2005 Express Edition

Microsoft®

Índice de contenidos

Introducción

La habitual expresión "los tiempos avanzan que es una barbaridad" que solemos oír con bastante frecuencia, resulta ciertamente aplicable al sector informático y, más aún si cabe, al mundo de los lenguajes y herramientas de programación. En apenas una generación humana hemos pasado de los lenguajes cercanos a la máquina a los actuales y modernos lenguajes orientados a componentes, dejando por el camino lenguajes sin estructuras propiamente dichas, como los antiguos BASIC y FORTRAN; lenguajes procedimentales, como Pascal y C, y lenguajes orientados a objetos, como C++, por mencionar los más conocidos.

En la actualidad la mayoría de las aplicaciones que se crean son desarrolladas a partir de *componentes software*, pequeñas porciones de código reutilizable que, en ocasiones, forman parte de los servicios que ofrece el lenguaje o el sistema operativo, mientras que en otras se adquieren por separado de terceras partes. En cualquier caso, los programadores de hoy necesitan lenguajes de programación que sean capaces de utilizar esos componentes como una construcción más, al tiempo que simplifican su desarrollo para uso propio o de terceros. Además, no se exige únicamente un compilador o un entorno básico de edición/compilación/depuración, sino un completo entorno que facilite el diseño de las interfaces de usuario, la gestión de los proyectos, que cuente con herramientas para el acceso a datos, etc.

Lo que Microsoft nos ofrece con *Visual C# 2005 Express Edition* es un entorno de desarrollo de última generación, con gran parte de los elementos de *Visual Studio 2005*, conjuntamente con el compilador de uno de los lenguajes de programación más avanzados que existen en la actualidad: C# 2.0. A esto se une una extensa documentación de referencia en línea, material de apoyo como este libro y una amplia comunidad de programadores a nivel mundial.

¿A quién va dirigido Visual C# 2005 Express Edition?

Nos encontramos ante un producto que se adquiere de forma gratuita y que, tras el correspondiente registro (también gratuito), puede utilizarse sin ningún tipo de limitación en el tiempo. Un producto que incorpora, como se ha dicho, el lenguaje de programación seguramente más avanzado que podamos encontrar actualmente, conjuntamente con un entorno muy funcional. Ese mismo lenguaje y mismo entorno son los que encontraremos en herramientas como Visual Studio, producto usado por miles de desarrolladores en todo tipo de empresas para crear sus aplicaciones.

Con *Visual C# 2005 Express* podemos aprender a desenvolvernos con un lenguaje y en un entorno que, posiblemente, en el futuro tengamos que utilizar cuando nos incorporemos a algún tipo de actividad laboral. Debemos pensar que Windows es el sistema operativo más difundido y para el que más aplicaciones se crean, por lo que

disponer de un cierto conocimiento sobre las herramientas necesarias para realizar ese trabajo será, sin duda, un punto a nuestro favor.

Además de como herramienta de aprendizaje, este producto puede servirnos también para desarrollar aplicaciones simples que, como aprenderemos en los capítulos de este libro, pueden tener interfaces de usuario basadas en ventanas, utilizar bases de datos y acceder a los distintos servicios de la plataforma .NET, todo ello sin ningún coste. Toda la experiencia adquirida puede asimismo servirnos con posterioridad, en caso de que tuviéramos que usar Visual Studio o cualquier otro entorno de desarrollo .NET.

Objetivos de este libro

La finalidad de este libro es guiarle mientras da sus primeros pasos en el aprendizaje de *Visual C# 2005 Express Edition*, mostrándole las estructuras del lenguaje de programación, las tareas comunes en el entorno de desarrollo y enseñándole a crear aplicaciones, con o sin interfaz gráfica de usuario, que usan algunos de los componentes y servicios de que dispone esta plataforma.

Asumimos que el lector tiene conocimientos generales sobre programación, aunque no esté familiarizado con versiones previas del lenguaje C#, Visual Studio o la plataforma .NET. El reducido espacio de este libro no nos permite, por ejemplo, adentrarnos en la teoría relativa a la programación orientada a objetos o los aspectos teóricos sobre tratamiento de bases de datos. Sí se tratarán, por el contrario, los temas relativos a orientación a objetos con C# y el acceso a bases de datos con los componentes de *Visual C# 2005 Express Edition*.

El objetivo es que cuando finalice la lectura de este libro tenga los conocimientos generales necesarios para crear aplicaciones básicas y que, al tiempo, se sienta capacitado para adentrarse en temas más avanzados.

En la redacción de este libro se ha utilizado una versión preliminar de Visual C# 2005 Express Edition *en inglés, por lo que las imágenes incluidas del entorno, ventanas y opciones aparecen en dicho idioma. El producto definitivo, sin embargo, estará traducido al castellano.*

Tipografías y recursos usados en el libro

En la redacción de este libro el autor ha recurrido a diversas tipografías y estilos a fin de resaltar en el texto los conceptos importantes y diferenciar las distintas categorías de elementos a medida que se hace referencia a ellos. Los tres estilos usados son los siguientes:

- *Letra en cursiva*: Se ha empleado para los términos que no están en castellano, los nombres de productos y también para destacar un término o concepto que aparece por primera vez.

- **Letra en negrita**: Los textos en negrita representan elementos de interfaz (opciones de menú, títulos de ventanas, botones, etc.) y teclas del ordenador (**Esc**, **Intro**, etc.)

- `Letra monoespacio`: Representa cualquier elemento de código: nombres de clases y objetos, de propiedades, eventos, valores introducidos en variables o propiedades, etc., en general cualquier texto que se asuma como código.

Además de estas tres tipografías, en el texto también se han usado ciertos recursos, como los recuadros en torno a los títulos o los fondos de color gris, para crear una clara separación virtual entre las distintas secciones del texto, a fin de facilitar su lectura, así como para destacar conceptos importantes, ideas o advertencias en forma de notas.

Sobre el autor

Francisco Charte Ojeda lleva dos décadas dedicándose a la enseñanza en el campo de la informática, tiempo durante el cual ha publicado varios centenares de artículos en las más importantes revistas del sector, desde los tiempos de *MSX Extra* y *MSX Club* hasta las actuales *PC Actual*, *Sólo Programadores* y *PC World*, pasando por *Unix Magazine*, *Revista Microsoft para Programadores*, *RPP*, *PC Magazine*, etc. También es autor de setenta libros, principalmente sobre sistemas operativos y lenguajes de programación, publicados por las editoriales Anaya Multimedia, RA-MA y McGraw-Hill. Desde 1997 mantiene su propio sitio Web, actualmente en la dirección `http://www.fcharte.com`, en el que se recoge información sobre toda su bibliografía y también ofrece públicamente artículos, noticias del sector y otros materiales de interés.

1. Primeros pasos

En este primer capítulo nos proponemos dar nuestros primeros pasos con *Visual C# 2005 Express Edition*, iniciando un nuevo proyecto, familiarizándonos con los distintos elementos que aparecen en el entorno de desarrollo, editando nuestra primera línea de código, aprendiendo a compilar, depurar y ejecutar la aplicación desde el propio entorno y a distribuir ese programa para que puedan utilizarlo los usuarios finales.

Todo este proceso nos servirá en los capítulos siguientes, y en cualquier otro proyecto que aborde con posterioridad, ya que en él conoceremos muchos conceptos, y algunos de los elementos y opciones que nos ofrece esta herramienta.

Al iniciar *Visual C# 2005 Express Edition* por primera vez nos encontramos con un entorno como el que aparece en la figura 1.1 (página siguiente), con elementos habituales como un menú de opciones, una barra de botones debajo y una serie de ventanas adosadas. La **Página de inicio** aparece en primer plano, ofreciendo una lista de los proyectos en los que hemos trabajado recientemente (vacía por ahora) y enlaces para abrir y crear otros proyectos; debajo una lista de accesos directos a temas de la ayuda y a la derecha una serie de hipervínculos a noticias relacionadas con el producto obtenidas a través de Internet. Esta página estará vacía si no disponemos de una conexión a la red.

La parte derecha del entorno está ocupada por el **Explorador de soluciones**, una ventana que nos servirá para gestionar los elementos que forman parte de cada proyecto. Observe la parte superior de ese panel: en el extremo derecho, tras el título, aparece una serie de iconos:

 Este botón da paso a un menú desplegable que nos permite establecer la configuración de la ventana, eligiendo entre acoplarla a otras (estado en el que se encuentra por defecto), dejarla flotante, ocultarla automáticamente, etc.

 Este icono aparece en las ventanas que se encuentran acopladas. Al hacer clic sobre él la ventana pasa a ocultarse automáticamente cuando no está en uso, quedando visible únicamente una pestaña. En el margen superior izquierdo de la figura 1.1 puede ver una pestaña con el título **Toolbox**, correspondiente al **Cuadro de herramientas** que se encuentra oculto. Situando el puntero del ratón sobre esa pestaña se abrirá la ventana, que volverá a ocultarse automáticamente cuando no se use a menos que hagamos clic sobre este icono.

 Como es fácil imaginar, este icono sirve para cerrar la ventana. Podemos cerrar todas aquellas ventanas que no utilicemos con frecuencia, para liberar

así el entorno y tener más espacio libre. En cualquier momento podemos recurrir a las opciones del menú **Ver** para volver a abrir las que nos interesen.

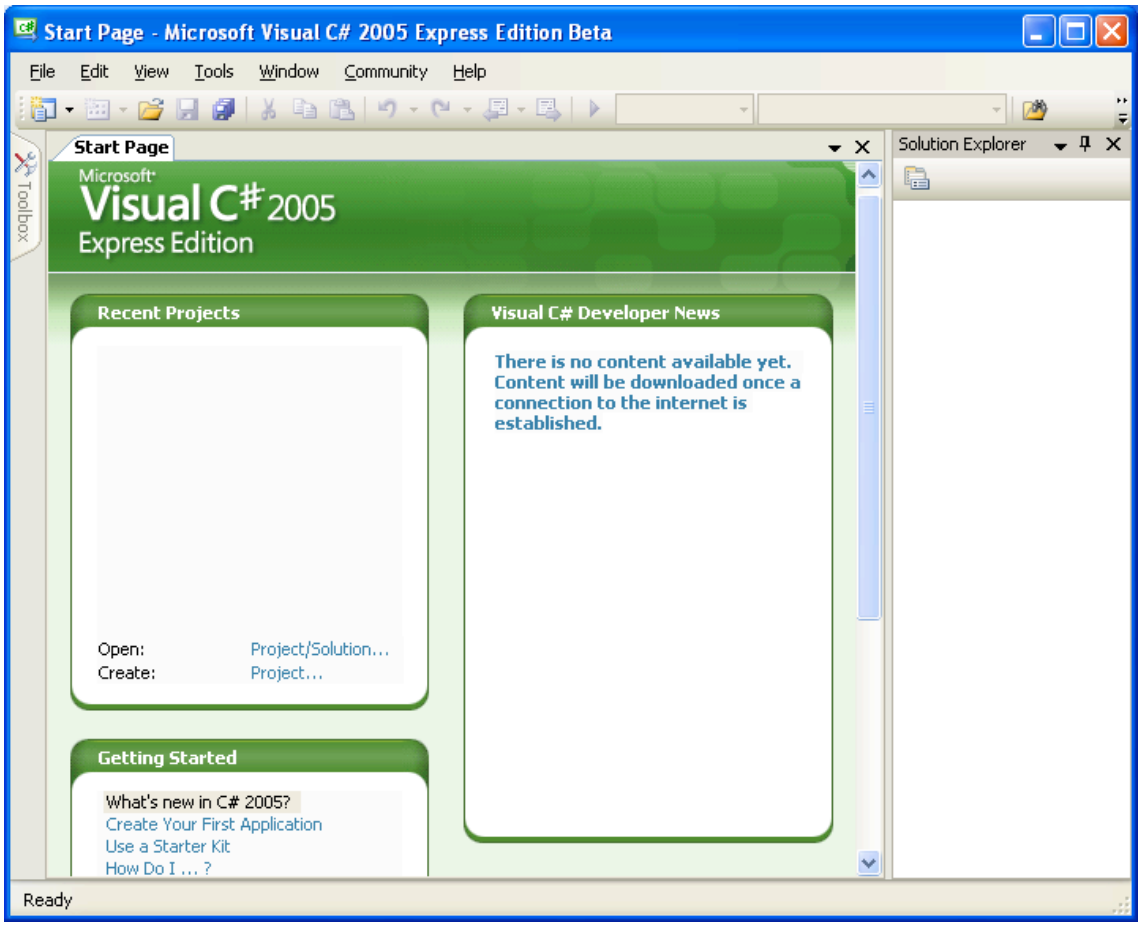

Figura 1.1. Aspecto inicial del entorno de desarrollo.

El área central del entorno, donde ahora mismo se encuentra la **Página de inicio**, sirve para ir mostrando las ventanas de los distintos editores y diseñadores con que cuenta el producto. Por cada página abierta aparecerá una pestaña en la parte superior, de tal forma que un simple clic nos llevará de una a otra.

*De manera similar a como utiliza la combinación de teclas **Alt-Tab** para cambiar entre las aplicaciones abiertas en el sistema, encontrándose en el entorno de* Visual C# 2005 Express Edition *puede recurrir al atajo de teclado **Control-Tab** para navegar por las páginas abiertas en el área central del entorno.*

Las ventanas no tienen una posición fija en el entorno, sino que podemos colocarlas donde más cómodo nos resulte. Basta con tomar la ventana por la barra de título y arrastrarla allí donde interese. Durante el proceso aparecerán unas guías, en forma de iconos semitransparentes (véase la figura 1.2), que facilitan la operación. Basta con llevar el punto del ratón a uno de los estos iconos para acoplar la ventana a uno de los márgenes o apilarla como una página más, viendo de forma interactiva dónde quedaría antes de que la soltemos.

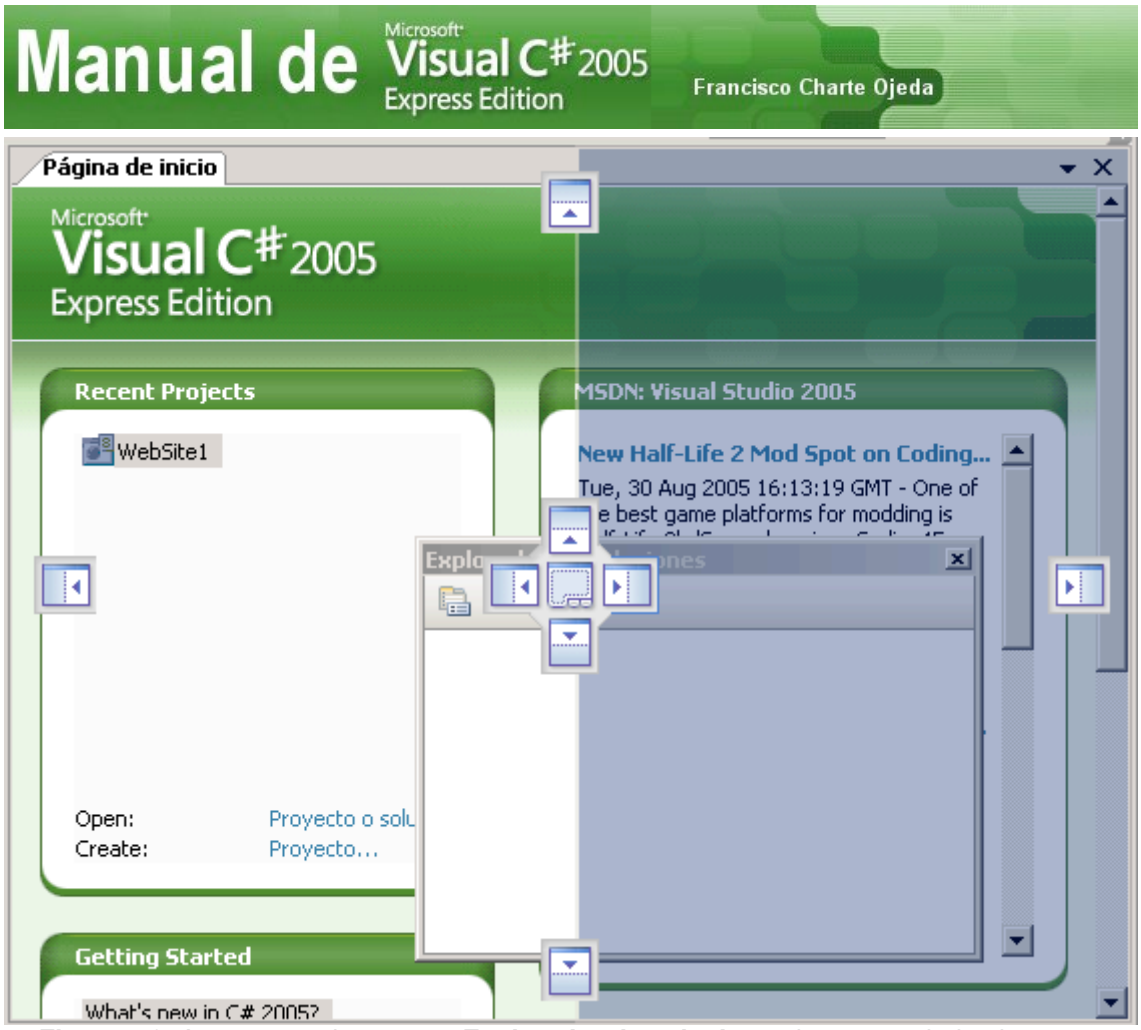

Figura 1.2. Arrastramos la ventana **Explorador de soluciones** hasta acoplarla al margen derecho del entorno.

1.1. Inicio de nuestro primer proyecto

A medida que vayamos trabajando con él, cada vez nos sentiremos más cómodos en el entorno de *Visual C# 2005 Express Edition* y aprenderemos a adaptarlo a nuestras necesidades. Por el momento ya sabemos cómo disponer las ventanas en la posición que más nos guste y establecer la configuración para que permanezcan acopladas, flotantes o se oculten automáticamente. Ahora vamos a proceder a crear nuestro primer proyecto.

Si tenemos abierta la **Página de inicio**, podemos hacer clic en el enlace **Crear proyecto** que aparece en la parte inferior de la sección **Proyectos recientes**. También podemos sencillamente pulsar el botón **Nuevo proyecto**, que por defecto es el primero de izquierda a derecha en la barra de botones; elegir la opción **Archivo>Nuevo proyecto** del menú principal o pulsar la combinación de teclas **Control-Mayús-N**. En cualquier caso siempre nos encontraremos con el cuadro de diálogo que aparece en la figura 1.3, en el que se enumeran las plantillas de proyectos que tenemos instaladas.

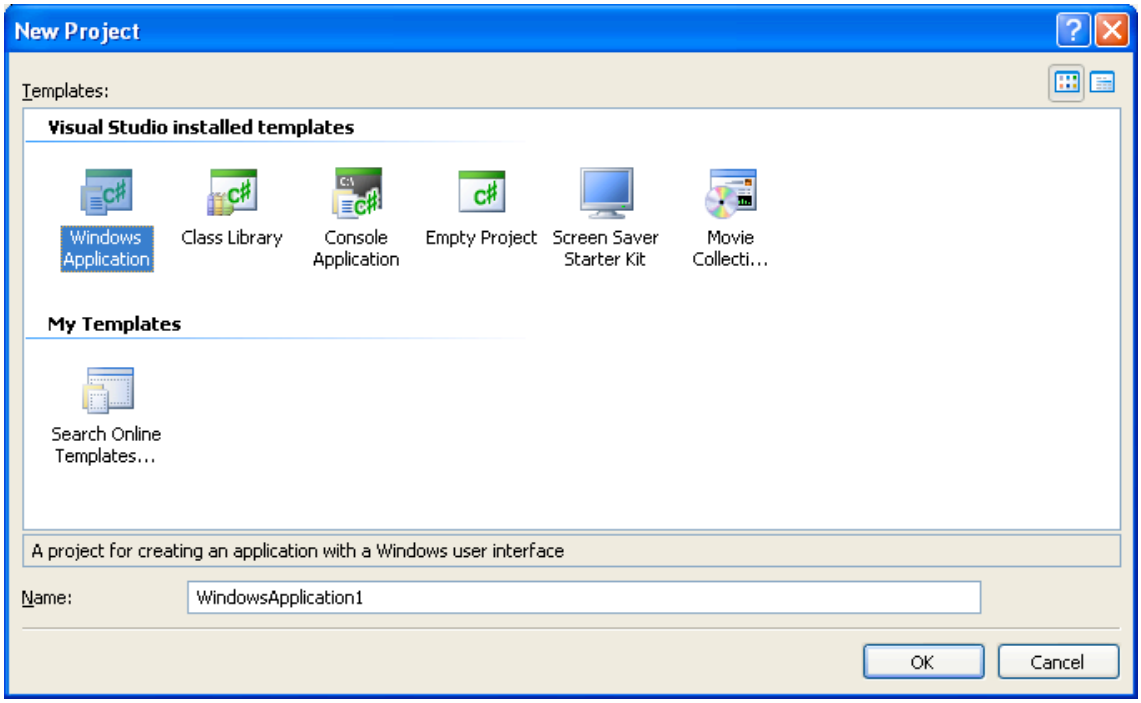

Figura 1.3. Cuadro de diálogo desde el que iniciaremos un nuevo proyecto.

Inicialmente hay disponibles media docena de plantillas, a las que se añadirán otras que podamos localizar mediante la opción **Buscar plantillas en línea** que aparece en la parte inferior. Por el momento aceptaremos las opciones propuestas por defecto: **Aplicación para Windows** con el nombre **WindowsApplication1** y pulsaremos el botón **Aceptar**. Transcurridos unos segundos aparecerán los elementos del nuevo proyecto en la ventana **Explorador de soluciones**, la ventana que hay inicialmente a la derecha, y en el área de contenido se abrirá el diseñador de formularios Windows, mostrando la ventana con que cuenta la aplicación en un principio.

1.1.1. Elementos de trabajo

Además de los que ya hay visibles, para poder componer la interfaz de usuario de nuestro primer programa necesitaremos trabajar con algunos elementos más, concretamente con el **Cuadro de herramientas** y la ventana **Propiedades**. El primero está en una ventana oculta en la zona izquierda del entorno. No tenemos más que situar el cursor sobre la pestaña que tiene el título de la ventana y, a continuación, hacer clic en el botón adecuado para dejarla adosada. Para hacer visible la segunda usaremos la opción **Ver>Ventana Propiedades**. En este momento el entono debería tener un aspecto similar al de la figura 1.4 (página siguiente).

En el **Cuadro de herramientas** se encuentran los diversos objetos que podemos utilizar para componer nuestra aplicación. En cierta manera son como los materiales para una construcción, teniendo cada uno de ellos unas propiedades y finalidad concretos. Dada la extensa lista de componentes que existen, en esta ventana aparecen clasificados en varios grupos: controles comunes, contenedores, menús y barras de herramientas, impresión, diálogos, etc. El título de cada categoría cuenta con un botón, a su izquierda, que nos permite cerrarlas y abrirlas según convenga.

El área central es donde aparece el diseñador propiamente dicho. La ventana mostrada, con el título por defecto **Form1**, es la base a la que iremos añadiendo componentes para diseñar la interfaz. A este elemento se le llama habitualmente *formulario*.

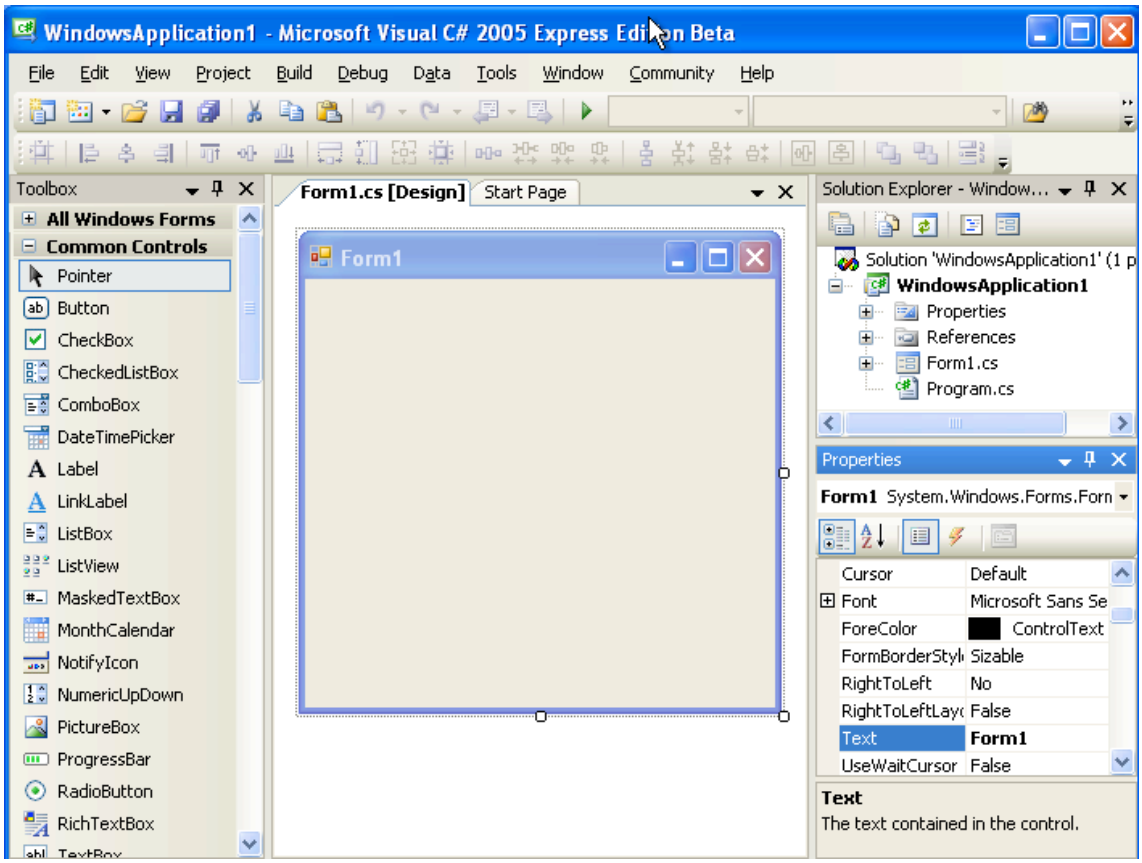

Figura 1.4. El entorno con los elementos habituales de trabajo durante el diseño de una interfaz basada en formularios Windows.

Los formularios son un componente más de Visual C# 2005 Express Edition*, con la particularidad de que actúan como contenedores, pueden alojar a otros componentes en su interior, y el entorno cuenta con diseñadores específicos para trabajar sobre ellos.*

A la derecha, en la parte superior, está el **Explorador de soluciones**, la ventana en la que se enumeran los elementos que conforman el proyecto sobre el que estamos trabajando. Cada uno de esos elementos cuenta con un menú emergente que debemos explorar para conocer todas las opciones disponibles: agregar nuevos elementos al proyecto, abrir un módulo en su diseñador o en el editor de código, etc.

Finalmente, debajo de la anterior, encontramos la ventana **Propiedades**. Ésta contiene la lista de propiedades del elemento que tengamos en cada momento seleccionado en el diseñador, facilitando su modificación. En este momento, estando el formulario vacío y, por tanto, siendo el único componente disponible, podemos usar la citada ventana por ejemplo para cambiar el título por defecto, sencillamente cambiando el valor que contiene la propiedad llamada `Text` por cualquier otro, por ejemplo

`Mi primer proyecto`. En cuanto pulse **Intro** comprobará que el título mostrado en la ventana cambia. De manera análoga podríamos alterar otras características de la ventana, pero para ello antes deberíamos conocer cuál es la utilidad de cada una de las propiedades. Muchas de ellas las conocerá en los capítulos siguientes.

1.1.2. Confección de la interfaz de usuario

Nuestras pretensiones en cuanto a la funcionalidad de este primer proyecto son muy humildes: simplemente queremos poder introducir un nombre, hacer clic sobre un botón y recibir un educado saludo como respuesta. Para alcanzar este objetivo tendremos que introducir en la ventana los componentes que formarán la interfaz de usuario, al menos un recuadro de texto en el que poder introducir el nombre y el mencionado botón.

Buscamos en el **Cuadro de herramientas** el control `TextBox`, nombre que recibe el típico recuadro de texto. Hacemos clic sobre él y a continuación desplazamos el puntero del ratón hasta situarlo, ya en el formulario, en el punto donde queremos insertarlo (véase la figura 1.5).

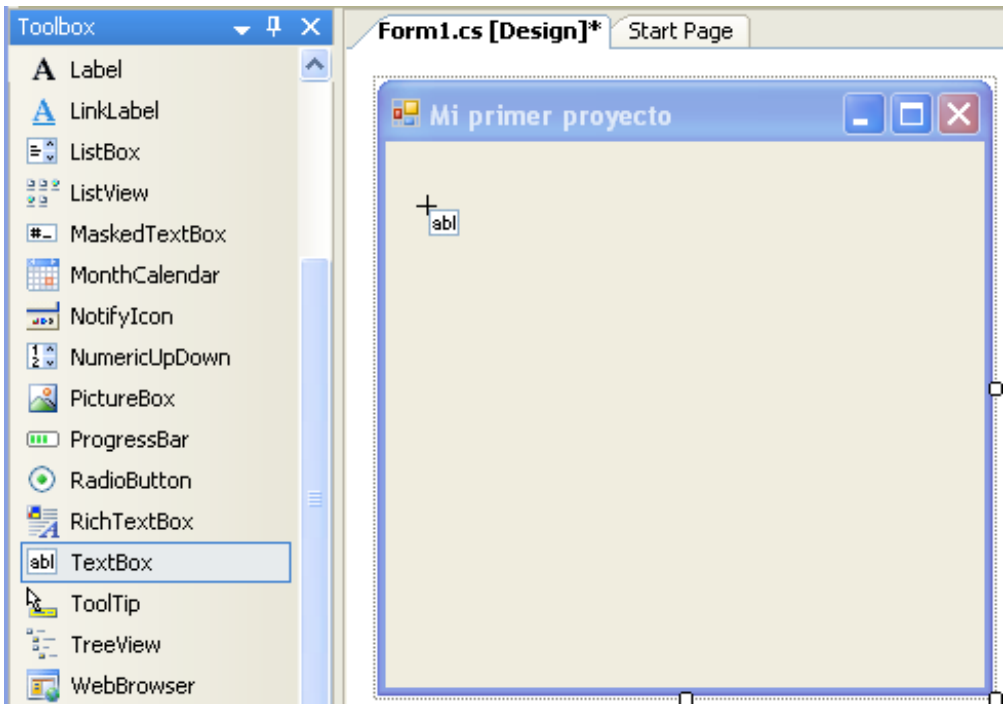

Figura 1.5. Seleccionamos el control a insertar y situamos el puntero en la posición donde deseamos colocarlo.

Denominamos componentes a todos aquellos objetos que pueden tomarse del ***Cuadro de herramientas*** *y colocarse en un diseñador, a fin de editar sus propiedades. Los componentes se dividen en dos grupos: los que forman parte de la interfaz de usuario, como botones, recuadros de texto y listas, y aquellos que aportan funcionalidad pero no son visibles en ejecución. A los primeros se les llama normalmente* controles *para diferenciarlos.*

Al hacer clic en la superficie del formulario veremos aparecer un recuadro de texto con unas dimensiones por defecto. Podemos usar los recuadros que aparecen en los extremos izquierdo y derecho (véase la figura 1.6) para hacer el recuadro más o menos grande.

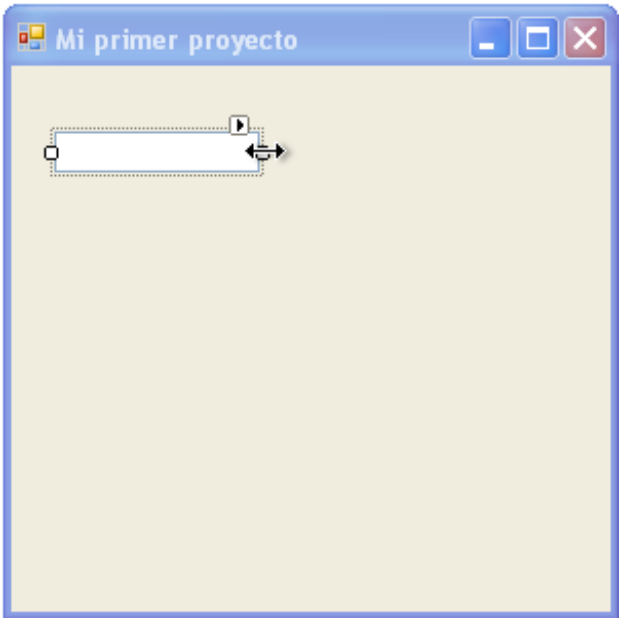

Figura 1.6. Ajustamos las dimensiones del control.

Finalmente modificaremos la propiedad `Text` de este control para que en el interior del recuadro de texto aparezca una indicación invitando a introducir el nombre. Sencillamente buscamos la citada propiedad en la ventana **Propiedades**, hacemos clic sobre ella e introducimos el texto, como puede verse en la figura 1.7 (página siguiente). Cuando ejecutemos el programa, y podamos entonces introducir un texto directamente en el recuadro como haría cualquier usuario, recurriremos a la misma propiedad `Text` para recuperarlo y procesarlo.

Siguiendo el mismo procedimiento que acaba de explicarse, localizaremos en el **Cuadro de herramientas** el control `Button`, haremos clic sobre él para seleccionarlo, colocaremos el puntero del ratón debajo del recuadro de texto y haremos clic de nuevo para colocarlo allí. Ya tenemos un botón, si bien muestra un título no muy explicativo. Cambiamos su propiedad `Text` y conseguimos finalmente una interfaz como la de la figura 1.8 (página siguiente). Hemos terminado la primera fase del desarrollo de nuestra aplicación: la composición de la interfaz de usuario.

Si en este momento ejecutásemos el proyecto, sencillamente pulsando la tecla **F5**, comprobaríamos cómo el programa muestra la ventana tal cual la hemos diseñado e, incluso, podemos modificar el contenido del recuadro de texto y pulsar el botón. Sin embargo no obtendremos respuesta alguna por parte del programa. Es lógico, puesto que aún no le hemos dicho qué queremos que haga cuando se actúe sobre el botón ni para qué pretendemos usar el valor que se haya recogido en el recuadro de texto. Para todo esto tendremos que escribir nuestra primera línea de código en lenguaje C#.

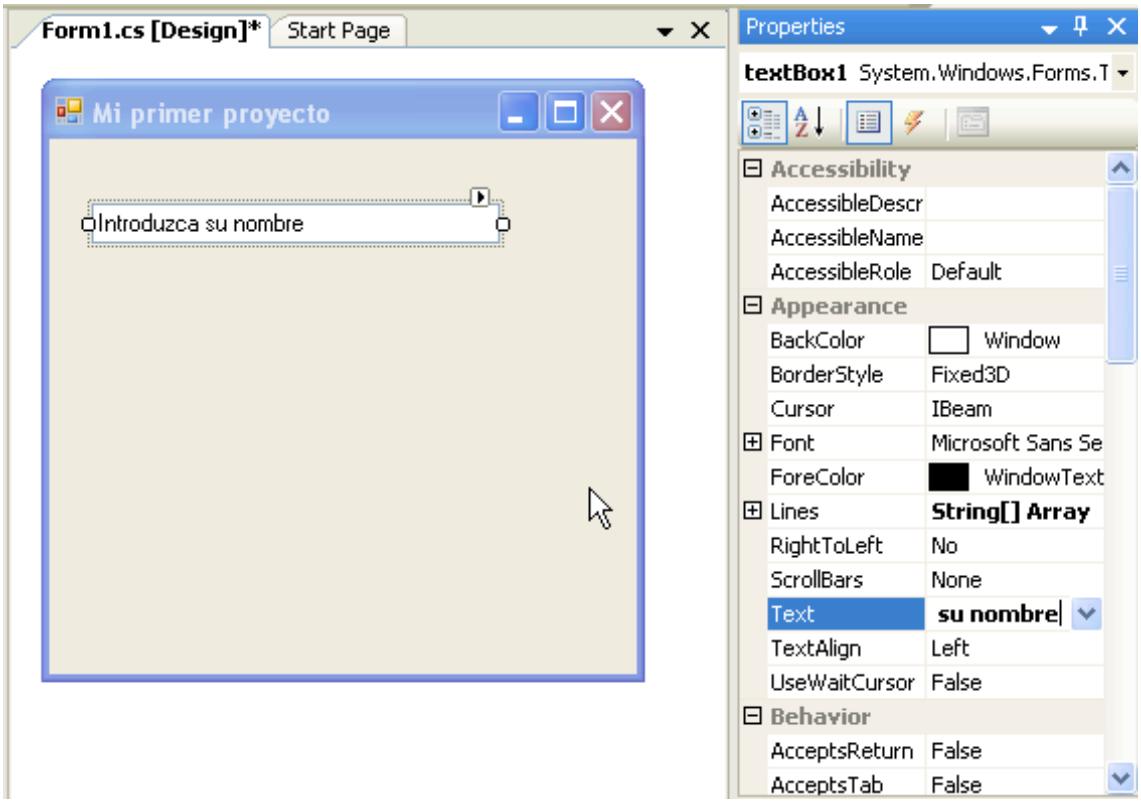

Figura 1.7. Modificamos la propiedad `Text` del recuadro de texto.

Figura 1.8. Aspecto de la interfaz de usuario de nuestro primer programa.

1.1.3. Proceso de eventos

Cuando utilizamos un programa, no importa cuál sea, todas nuestras acciones: movimiento del puntero del ratón, pulsación de sus botones, pulsación de las teclas del teclado, son traducidos por el sistema operativo en una serie de mensajes que envía a

la aplicación destinataria. Aunque podríamos, no tenemos necesidad de procesar esos mensajes, generalmente de bajo nivel, nosotros mismos, sino que es la plataforma de servicios .NET, conjuntamente con todos sus componentes, la que se encarga de tratarlos adecuadamente y, si procede, facilitarlos a la aplicación en forma de eventos.

Un evento, por tanto, es una señal que llega al programa para indicarle que ha tenido lugar algún tipo de suceso, aportando información sobre el mismo: qué tecla se ha pulsado, hasta qué posición se ha desplazado el puntero del ratón, qué elemento de una lista ha sido seleccionado, etc. Cuando se pulsa un botón, el evento que se produce se llama `Click` y, si nos interesa, es posible asociar un método propio con código que se ejecutaría en el momento en que se genere dicho evento. Esto es, precisamente, lo que nos interesa en este caso concreto.

Cada uno de los componentes de *Visual C# 2005 Express Edition* cuenta con una lista concreta de eventos, algunos de los cuales son genéricos (todos los controles disponen de ellos) y otros específicos. Para ver esa lista de eventos no tenemos más que hacer clic en el botón **Eventos** que hay en la parte superior de la ventana **Propiedades** (véase la figura 1.9). Para volver a ver las propiedades usaríamos el botón **Propiedades** que hay a la izquierda. Haciendo doble clic sobre cualquiera de los eventos, suponiendo que conociésemos la finalidad que tiene, generaríamos el método asociado y podríamos escribir el código que deseamos ejecutar cuando tenga lugar.

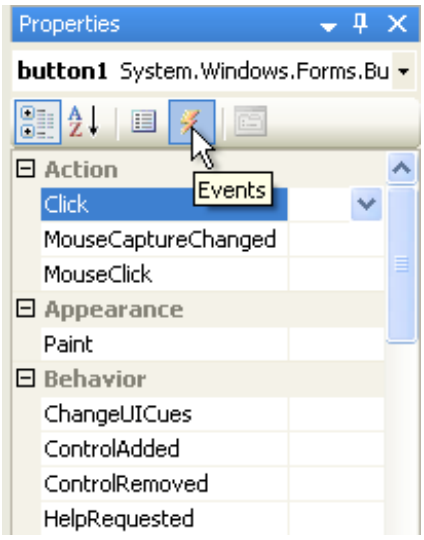

Figura 1.9. Lista de eventos del componente `Button`.

De todos los eventos con que cuenta cada control, uno de ellos suele ser el utilizado con mayor frecuencia. En el caso de los botones, tanto el control `Button` como otros tipos de botones existentes, ese evento es `Click`, porque lo que interesa de un botón es saber cuándo ha sido pulsado. Dicho evento es conocido como *evento por defecto*, teniendo la particularidad de que puede accederse a él sin necesidad de recurrir a la lista de eventos de la ventana **Propiedades**, basta con hacer doble clic sobre él en el diseñador.

En el caso concreto de este ejemplo, por tanto, haríamos doble clic sobre el botón insertado previamente. Veríamos cómo de inmediato se abre el editor de código, apareciendo el cursor (véase la figura 1.10) colocado en el interior de un método llamado `button1_Click`. Éste será el método que se ejecutará cada vez que se genere el

evento `Click` del botón, es decir, cada vez que se pulse el botón. Cualquier sentencia que introduzcamos aquí, por tanto, se ejecutará en el momento en que se pulse el botón.

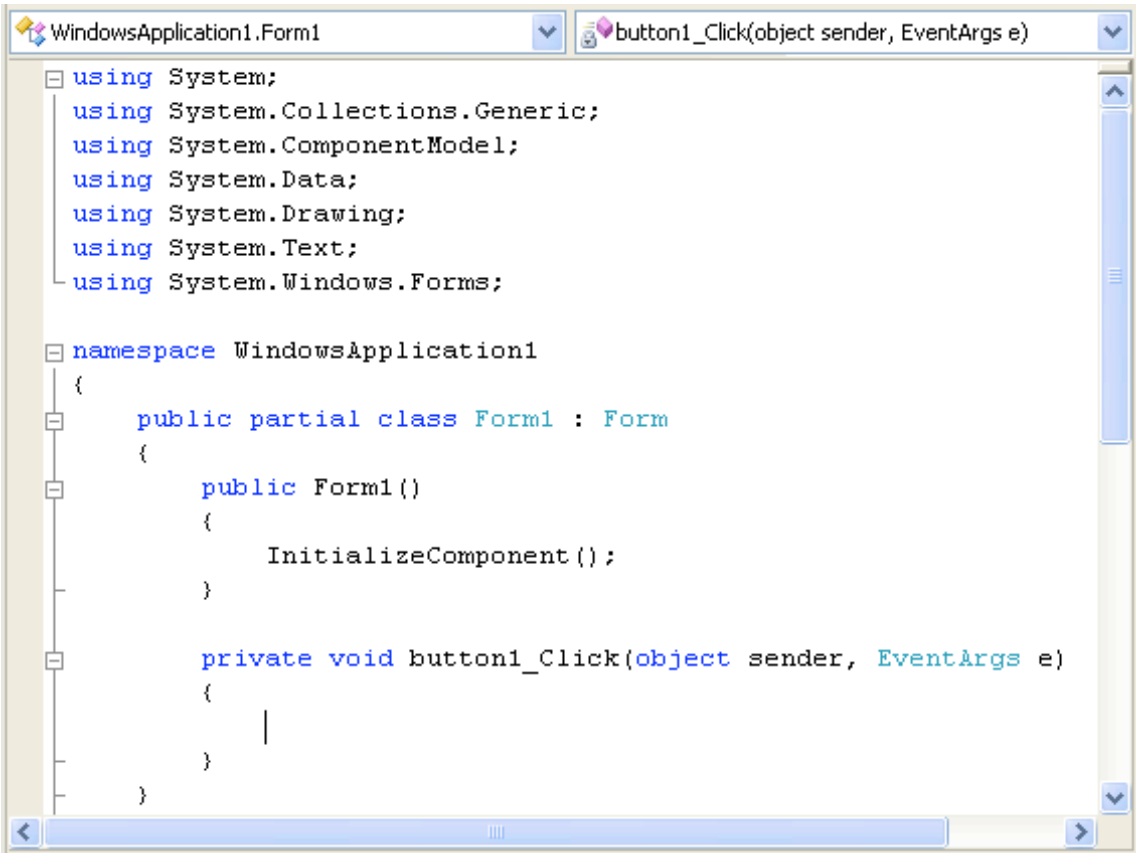

Figura 1.10. Al hacer doble clic sobre el botón se abre el editor de código.

1.1.4. Edición de código

El cursor se encuentra colocado en el editor de código en el punto adecuado para que introduzcamos las sentencias que deben ejecutarse al pulsar el botón. Lo que queremos hacer es tomar el contenido de la propiedad `Text` del recuadro de texto, añadirle un texto complementario y después mostrar el resultado en una ventana.

Cada uno de los componentes que se introducen en un diseñador, como el botón y el recuadro de texto de nuestro ejemplo, reciben un nombre que se almacena en su propiedad `Name`. Ese nombre puede ser el que a nosotros nos interese pero, puesto que no lo hemos modificado, seguirá por defecto un patrón que establece que el nombre será la clase o tipo de control seguida de un número de 1 en adelante. Así, el recuadro de texto, control que se llama `TextBox`, tendrá el nombre `textBox1`, mientras que el botón, control `Button`, por defecto se llamará `Button1`.

Para acceder a un miembro que pertenece a un cierto objeto, sin importar la categoría del miembro (propiedad, método o evento) o el tipo del objeto, siempre usaremos la siguiente notación:

```
NombreObjeto.NombreMiembro
```

Dado que en este caso lo que queremos es leer la propiedad `Text` del control `textBox1` y añadirle un texto, la sentencia a escribir sería similar a la siguiente:

```
"Texto " + textBox1.Text;
```

De esta manera componemos una cadena de caracteres formada por el texto facilitado como literal, entre comillas dobles, y el contenido del recuadro de texto. El operador +, por tanto, en este contexto lo que hace es concatenar dos cadenas para formar una única.

En el lenguaje C# los retornos de carro no tienen ningún significado especial, por lo que una sentencia puede contar con tantas líneas físicas como sea preciso. Esto implica, sin embargo, que debe existir algún marcador de fin de sentencia. Como en casi todos los lenguajes de la familia de C/C++, dicho marcador es el punto y coma que dispondremos al final de cada sentencia.

Para que esa cadena de caracteres aparezca en una ventana, que es nuestro objetivo, nos serviremos de una clase llamada `MessageBox`, concretamente del método `Show` de dicha clase. No tenemos más que entregar como argumento la cadena y este método la mostrará en una pequeña ventana independiente de la del programa. Por tanto la sentencia que vamos a escribir es la siguiente:

```
MessageBox.Show("Hola " + textBox1.Text);
```

IntelliSense, la ayuda inteligente a la escritura de código

Volvamos al editor de código, la ventana que se había abierto al hacer doble clic sobre el botón, y comencemos a escribir la sentencia que acaba de indicarse. De inmediato, al introducir los primeros caracteres de `MessageBox`, se abrirá una lista ordenada de posibles elementos a los que podemos hacer referencia, concretamente todos aquellos cuyo nombre comienza por la secuencia de caracteres que ya hemos escrito. En la figura 1.11 (página siguiente), por ejemplo, se han escrito los caracteres `Messa`, tras lo cual se ha usado el cursor hacia abajo para elegir de la lista la clase `MessageBox`. No hace falta terminar de escribir la referencia, basta con pulsar **Intro** o, en este caso, el punto intermedio para acceder a uno de sus miembros.

Cuando tras el nombre de una clase, objeto, estructura o interfaz introducimos un punto, el editor automáticamente mostrará la lista de los miembros que podemos utilizar según el contexto que se deduce de la sentencia actual. De esta manera es posible ahorrar bastante tiempo a la hora de introducir código, aparte de no ser imprescindible recordar de memoria todas las propiedades, métodos y eventos de todos los objetos existentes.

De manera análoga, al abrir los paréntesis para ir a invocar a un método (véase la figura 1.12 en la página siguiente) el editor muestra en una pequeña ventana emergente la lista de argumentos que debemos entregar. En caso de que el método cuente con varias versiones distintas, cada una de ellas aceptando una lista de parámetros diferente, podemos usar las teclas de desplazamiento del cursor arriba y abajo para seleccionar aquella en la que estemos interesados.

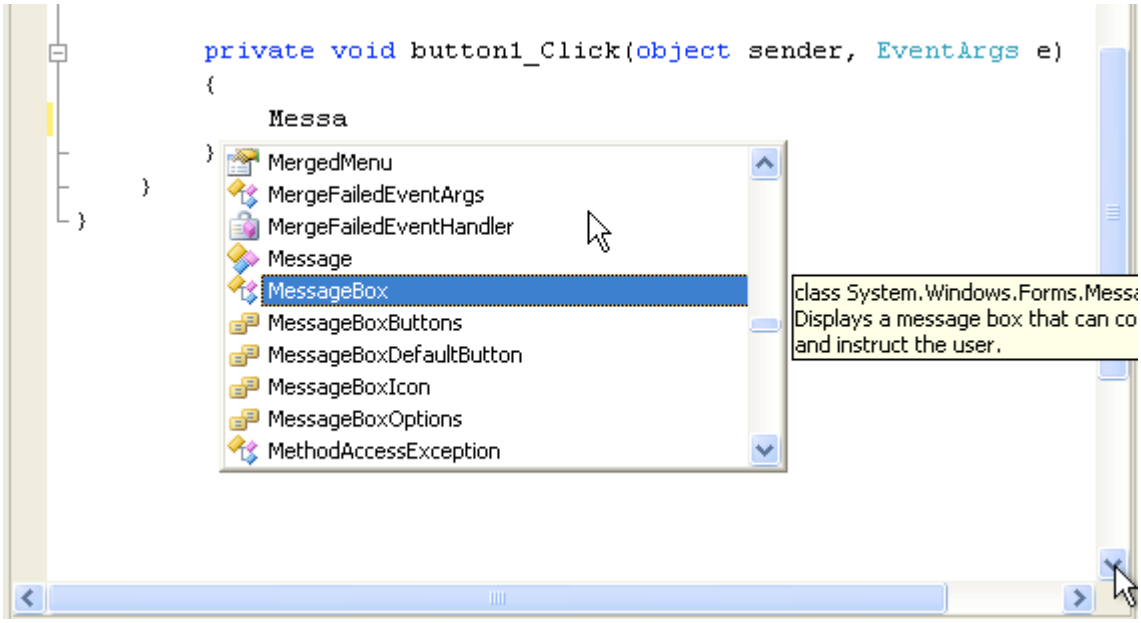

Figura 1.11. Lista de elementos a los que podemos hacer referencia según el contexto actual.

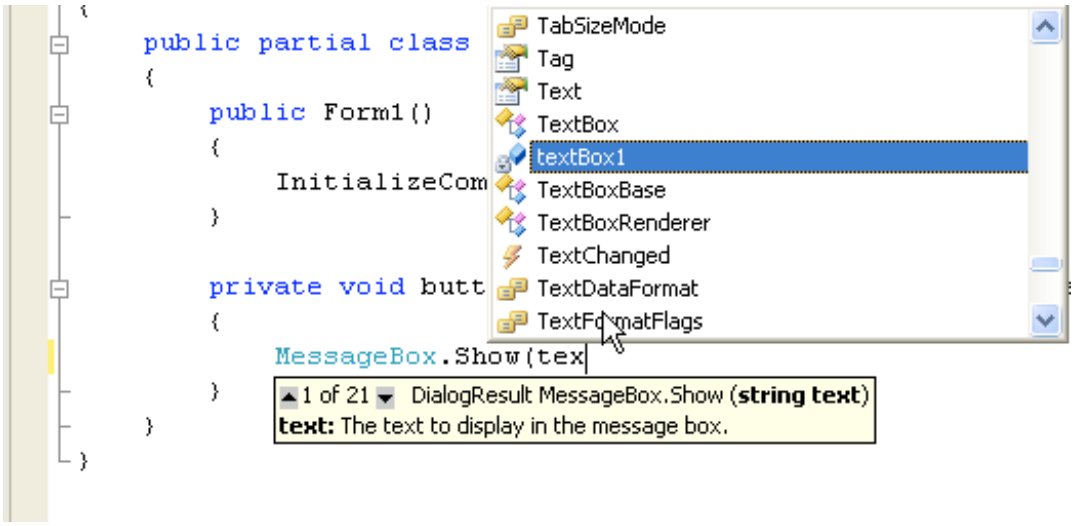

Figura 1.12. Al llamar a un método se abre una ventana con la lista de parámetros a entregar.

Todas estas ayudas a la edición de código, que consiguen hacer mucho más cómodo nuestro trabajo, se denominan genéricamente *IntelliSense*.

1.1.5. Ejecución del programa

Diseñada la interfaz y asociado el código al evento `Click` del botón, nuestro programa ya está terminado. Podemos proceder a ejecutarlo para comprobar si su funcionamiento es el que esperábamos. Para ello podemos pulsar la tecla **F5**, hacer clic en el botón  (**Iniciar depuración**) o bien recurrir a la opción **Depurar>Iniciar depuración**. En cualquier caso se procederá a compilar el proyecto e iniciar su ejecución, controlando cualquier posible error que se produjese. En este caso no debe haber ningún problema, dada la sencillez de la propuesta, y veremos aparecer la ventana que habíamos diseñado. Tras introducir un nombre y hacer clic en el botón veremos aparecer el mensaje de saludo.

Figura 1.13. El programa en funcionamiento.

Tenemos en funcionamiento nuestro primer programa escrito con *Visual C# 2005 Express Edition*, un programa sencillo pero que con otros lenguajes y herramientas nos hubiese supuesto mucho más trabajo.

1.2. Administración de los elementos del proyecto

El sencillo proyecto que hemos desarrollado a modo de ejemplo está compuesto de varios módulos, a los que pueden ir añadiéndose muchos otros a medida que sea necesario. El **Explorador de soluciones** es la herramienta que nos permitirá administrar los distintos elementos que conformen nuestro proyecto, abriéndolos en el editor o diseñador que corresponda, estableciendo opciones específicas, añadiéndolos y eliminándolos.

Al elegir cualquiera de los elementos, por ejemplo el módulo `Form1.cs` que corresponde al formulario, en la parte superior de la ventana aparecerán unos botones que facilitan el acceso al código y al diseñador, al tiempo que en la ventana **Propiedades** podemos establecer las opciones de proceso a la hora de compilarlo.

Cada elemento de los enumerados en el **Explorador de soluciones**, incluyendo el nodo que corresponde al propio proyecto, dispone de un menú emergente con opciones especificas. En la figura 1.14 (página siguiente), por ejemplo, puede ver que con ese menú se facilita la compilación del proyecto, su publicación para que los usuarios puedan obtenerlo, la adición de nuevos elementos y referencias a objetos externos, la depuración, etc.

A pesar de que nosotros hemos escrito una sola línea de código para desarrollar nuestro programa, lo cierto es que el proyecto se compone de una cantidad bastante mayor de código, todo él generado por el asistente de *Visual C# 2005 Express Edition* que usamos al principio para crear el proyecto. Esos módulos son los responsables, por ejemplo, de crear el formulario y los controles, establecer sus propiedades, etc.

Los diseñadores se encargan de ir adaptando el código generado inicialmente, agregando los componentes que se inserten e ir introduciendo las sentencias apropiadas para configurar las propiedades. Desde el **Explorador de soluciones** podemos abrir cualquiera de esos módulos, con un simple doble clic, y examinar el trabajo que el entorno hace para nosotros.

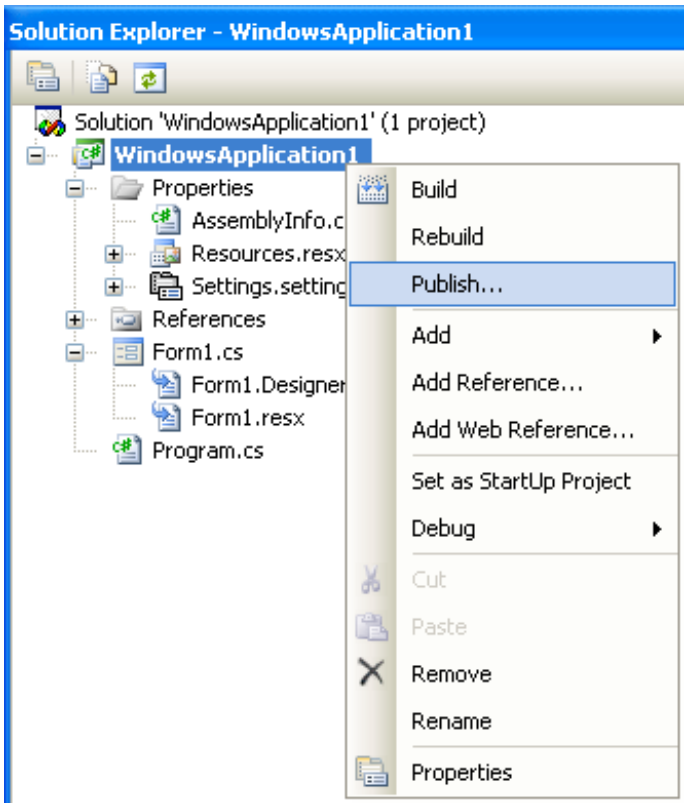

Figura 1.14. Menú emergente asociado al proyecto.

1.3. Opciones de depuración

Aunque en esta ocasión concreta, dada la simplicidad del ejemplo, no ha necesitado recurrir a la depuración del proyecto, en la práctica, con proyectos reales, el proceso de depuración se convierte en un paso más de todo el ciclo de desarrollo. En el entorno encontraremos todas las opciones disponibles para ejecutar los programas paso a paso, viendo el resultado que se genera con cada sentencia mientras examinamos el contenido de variables y objetos. Hay opciones para establecer puntos de parada con condiciones avanzadas, configurar examinadores de datos, ver la pila de llamadas hasta llegar al punto actual, etc.

Para ver en la práctica cómo usar algunas de estas opciones, encontrándose en el editor de código haga clic con el botón principal del ratón en el margen izquierdo de la única sentencia que hemos escrito, concretamente en el punto en que aparece un círculo de color rojo en la imagen 1.15. Con esta sencilla acción acabamos de establecer un punto de parada incondicional, lo cual significa que la ejecución del programa se detendrá en cuando llegue a este punto, quedando a la espera de nuestras instrucciones. Un nuevo clic en el mismo punto desactivaría el punto de parada y la línea volvería a su color normal.

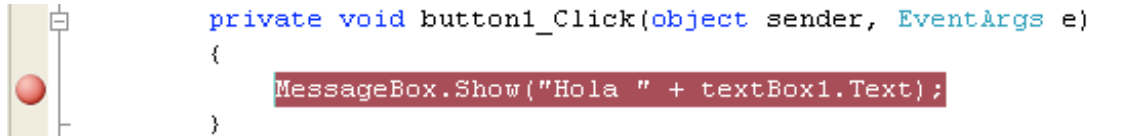

Figura 1.15. Establecemos un punto de parada en la única línea de código que hemos escrito.

A continuación iniciamos la ejecución del programa con la tecla **F5** o cualquiera de las otras opciones indicadas antes. Aparece la ventana, introducimos un nombre y pulsamos el botón, momento en el que se detiene la ejecución y el entorno de desarrollo pasa a primer plano mostrando un aspecto similar al de la figura 1.16.

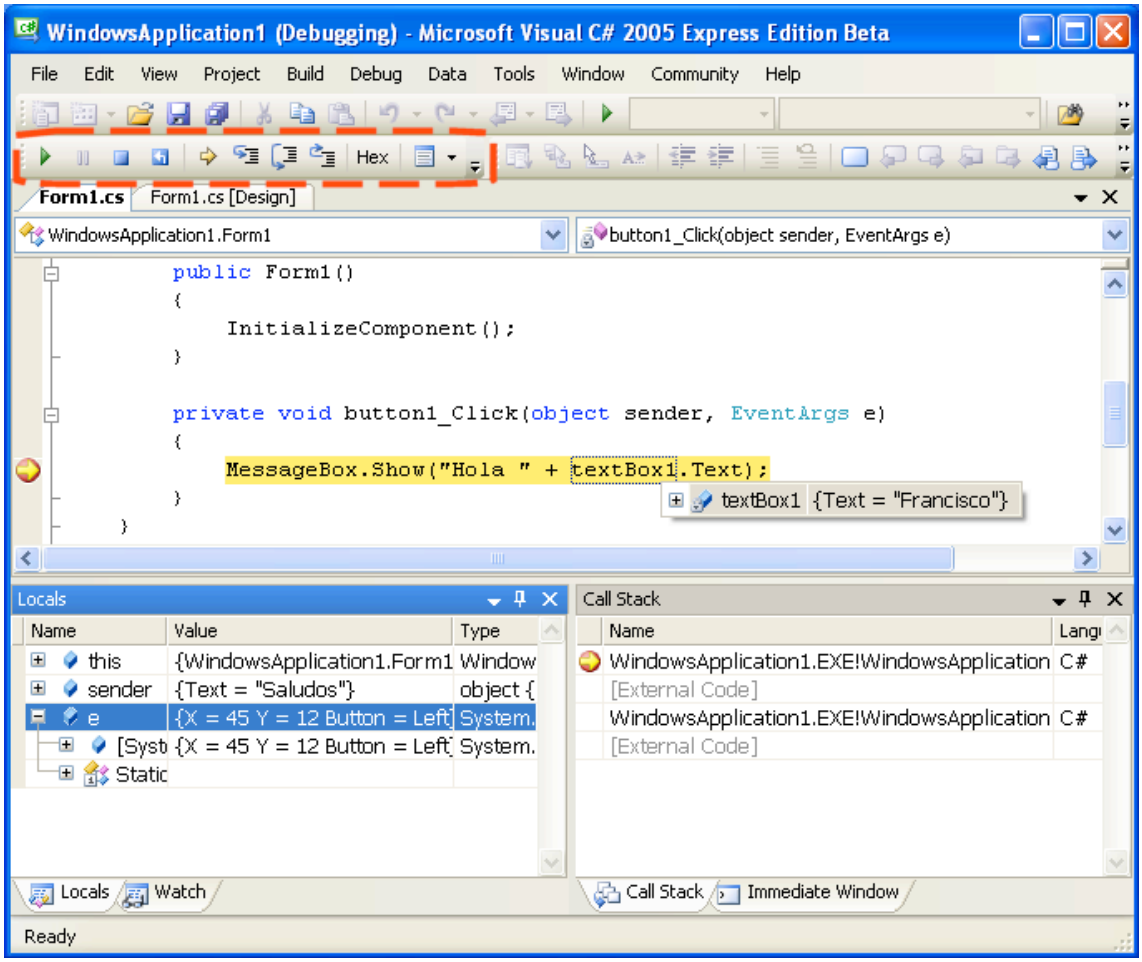

Figura 1.16. Aspecto del entorno de desarrollo en modo de depuración.

La sentencia que iba a ejecutarse, justo en el momento en que se ha detenido el programa, es la que aparece destacada y con una flecha en el margen izquierdo. En la parte inferior hay varias ventanas con información diversa. La de la izquierda enumera las variables locales correspondientes al contexto de ejecución actual, en este caso el interior del método `button1_Click`, mostrando el nombre y contenido de cada una de ellas.

Para obtener el contenido de cualquier variable o propiedad a la que se haga referencia en el texto, por ejemplo la propiedad `Text` del control `textBox1`, basta con situar el puntero del ratón sobre ella. En ese momento aparece una ventana flotante con la información pertinente.

Usando la barra de botones específica para depuración, destacada con un contorno rojo en la figura 1.16, podemos continuar la ejecución del programa, detenerla completamente, ejecutar paso a paso, etc. También es posible recurrir a las opciones del menú emergente, así como a las del menú **Depurar**, para llevar a cabo todas esas tareas.

1.4. Publicación del proyecto

Aunque por el momento estamos aprendiendo a usar *Visual C# 2005 Express Edition*, y por tanto los proyectos que vayamos desarrollando sean principalmente para nosotros mismos o simples programas de prueba, con el tiempo crearemos aplicaciones para terceros. En ese momento necesitaremos ofrecer el proyecto en algún formato que permita a los usuarios finales su instalación y uso. Gracia a *ClickOnce*, una de las novedades de esta nueva versión del producto, esa tarea resulta más fácil que nunca.

Para iniciar la publicación del proyecto tendrá que elegir la opción **Publicar**, ya sea del menú emergente del proyecto (en el **Explorador de soluciones**) o bien del menú **Generar**. En cualquier caso se pondrá en marcha un asistente de varios pasos. En el primero deberemos introducir el destino donde se alojarán los archivos de instalación del programa, un destino que puede ser un directorio de una unidad de nuestro equipo, un servidor Web, un servidor FTP o una unidad compartida de red. Dependiendo de la opción elegida, los pasos siguientes pueden diferir ligeramente.

Asumiendo que hemos facilitado como destino el camino completo de una carpeta, en el segundo paso podremos elegir entre tres formas distintas de instalación: desde un servidor Web, desde una unidad compartida de red o a través de CD/DVD. La primera opción es muy interesante, ya que permite la instalación remota de aplicaciones sencillamente introduciendo un URL en un navegador. La tercera (véase la figura 1.17, en la página siguiente) copiará los archivos en la carpeta que hayamos indicado, desde ahí podremos grabarlos en un CD/DVD y distribuirlos siguiendo el procedimiento más clásico.

Independientemente de cuál sea el procedimiento elegido para la instalación, comprobaremos que en la carpeta o servidor que hubiésemos indicado como destino aparecerá un archivo `setup.exe` junto con otros dos archivos y una carpeta con el código de la aplicación. La instalación se efectúa ejecutando el programa `setup.exe`, como es habitual, lo cual agregará una nueva entrada en el menú **Programas** del usuario. Éste puede desinstalar en cualquier momento el programa como haría con cualquier otra aplicación, abriendo **Panel de control** y usando la opción **Agregar o quitar programas**.

En el caso de que optemos por poner los archivos de instalación en un servidor Web, deberemos comunicar a los usuarios el URL completo que deberán usar para acceder a la aplicación. Por suerte el propio asistente de publicación se encarga de generar una página con esta finalidad, por lo que solamente tenemos que colocarla en el lugar adecuado y los clientes podrán acceder al programa y ejecutarlo prácticamente como si fuese una aplicación Web.

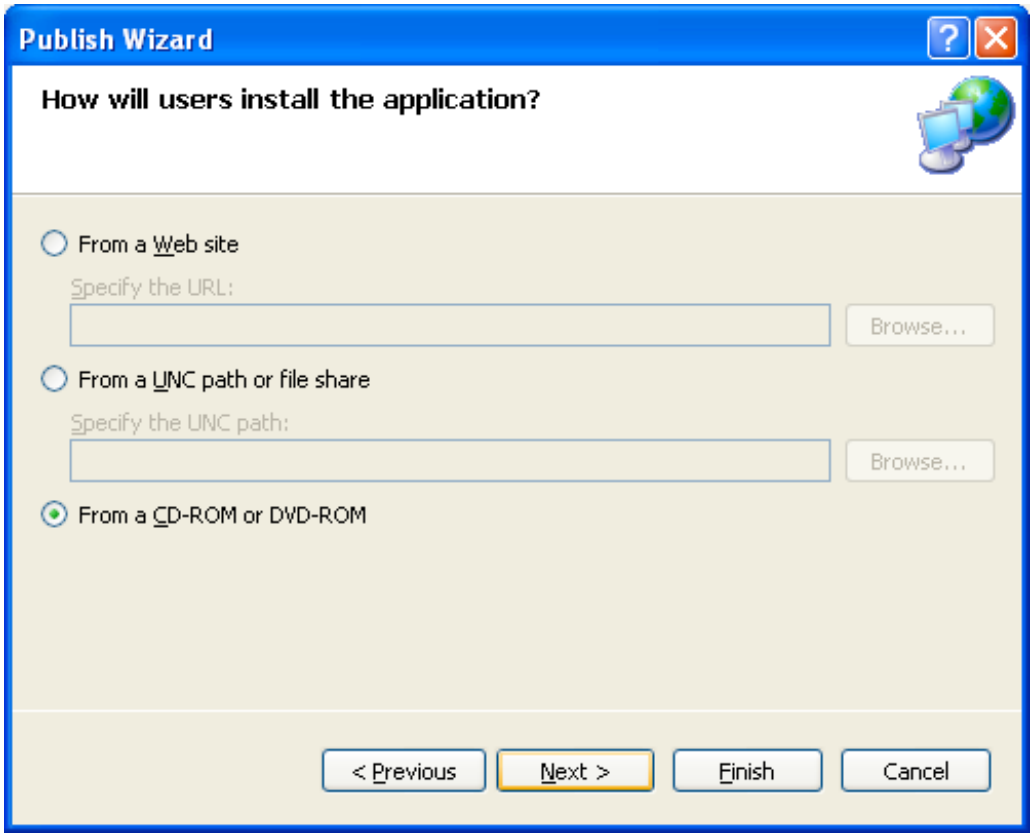

Figura 1.17. Seleccionamos el origen desde el que instalarán la aplicación los usuarios.

Resumen

Al finalizar este primer capítulo ya nos hemos familiarizado con el entorno de desarrollo de *Visual C# 2005 Express Edition*, habiendo desarrollado un primer proyecto que hemos ejecutado, depurado y publicado para que otros usuarios puedan instalarlo en sus equipos. Hemos conocido muchos elementos de la herramienta, como el editor de código, el diseñador de formularios Web, el **Cuadro de herramientas**, el **Explorador de soluciones** o la ventana **Propiedades**. Todos ellos le serán útiles de aquí en adelante, independientemente del tipo de proyecto que aborde.

También hemos adquirido unos conceptos, muy básicos, sobre la sintaxis del lenguaje C#. Sabemos, por ejemplo, que el final de las sentencias viene marcado por el carácter punto y coma, así como que para acceder a un miembro de un objeto se colocan los identificadores de objeto y miembro separados por un punto. En el capítulo siguiente profundizaremos en los aspectos más interesantes del lenguaje C#.

2. Nos familiarizamos con el lenguaje

C# es uno de los lenguajes más modernos que podemos encontrar en la actualidad, especialmente cuando se le compara con otros de la misma familia, como C y C++, o incluso respecto a otros de nacimiento también reciente, como Java. Un nuevo lenguaje de programación no se desarrolla sencillamente porque sí, existiendo siempre razones suficientes que justifiquen la gran inversión que supone un esfuerzo de este tipo.

Hasta la llegada de C# no existía un lenguaje de programación verdaderamente orientado al uso y creación de componentes, siendo quizá ésta su aportación más importante. Esta orientación se refleja, por ejemplo, en el uso de atributos declarativos para establecer ciertas propiedades de los objetos, sin necesidad de escribir código funcional.

El objetivo de este capítulo es conseguir que nos familiaricemos con el lenguaje C#, recorriendo para ello las construcciones de uso más habitual. Obviamente resulta imposible abordar en tan poco espacio todas las posibilidades del lenguaje, todos los detalles y explicarlos debidamente. Para esto, no obstante, está la documentación de referencia sobre el lenguaje C# y la propia especificación del estándar.

2.1. OOP y COP

C# es un lenguaje de programación basado fuertemente en C++, aunque tiene influencias de muchos otros, por lo que ya es, de partida, un lenguaje OOP (*Object Oriented Programming*) o de programación orientada a objetos. En este sentido las construcciones con que cuenta C# son muy similares a las de C++, utilizándose la misma sintaxis para definir clases y estructuras, declarar sus miembros, la visibilidad de acceso, etc.

Lo que diferencia a C# de C++ o Java, sin embargo, es el hecho de ser un lenguaje COP (*Component Oriented Programming*) u orientado a la programación con componentes. Esta naturaleza se denota en la existencia de construcciones en el lenguaje para definir propiedades y eventos, inexistentes en la mayoría de los lenguajes, sin necesidad de recurrir a ningún tipo de extensión.

En C#, además, no se hace una distinción estricta entre tipos intrínsecos de datos y objetos, como sí ocurre en los lenguajes mencionados, de tal forma que cualquier variable puede ser utilizada como un objeto, sea del tipo que sea. Esto es posible gracias a un mecanismo automático de conversión, en ambos sentidos (de valor a objeto y viceversa) conocido como *boxing-unboxing*.

Otra de las características diferenciadoras de C#, con la que cuentan otros lenguajes .NET, la encontramos en el uso de atributos declarativos para asignar determinadas propiedades a clases de objetos, estructuras, métodos, etc. Estos atributos se recopilan, durante la compilación del código, y almacenan como metainformación conjuntamente con el código ejecutable. A la hora de ejecutar el programa, el entorno de ejecución o CLR (*Common Language Runtime*) recupera esa información asociada al código y adecua el contexto según sea necesario.

La existencia de los atributos, y los servicios de reflexión que permiten recuperarlos durante la ejecución, hace innecesario en C# el uso de recursos habituales en otros lenguajes de programación, como las bibliotecas de tipos COM tan corrientes en Visual Basic 6 o los archivos de definición de interfaces (IDL).

A pesar de ser un descendiente de C++, en C# se ha tendido a simplificar el lenguaje y, para ello, se ha restringido el uso de las construcciones más peligrosas, como es por ejemplo la aritmética de punteros, aportando otras con la misma funcionalidad pero más sencillas y que implican menos riesgos. Todo ello contribuye a incrementar la productividad del programador, al encontrarse con muchos menos problemas durante el desarrollo de proyectos grandes o complejos.

Actualmente el lenguaje C# es un estándar ECMA (*European Computer Manufacturers Association*) e ISO (*International Organization for Standardization*), calificación que pocos lenguajes alcanzan en tan poco tiempo. Este hecho permite que cualquiera que obtenga la especificación del estándar pueda implementar un compilador totalmente compatible y, en cualquier caso, es una garantía para empresas y organizaciones en cuanto al control que se ejerce sobre la evolución de este lenguaje.

2.2. Fundamentos del lenguaje

Tras adquirir la visión general sobre el lugar que ocupa el lenguaje C# respecto a otros también muy extendidos, llega el momento de conocer los fundamentos que nos permitan usarlo para crear cualquier tipo de programa.

El código de un programa escrito en C# se almacena en un archivo de texto, la extensión por convención es `.cs`, que después es entregado al compilador de C#. Esta operación puede llevarse a cabo desde la línea de comandos o, la opción más cómoda, desde el entorno de *Visual C# 2005 Express Edition*. Independientemente del procedimiento que usemos, el compilador generará a partir del código un ensamblado conteniendo código MSIL (*Microsoft Intermediate Language*). Éste será finalmente ejecutado por el CLR o núcleo de la plataforma .NET.

En el interior de ese módulo de código fuente alojaremos definiciones de clases, estructuras y otros tipos de datos. Si pretendemos generar un ejecutable, no una biblioteca de enlace dinámico con código, una de las clases deberá contar con un método llamado `Main` que se caracterizará por ser estático. Esto significa que no será necesario crear un objeto de esa clase para poder llamar a `Main`, método que actuará

como punto de entrada al programa. En este método normalmente se procederá a crear los objetos que necesite la aplicación para realizar su trabajo.

Si abrimos en el editor el archivo `Program.cs` del proyecto de ejemplo creado en el capítulo previo, sencillamente haciendo doble clic sobre él en el **Explorador de soluciones**, veremos que contiene la definición de la clase siguiente:

```
static class Program
{
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main()
    {
        Application.EnableVisualStyles();
        Application.Run(new Form1());
    }
}
```

El nombre de la clase no tiene necesariamente que ser `Program` ni tampoco es obligatorio el modificador `static` al nivel de clase, una novedad de C# 2.0 cuya finalidad es permitir la definición de clases de las que no pueden crearse objetos. Vamos a ir analizando algunos aspectos de este código.

2.2.1. Minúsculas, mayúsculas y delimitadores

Como otros lenguajes pertenecientes a la misma familia, C# diferencia entre mayúsculas y minúsculas. Esto significa que para el compilador no es lo mismo `Main` que `main`, sino que se trata de dos identificadores distintos. También es la razón de que las palabras clave del lenguaje, tales como `static`, `class` o `void`, deban ser escritas siempre en minúsculas, como se ha establecido en la especificación del lenguaje, ya que de lo contrario el compilador las interpretaría como identificadores corrientes.

La distinción entre mayúsculas y minúsculas afecta también a los nombres de los objetos y clases predefinidos. En el código anterior, por ejemplo, `Application` es el nombre de un objeto con el que cuentan todas las aplicaciones Windows escritas con C#, nombre que debe aparecer tal y como se ve, con la inicial en mayúscula. Si usamos `application` todo irá mal.

Es importante tener en cuenta que C#, como otros lenguajes de la familia de C++, diferencia entre mayúsculas y minúsculas. El compilador puede indicar un error si cambiamos una mayúscula por una minúscula, o viceversa, al no reconocer un cierto identificador.

En cuanto a los delimitadores, los símbolos que se usan para marcar el inicio y/o fin de un elemento, los más usuales son los siguientes:

- `;` - Ya sabemos que el punto y coma marca el final de cada sentencia ejecutable de un programa. Este delimitador no aparece, sin embargo, en la línea

que actúa como cabecera de una clase o un método, porque ya existen otros delimitadores, como los paréntesis y llaves, que informan al compilador de su finalización.

- `.` - Este delimitador separa los elementos que componen una referencia compuesta, por ejemplo el nombre de una estructura y el de una de sus variables, o el de un objeto y una de sus propiedades.

- `()` - Los paréntesis, como es habitual en casi todos los lenguajes, delimitan las listas de parámetros tanto en la definición de los métodos como en su invocación.

- `{}` - Mediante las llaves se marca el inicio y fin de un bloque de código. Éste puede ser la definición de una clase, el cuerpo de un método o el conjunto de sentencias afectado por un condicional o un bucle, por mencionar los casos más usuales.

- `[]` - Con los corchetes se delimitan los atributos que afectan al elemento definido a continuación, por ejemplo `STAThread` en el caso del método `Main`, así como los índices a la hora de acceder a matrices y colecciones de datos.

A medida que vayamos escribiendo y examinando código en C# nos acostumbraremos rápidamente tanto a estos delimitadores como al correcto uso de las mayúsculas y minúsculas. Sencillamente es una cuestión de hábito.

2.2.2. Comentarios y documentación XML

C# acepta la introducción de comentarios usando distintos marcadores, siendo los más usuales `//`, que trata como comentario desde el punto en que aparezca hasta el final de la línea física, y la pareja `/*` y `*/`, con la que se trata como comentario todo lo que aparezca entre ellos.

A éstos hay que añadir una tercera posibilidad: el marcador `///`. Éste es similar a `//` en el sentido de que trata como comentario hasta el final de la línea, pero tiene la peculiaridad de ser usado para introducir en el propio código la documentación asociada, usando para ello una serie de etiquetas XML. Estos comentarios pueden extraerse posteriormente y generar una verdadera documentación.

En el fragmento de código anterior puede ver cómo el método `Main` está precedido de un bloque de comentarios `///`, concretamente una etiqueta `summary` aportando el resumen de la finalidad del método. De manera similar podrían utilizarse las marcas `param` y `returns` para explicar la lista de parámetros de entrada y el valor de retorno, `remarks` para introducir comentarios o `example` para facilitar un ejemplo.

A medida que vaya introduciendo comentarios con el marcador `///`, IntelliSense *se encargará de introducir el marcador en cada nueva línea de manera automática y de facilitarle una lista de las etiquetas de documentación disponibles (véase la figura 2.1, en la página siguiente).*

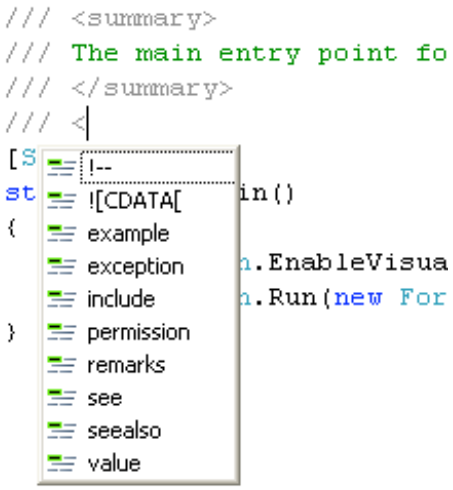

Figura 2.1. *IntelliSense* nos ayuda a documentar el código.

Para que los comentarios que vayamos introduciendo se conviertan en una documentación palpable, tendremos que abrir el menú emergente asociado al proyecto (en el **Explorador de soluciones**) y seleccionar la opción **Propiedades**. Esto hará aparecer en el área central una página con múltiples pestañas, conteniendo cada una de ellas una categoría de opciones que afectan al proyecto. En la pestaña **Generar** encontraremos la opción **Archivo de documentación XML** (véase la figura 2.2), bastará con marcarla para obtener el archivo XML con la documentación.

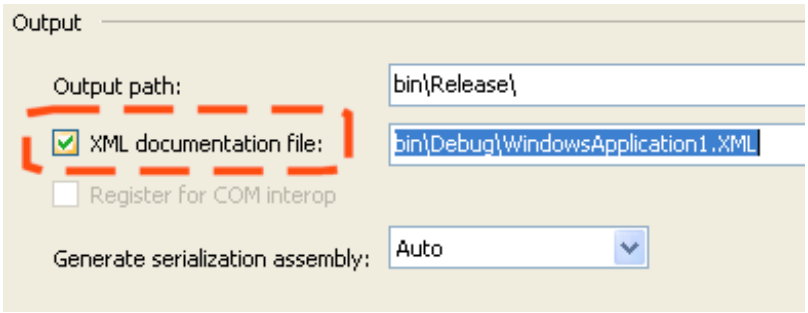

Figura 2.2. Activamos la opción de generación del archivo XML con la documentación.

Marcada la opción, bastará con volver a compilar el proyecto para obtener el citado archivo XML con la documentación extraída de los comentarios.

2.2.3. Tipos de datos y operadores

Para programar en un cierto lenguaje es fundamental conocer los tipos de datos con que cuenta, ya que éstos son necesarios a la hora de declarar una variable, especificar el tipo de información que puede alojar una propiedad, el tipo de los parámetros de entrada o salida de una función, etc.

En C# los tipos de datos pueden clasificarse en dos grandes categorías: tipos por valor y tipos por referencia. Los primeros son más simples y se caracterizan porque la variable o propiedad contiene directamente el valor en cuestión, mientras que los segundos se dan cuando el contenido es una referencia, un puntero si queremos llamarlo así, a la localización en memoria donde se encuentra la información. Los tipos

por valor son sencillos: números, caracteres, valores de una enumeración; mientras que los tipos por referencia pueden tener cualquier estructura.

Los tipos numéricos, con signo y sin signo, son `byte`, `short`, `int`, `long`, `sbyte`, `ushort`, `uint` y `ulong`, con un tamaño de 8, 16 y 32 bits como es habitual. Para operar con números que necesitan una precisión absoluta tenemos el tipo `decimal`. A estos se añaden los tipos en punto flotante: `float` y `double`. El conjunto de tipos fundamentales se completa con los tipos `char` y `string`, para almacenar caracteres y cadenas de caracteres, y el tipo `bool`, que solamente puede contener los valores `true` y `false`.

En cuanto a operadores se refiere, C# dispone del conjunto habitual en la mayoría de los lenguajes de la familia C++. En la tabla 2.1 se enumeran agrupados en las tres categorías clásicas: aritméticos, relacionales y lógicos.

Operadores de C#	
Operadores aritméticos	
+	Suma
-	Diferencia
*	Producto
/	División
%	Resto de división entera
Operadores relacionales	
==	Igualdad
!=	Desigualdad
<	Menor que
<=	Menor o igual que
>	Mayor que
>=	Mayor o igual que
Operadores lógicos	
&&	Y/AND
ll	O/OR
!	Negación/NOT

Tabla 2.1. Operadores del lenguaje C#.

También existe en C# el operador ternario o condicional `?:`, así como el operador `is` que permite comprobar si un cierto objeto es de un tipo concreto.

2.2.4. Estructuras de control básicas

Además de los tipos de datos y los operadores, que nos permiten crear expresiones de distintos tipos, también necesitaremos contar con estructuras de control que, por ejemplo, permitan ejecutar condicionalmente una o más sentencias o repetir esa ejecución.

La instrucción por excelencia para la codificación de condicionales es `if/else`, que podemos usar en C# para escribir bloques como el siguiente:

```
if (expresión) {
  sentencias a ejecutar;
  si la expresión es cierta;
} else {
  sentencias a ejecutar;
  si la expresión es falsa;
}
```

Las llaves son necesarias únicamente en caso de que haya más de una sentencia en el bloque asociado al `if` o el `else`. Estas sentencias pueden, a su vez, ser otros condicionales.

Si va a utilizarse un condicional múltiple para comparar un mismo dato respecto a múltiples valores de referencia, en lugar de usar varios `if/else` anidados podemos recurrir a la instrucción `switch`, que tiene la siguiente sintaxis:

```
switch(dato) {
  case valor1:
    instrucciones;
  case valor2:
    instrucciones;
  default:
    instrucciones;
}
```

Para ejecutar repetitivamente una o varias sentencias podemos recurrir a distintos tipos de bucles, representados por las instrucciones `while`, `do`, `for`, y `foreach`. El bucle condicional típico se implementa con `while`, si precisamos la comprobación antes de ejecutar las sentencias del bucle, o `do/while`, en caso de que la comprobación deba efectuarse al final. Por ejemplo:

```
while(condición) {
  sentencias;
}

do {
  sentencias;
} while(condición);
```

En el primer caso las sentencias podrían no llegar a ejecutarse nunca, en caso de que la condición sea inicialmente falsa, mientras que en el segundo siempre se ejecutarían al menos una vez.

La instrucción `for` es la adecuada cuando aparte de una condición de control es necesario llevar a cabo una inicialización, antes de comenzar el bucle, y una actualiza-

ción en cada ciclo. El caso más típico es el de un bucle por contador como el siguiente:

```
for(int n = 1; n <= 10; n++) {
  sentencias;
}
```

Las sentencias introducidas entre llaves se ejecutarían diez veces en total, tomando la variable `n`, de tipo `int`, los valores desde el 1 al 10.

Finalmente, en cuanto a bucles se refiere, tenemos la instrucción `foreach`. Ésta se dirige especialmente a los casos en que se precisa recorrer todos los elementos de una matriz o una colección, por ejemplo:

```
foreach(string nombre in nombres) {
  sentencias;
}
```

Asumiendo que `nombres` es una matriz de cadenas de caracteres, el bucle se ejecutaría tantas veces como elementos tuviese dicha matriz y la variable `nombre` tomaría a cada ciclo el contenido de un elemento.

2.3. Definición de nuevos tipos

Escribir un programa no se reduce generalmente a definir variables, componer algunas expresiones e introducirlas en estructuras de control como las que acaban de describirse en el punto previo. En la mayoría de los casos es preciso definir tipos de datos propios. De hecho, esto es una necesidad en C# puesto que todo el código de un programa debe residir en el interior de una clase, a pesar de que en ocasiones no lo notemos porque los asistentes se encarguen de la mayor parte del trabajo.

Los tipos de datos que podemos definir en C# son los siguientes:

- **Enumeraciones**: Una enumeración es un tipo de datos simple y por valor que permite, sencillamente, asignar nombres a una secuencia de valores concretos.

- **Estructuras**: También son tipos de datos por valor, pero no simples como las enumeraciones puesto que una estructura se forma con un conjunto de campos, cada uno con un nombre y un tipo, y opcionalmente una serie de métodos para operar sobre ellos.

- **Clases**: La clase es el tipo por excelencia en un lenguaje OOP, permitiendo definir el molde a partir del cual se crearán los objetos. Es un tipo de dato por referencia.

- **Interfaces**: Mediante las interfaces se facilita la definición de una serie de métodos, en realidad solamente su nombre y lista de parámetros, que posteriormente serán implementados por una clase o una estructura. Se trata de un tipo por referencia y facilita el uso de objetos cuya clase se desconoce en

el momento de escribir el código, pero sí se sabe que implementan una cierta interfaz.

- **Delegados**: Un delegado no es más que un puntero a un método de un objeto, pero sin los peligros que en lenguajes como C++ implica el uso de ese tipo de información. Un delegado tiene la misma funcionalidad pero también cuenta con una fuerte comprobación de tipos, por lo que representa un medio seguro de mantener referencias a métodos.

Una vez definidos los tipos, se procedería a crear variables a partir de ellos como si se tratase de cualquier tipo intrínseco. Pueden declararse variables simples, por ejemplo para crear un objeto o una estructura, o matrices, para almacenar múltiples objetos usando un mismo identificador.

2.3.1. Enumeraciones y estructuras

Comenzaremos abordando los tipos más simples: enumeraciones y estructuras. Siempre que tengamos una serie de valores relacionados y que representen elementos a los que puede darse nombre, podemos definir una enumeración que facilite la escritura de código. A la hora de representar los días de la semana, por ejemplo, en lugar de usar los valores 1 a 7 podríamos emplear directamente los nombres de los días si, con anterioridad, definiésemos una enumeración así:

```
enum Dia : byte {
 Lunes=1, Martes, Miercoles, Jueves, Viernes, Sabado, Domingo
};

// Una variable llamada cita con el valor Miercoles
Dia cita = Miercoles;
```

Cada uno de los identificadores establecidos en la definición de la enumeración, en este caso los nombres de los días, podrá ser utilizado en cualquier tipo de expresión que permita el tipo, incluyendo la asignación, comparación con otros valores, conversión, etc.

Una estructura es un conjunto de datos, cada uno con su tipo e identificador, que guardan una cierta relación entre sí. Las estructuras, además, pueden contar también con métodos, generalmente especializados en la manipulación de datos internos. Suponiendo que quisiésemos contar con un tipo de datos que nos permitiese conservar toda la información de cada asignatura, podríamos definir una estructura como la siguiente:

```
struct Asignatura {
  public string Nombre, Aula, Profesor;
  public int Creditos;
  public Dia DiaClase;
};
```

`Asignatura` es el nombre que le damos al nuevo tipo, en cuyo interior hay tres campos de tipo `string`, uno de tipo `int` y otro de tipo `Dia`. El modificador `public` indica que todos ellos son accesibles desde el exterior de la propia estructura. Definido el tipo, para declarar una variable e introducir datos en ella procederíamos de la siguiente forma:

```
Asignatura UnaAsignatura = new Asignatura(
 "Fundamentos físicos de la informática", "E-5", "Luis Péres",
 9, Jueves);

if (UnaAsignatura.Creditos > 6)
  ...
```

Con el operador `new` creamos una nueva variable de tipo `Asignatura` introduciendo en ella los valores facilitados entre paréntesis. Dichos valores deben aparecer en el mismo orden en que están definidos los campos en la estructura. La variable recién creada se asigna a `UnaAsignatura`, identificador que podemos emplear en condicionales, asignaciones, etc.

2.3.2. Clases

Como se indicaba anteriormente, las clases son el tipo de dato por excelencia en un lenguaje de programación orientado a objetos, actuando como un molde a partir del cual se crean los objetos. En ese molde se definen los datos que precisará el objeto para efectuar su trabajo, se establece la funcionalidad mediante sentencias ejecutables en los métodos, se exponen elementos de interacción con los objetos como las propiedades y los eventos, etc.

La definición de una clase se asemeja a la de una estructura, si bien puede contener elementos no permitidos en éstas y aporta características propias de la OOP como son la herencia, derivando una nueva clase de otra ya existente, y el polimorfismo, a través de un ascendiente común o bien por la implementación de una interfaz común. La sintaxis general para definir una clase es la siguiente:

```
[atributos]
modificadores class Nombre : ClaseBase, Interfaz {
  definición de miembros;
}
```

Solamente puede especificarse una clase base, es decir, en C# no existe la herencia múltiple como en C++. Sí es posible, por el contrario, hacer referencia a más de una interfaz para implementar los métodos indicados en ellas, una posibilidad más simple que la herencia múltiple y con aplicaciones similares.

Además de campos de datos y métodos, una clase puede contar con uno o varios constructores y un destructor, así como con propiedades y eventos. Los constructores se distinguen por ser métodos sin tipo de retorno y con el mismo nombre que la clase, mientras que los destructores usan ese mismo nombre precedido por el carácter ~.

Los modificadores, aplicables tanto a la clase como a sus miembros, establecen la visibilidad de acceso y son cuatro: `public`, `protected`, `internal` y `private`, según el cual se permitiría el acceso público, únicamente a las clases derivadas, solamente a las clases derivadas y aquellas que se encuentran en el mismo ensamblado y solamente a los miembros de la propia clase, respectivamente.

En C# las clases pueden contener definiciones de otros tipos de datos, tales como estructuras, enumeraciones, delegados e, incluso, otras clases. La definición de clases anidadas suele utilizarse cuando es necesario crear clases de uso interno a otras.

2.3.3. Espacios de nombres

Un proyecto complejo puede contar con un importante número de definiciones de clases, interfaces y otros tipos de datos, cada uno de ellos con sus respectivos identificadores. Hay que tener en cuenta que la biblioteca de servicios de la plataforma .NET, accesible directamente desde C#, aporta cientos de clases, a las que se suman las nuestras y las de terceras partes que pudiéramos usar. Esto supone que la coincidencia en el uso de identificadores para un determinado tipo no es algo infrecuente, pudiendo dar lugar a conflictos.

En C# este problema tiene una sencilla solución: el uso de los espacios de nombres. Un espacio de nombres, del inglés *namespace*, no es más que un contenedor al que se asigna un nombre, existiendo la posibilidad de anidar unos espacios de nombres dentro de otros. La finalidad de un espacio de nombres es albergar todo tipo de definiciones, ofreciendo un acceso cualificado en caso de que fuese necesario. Por ejemplo:

```
namespace ContenedorA {
  public class MiClase {
   // definición de la clase
  }
}

namespace ContenedorB {
  public class MiClase {
   // definición de la clase
  }
}
```

En este caso tenemos dos clases que se llaman igual, a pesar de que su funcionalidad seguramente es distinta, alojadas cada una en un espacio de nombres diferente. Esto permite usar una u otra mediante referencias completas del tipo `ContenedorA.MiClase` o `ContenedorB.MiClase`.

Todos los servicios ofrecidos por la plataforma .NET se encuentran distribuidos en una serie de espacios de nombres con dependencias jerárquicas, partiendo desde `System`, que sería el espacio raíz, con espacios anidados como `System.Data`, `System.Windows`, `System.Xml`, etc. Para evitar tener que escribir las referencias completas siempre, incluso cuando no existen conflictos de nomenclatura, al inicio del módulo de código pueden agregarse las sentencias `using` que fuesen precisas, como se hace a continuación:

```
using System.Windows.Forms;
```

Esto nos permitiría hacer referencia a la clase `Button`, por ejemplo para crear un botón, escribiendo sencillamente `Button`, en lugar de la referencia completa `System.Windows.Forms.Button` que indica que dicha clase se encuentra en el espacio de nombres `Forms`, que a su vez está en el espacio `Windows` que se encuentra en `System`.

2.4. Novedades de C# 2.0

C# es un lenguaje relativamente nuevo y, por tanto, en evolución casi constante. La versión del compilador que incorpora *Visual C# 2005 Express Edition* es la correspondiente a la especificación de C# 2.0, una versión que añade una serie de importantes novedades y mejoras. Sin ánimo de ser exhaustivos, ni entrar en excesivos detalles que no nos serán de demasiada utilidad hasta que nos familiaricemos suficientemente con el lenguaje, estas novedades son:

- **Definición parcial de tipos**: Ahora es posible dividir la definición de un tipo, por ejemplo una clase, entre varios archivos físicos de código fuente. Para ello se precede la definición con la palabra `partial`. Es un recurso utilizado internamente por los diseñadores para separar el código que generan ellos del escrito por nosotros, si bien puede tener otras aplicaciones como la división entre varios equipos de trabajo del código de una clase.

- **Tipos genéricos**: Ciertas estructuras de datos, como las listas, pilas y diccionarios, son útiles indistintamente de cuál sea el tipo de cada uno de los elementos que se almacenen en ellas. Hasta el momento era necesario definir dichas estructuras para cada tipo concreto, si se quería contar con una comprobación fuerte de tipos durante la compilación y un mejor rendimiento, o bien recurrir al tipo `object` que puede contener cualquier valor o referencia pero sin comprobación estricta y un peor rendimiento en ejecución. Con los tipos genéricos se obtiene lo mejor de ambos mundos, ya que solamente hay que definir las estructuras una vez, pero de tal forma que al usarlas se indica el tipo concreto de los elementos y se obtiene una comprobación estricta de tipos e igual rendimiento.

*Aunque con notables diferencias de implementación interna, funcionalmente hablando los tipos genéricos de C# serían similares a las plantillas (*templates*) de C++.*

- **Métodos anónimos**: Gracias a los métodos anónimos es posible escribir bloques de código asociados directamente a eventos, sin necesidad de codificarlos como métodos independientes conformes a la firma de un cierto delegado. Es un recurso que, principalmente, sirve para simplificar la implementación de la funcionalidad asociada a los eventos de los componentes.

- **Tipos enumerables**: Hasta ahora, para que la información contenida en un tipo propio pudiese ser enumerada mediante la instrucción `foreach` era preciso implementar la interfaz `IEnumerable`, encargada de devolver una referencia a un objeto `IEnumerator` que, a través de sus propiedades y

métodos, facilitaría el número de elementos existentes y su recuperación. Todo este procedimiento se ha simplificado en C# 2.0 gracias a la adición de la palabra clave `yield`, cuya finalidad es devolver directamente los valores de cada elemento.

A éstas, que podemos considerar las más importantes, se añaden otras novedades como los tipos de datos que pueden ser nulos, las clases estáticas o el operador **`::`** de resolución de ámbito.

Resumen

Al finalizar la lectura de este capítulo ya tenemos unas nociones generales sobre las posibilidades y sintaxis del lenguaje C#, un lenguaje de programación moderno, orientado a objetos, orientado a componentes y seguro cuando se le compara con otros como C++.

En principio, para desarrollar las primeras aplicaciones de ejemplo de los capítulos siguientes, no necesitaremos saber mucho más sobre C#, ya que los espacios de nombres, clases y otros tipos serán definidos por los asistentes, limitándonos nosotros a escribir las sentencias en el interior de los métodos. Para esto sí tenemos que conocer aspectos como las estructuras de control básicas (bucles y condicionales), los operadores y tipos de datos.

También deberemos conocer las clases que ofrece la propia plataforma .NET para acceder a los distintos recursos: consola del sistema, controles en un formulario Windows, etc. Éstos serán algunos de los temas tratados en los capítulos siguientes.

3. Cómo crear aplicaciones de consola

Las aplicaciones de consola reciben este nombre porque se ejecutan en la consola del sistema, la línea de comandos de texto, y no cuentan con una interfaz gráfica basada en ventanas. El uso de la consola de texto resulta adecuado cuando la interacción con el usuario es lo suficientemente sencilla como para no precisar más que la introducción de datos por teclado, sin recurrir a elementos como las listas, botones de radio y otros controles de Windows.

En cuanto a su estructura, las aplicaciones de consola representan uno de los tipos de proyecto más sencillos que podemos crear. Una aplicación de este tipo consta generalmente de una única clase, conteniendo el método `Main` y cualquier otro adicional que pudiéramos necesitar. En el código se hará uso intensivo de la clase `System.Console` a fin de acceder a la consola, enviando y recogiendo información. Los métodos de esta clase son estáticos, lo que significa que no es necesario crear un objeto `Console` para poder invocarlos sino que podemos usar la notación `Console.Método(parámetros)`.

3.1. Uso del asistente para aplicaciones de consola

Aunque podríamos partir de un proyecto completamente vacío, añadir un módulo de código, las referencias necesarias al espacio de nombres `System` y escribir la aplicación de consola partiendo de cero, lo cierto es que *Visual C# 2005 Express Edition* cuenta con un asistente que se encarga de dar esos pasos, creando para nosotros una plantilla típica de aplicación de consola.

En la ventana **Nuevo proyecto** encontramos un elemento (véase la figura 3.1 en la página siguiente) que es el encargado de generar una nueva aplicación de consola. Opcionalmente podemos introducir un nombre para dicha aplicación en la parte inferior, si no queremos usar el propuesto por defecto. Un clic en el botón **Aceptar** y el asistente creará el proyecto, añadiendo un módulo con el código siguiente:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace AppConsola {
    class Program {
        static void Main(string[] args)
        {
        }
    }
}
```

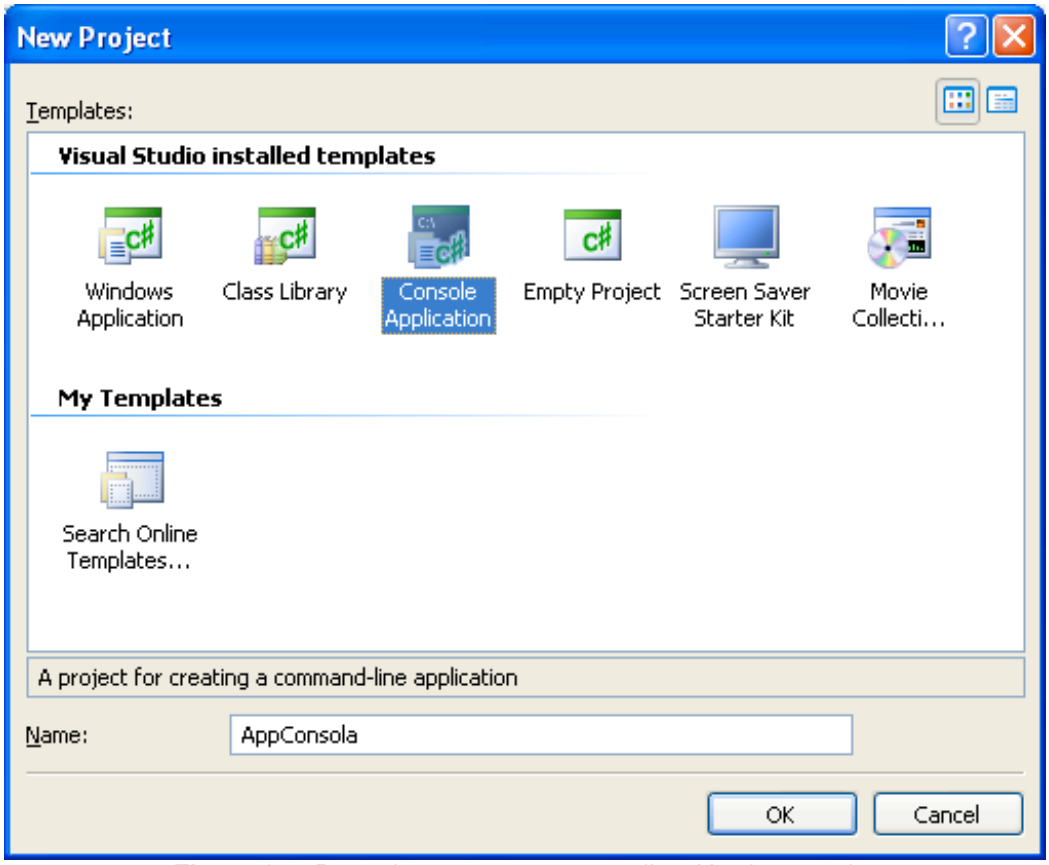

Figura 3.1. Procedemos a crear una aplicación de consola.

Las tres primeras líneas hacen referencia a otros tantos espacios de nombres en los que se encuentran las clases más usadas en este tipo de aplicaciones. A continuación se encuentra la definición de un espacio de nombres propio, cuyo nombre es el introducido en la ventana **Nuevo proyecto**, en cuyo interior hay una clase con el método `Main`. Puede modificar libremente el nombre de la clase, pero no así el del método `Main`, sus parámetros o tipo de retorno.

3.1.1. Envío de información a la consola

El código generado por el asistente conforma una aplicación de consola completa, pero no cuenta con instrucción ejecutable alguna que envíe o solicite información de la consola. Si la ejecutamos veremos aparecer la consola del sistema durante un momento, a continuación desaparecerá y la ejecución llegará a su fin.

Para enviar cualquier información a la consola, un mensaje, el contenido de una variable, el resultado de una expresión, etc., recurriremos a los métodos `Write` y `WriteLine` de la clase `Console`. La única diferencia que hay entre ambos es que `WriteLine` introduce un salto de línea al final, mientras que `Write` no lo hace. En realidad ambos métodos cuentan con casi una veintena de versiones diferentes, cada una de las cuales toma una lista de argumentos distinta. Esto nos permite enviar a la consola un carácter, un número entero o de punto flotante, un valor `bool`, etc.

En caso de que necesitemos enviar a la consola más de un dato, por ejemplo el contenido de varias variables, tenemos dos opciones: usar repetidamente el método

`Write/WriteLine`, una vez por cada dato, o bien usar la versión de dicho método que toma como primer parámetro una cadena de formato y, a continuación, tantos datos como sean necesarios. Esa cadena de formato indicará el lugar en el que debe introducirse cada dato, haciendo referencia a ellos mediante la notación `{n}`, donde `n` es un número comprendido entre 0 y el número de datos menos 1. Por ejemplo:

```
int Dias = 31;
string Mes = "Enero";

Console.Write("El mes de {0} tiene {1} días", Mes, Dias);
```

Si tenemos que enviar a la consola la información contenida en algún objeto, en lugar de una variable simple como las citadas antes, siempre podemos intentar recurrir a su método `ToString`. Éste se encuentra definido en `object` y, por tanto, está presente en todos los tipos de datos.

3.1.2. Recogida de información desde la consola

Cuando lo que se necesita es recoger información desde la consola, un dato introducido por el usuario del programa a través del teclado, usaremos los métodos `Read` y `ReadLine`. El primero de ellos lee un carácter de la consola, mientras que el segundo recoge una secuencia de caracteres hasta encontrar un retorno de carro. No son necesarios parámetros en ninguno de los casos, obteniéndose en el primero de ellos un valor de tipo `int` y en el segundo de tipo `string`.

A menos que necesitemos un control carácter a carácter de la información introducida a través del teclado, en cuyo caso emplearíamos `Read`, lo habitual es que usemos el método `ReadLine` de `Console` para obtener líneas completas. Por ejemplo:

```
Console.Write("Cuál es tu nombre: ");
string Nombre = Console.ReadLine();
Console.Write("Hola {0}", Nombre);
```

En caso de que el dato que esperamos por la consola no sea una cadena de caracteres, sino un número o algún otro tipo de dato, tendremos que recurrir al método `Parse` del tipo en cuestión. Éste toma como argumento la cadena de caracteres y extrae el dato en el tipo deseado. Si el formato de la cadena no es el adecuado se producirá una excepción. El siguiente fragmento de código solicita la edad mediante el método `ReadLine`, usando el método `Parse` de `int` para obtenerlo como dato numérico y, así, poder efectuar una comparación en la sentencia `if`:

```
int Edad = int.Parse(Console.ReadLine());
if (Edad >= 18)
  Console.WriteLine("Ya eres mayor de edad");
else
  Console.WriteLine("Aún eres menor de edad");
```

Podríamos haber optado por guardar el valor devuelto por `ReadLine` en una variable de tipo `string` y, a continuación, efectuar la conversión con `Parse` alojando el resultado en una variable `int`. En este caso, sin embargo, se ha entregado directamente a `Parse` la información recuperada por `ReadLine`, lo cual nos ahorra un paso y algo de código.

3.2. Uso de la consola rápida

Al ejecutar cualquiera de los ejemplos previos, introduciendo las sentencias de código en el interior del método `Main` con que ya cuenta el proyecto, comprobará que aparece la típica ventana de consola del sistema pero que desaparece con rapidez, quedando el resultado en una ventana del propio entorno de *Visual C# 2005 Express Edition* llamada **Consola rápida**. Si en el código solicitamos alguna información, mediante los métodos `Read` o `ReadLine`, ésta debemos introducirla a través de dicha ventana y no mediante la consola del sistema.

La finalidad de la **Consola rápida** es hacer más cómoda la comprobación de las aplicaciones de consola, programas que, cuando sean finalmente distribuidos a su ubicación definitiva, serán ejecutados desde la línea de comandos. Puede activar y desactivar el uso de la **Consola rápida** por parte del entorno mediante la opción **Redirigir resultados de la consola a la ventana Consola rápida** (véase la figura 3.2) que se encuentra en la página **Depuración>General** de la ventana de opciones. Ésta se abre mediante la opción **Herramientas>Opciones** del menú principal.

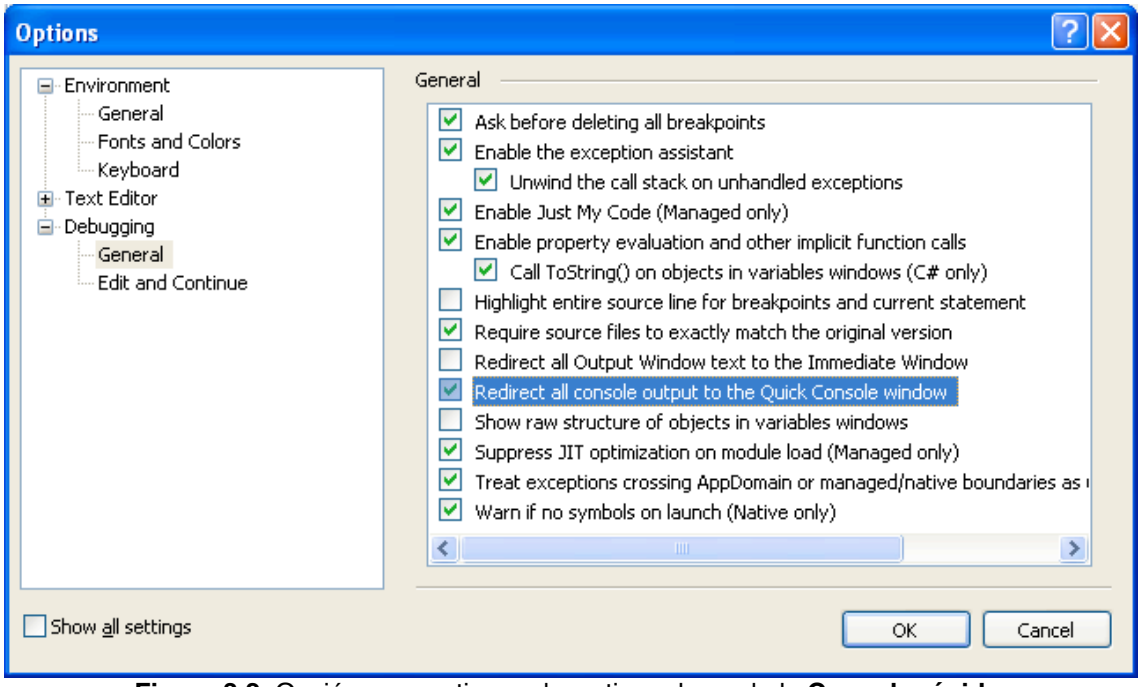

Figura 3.2. Opción para activar y desactivar el uso de la **Consola rápida**.

Si hemos cerrado la ventana de **Consola rápida** tras la ejecución de un programa, al volver a ejecutarlo debe aparecer automáticamente. No obstante, siempre podemos recurrir a la opción **Depurar>Ventanas>Consola rápida** para hacerla aparecer.

*Debemos tener en cuenta que la **Consola rápida** no contempla el uso de ciertos métodos avanzados de la clase `Console`, como los que permiten controlar la posición del cursor y los colores, generándose una excepción si intentamos ejecutar un programa que use dichas características teniendo la **Consola rápida** activada. Para*

solucionar este problema no tenemos más que usar la opción antes indicada para desactivarla, utilizando en su lugar la consola tradicional de la línea de comandos.

3.3. Características avanzadas de la clase `Console`

La clase `Console` ha sido mejorada en esta nueva versión de la plataforma .NET, agregando características avanzadas que permiten controlar aspectos como la posición del cursor, los colores de fondo y tinta, las dimensiones de la ventana y el buffer donde se almacena la información que en ella aparece, etc. Estas posibilidades no resultan imprescindibles para crear una aplicación de consola, pero permiten mejorar tanto el aspecto mostrado en la ventana como la ínteractuación del programa con el usuario.

Ciertas operaciones pueden llevarse a cabo tanto modificando el valor de una o varias propiedades como a través de la invocación de un método. Para establecer la posición del cursor, por ejemplo, es posible asignar la nueva columna a la propiedad `CursorLeft` y la nueva fila a `CursorTop` o, como alternativa, llamar al método `SetCursorPosition` facilitando la columna y la fila. Es decisión nuestra emplear una vía u otra según nos convenga.

3.3.1. Controlar la posición y colores del texto

Los métodos que hemos conocido hasta ahora, como `WriteLine` y `ReadLine`, operan siempre a partir de la posición actual del cursor en la consola, limitándose a introducir saltos de línea al final y, si es preciso, desplazar el contenido actual de la ventana para dar cabida a las nuevas líneas. No tenemos, por tanto, un gran control sobre la posición en que se muestran o solicitan los datos, como tampoco lo tenemos sobre los colores del texto ya que siempre se emplean los establecidos por defecto en la configuración de la consola.

Antes de enviar o solicitar cualquier información a la consola, con los métodos antes citados, podemos colocar el cursor donde prefiramos recurriendo a las propiedades `CursorLeft` y `CursorTop` o bien el método `SetCursorPosition`. Las propiedades también pueden leerse, facilitando así la recuperación de la posición actual del cursor. También podemos ocultar y mostrar el cursor cuando sea necesario, mediante la propiedad `CursorVisible`, así como cambiar su tamaño relativo a través de la propiedad `CursorSize`.

Teniendo el cursor en la posición que nos interese, recurriremos a las propiedades `BackgroundColor` y `ForegroundColor` para establecer el color de fondo y texto, respectivamente. Estas propiedades pueden tomar cualquiera de los valores de la enumeración `ConsoleColor`, valores que aparecerán en forma de lista en el propio editor de código cuando vayamos a hacer una asignación.

Las sentencias mostradas a continuación generan el resultado que aparece en la figura 3.3, con una serie de campos colocados de forma tabulada. Solamente se solicita el primero de ellos y no lo utilizamos para nada, es una simple demostración de

cómo usar el color y la posición del cursor. El método `Clear` se encarga de borrar todo el contenido de la consola y establecer el color de fondo asignado a `BackgroundColor`, mientras que la propiedad `Title` asigna el título que mostrará la ventana.

```
Console.Title = "Datos de registro";
Console.BackgroundColor = ConsoleColor.White;
Console.Clear();
Console.ForegroundColor = ConsoleColor.Blue;
Console.SetCursorPosition(5, 2);
Console.Write("Nombre    :");
Console.SetCursorPosition(5, 4);
Console.Write("Apellidos :");
Console.SetCursorPosition(5, 6);
Console.Write("Dirección :");
Console.ForegroundColor = ConsoleColor.Black;
Console.SetCursorPosition(20, 2);
Console.ReadLine();
```

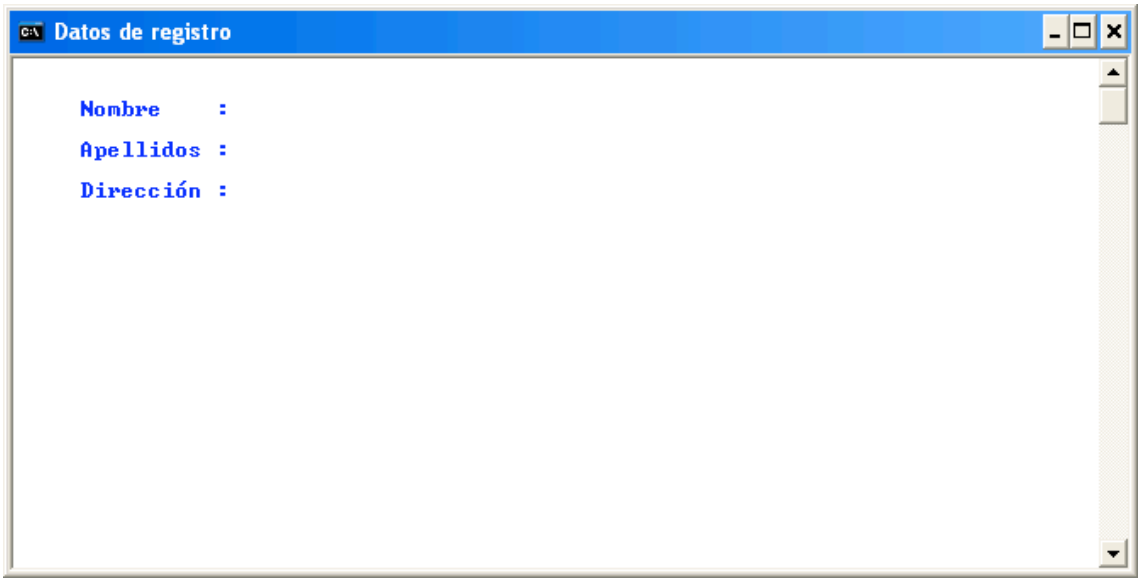

Figura 3.3. Una interfaz de texto con uso de color y posición del cursor.

Para ejecutar este programa no hemos de olvidar que es necesario desactivar la **Consola rápida**, ya que de lo contrario obtendremos un mensaje de error en lugar del resultado que esperamos.

3.3.2. Dimensiones de la ventana

La consola del sistema asigna unas dimensiones predeterminadas a la ventana en que se ejecuta, dimensiones que coinciden con el estándar de los tiempos de DOS: 80 columnas por 25 filas. En realidad es posible enviar más de 25 líneas de información a la consola, si bien en la ventana solamente se verán parte de ellas mientras que el resto se almacenan en un buffer interno. Éste es el que permite que usemos la barra de desplazamiento arriba y abajo para poder desplazar la ventana sobre el contenido total, a fin de ver la parte que interese en cada momento.

El tamaño de la ventana puede cambiarse mediante las propiedades `WindowWidth` (ancho) y `WindowHeight` (alto), si bien las nuevas dimensiones nunca deben superar el ancho y alto máximos indicados en `LargestWindowWidth` y `LargestWindowHeight`.

También es posible controlar el tamaño del buffer en el que se almacena la información, con las propiedades `BufferWidth` y `BufferHeight`, si bien éste es un aspecto que normalmente se deja en manos del sistema que lo adecua según el tamaño de la propia ventana.

Agregando las dos sentencias siguientes al inicio del ejemplo anterior, en el interior del método `Main` de una aplicación de consola, comprobaremos cómo la ventana se ajusta a las máximas dimensiones posibles según la resolución de nuestra pantalla.

```
Console.WindowHeight = Console.LargestWindowHeight;
Console.WindowWidth = Console.LargestWindowWidth;
```

3.3.3. Detección de pulsaciones de teclado

El inconveniente de los métodos `Read` y `ReadLine` de la clase `Console`, únicos que conocemos por el momento para recoger información procedente del teclado, es que no efectúan realmente su trabajo hasta que el usuario pulsa **Intro** y éste, potencialmente, puede usar el teclado para desplazar el cursor, introducir cualquier carácter e incluso interrumpir la ejecución del programa. Es decir, el usuario tiene todo el control sobre la aplicación.

Si no queremos que se dé esta situación podemos usar el método `ReadKey`, nuevo en esta versión de la clase `Console`, conjuntamente con la propiedad `KeyAvailable` que nos permite saber si hay o no pulsaciones de tecla pendientes de procesar. Para impedir que la combinación **Control-C** interrumpa la ejecución del programa no tenemos más que dar el valor `true` a la propiedad `TreatControlCAsInput`.

El método `ReadKey` no devuelve directamente un entero correspondiente al carácter pulsado, sino que facilita una estructura `ConsoleKeyInfo` en la que encontramos tres miembros: `Key`, `KeyChar` y `Modifiers`. El primero contendrá uno de los valores de la enumeración `ConsoleKey` indicando la tecla pulsada, incluyendo las que no generan caracteres como las teclas de edición y función. Si la tecla genera un carácter podemos recogerlo de `KeyChar`. Por último, `Modifiers` nos permite saber si estaba pulsada alguna de las teclas muertas (**Mayús**, **Control** y **Alt**).

Por defecto el método `ReadKey` de `Console` mostrará en la consola el carácter correspondiente a la pulsación recogida. Si no queremos que ocurra esto debemos facilitar como parámetro el valor `true`.

Obviamente no todo son ventajas, ya que usar este método para recoger las pulsaciones del teclado requerirá un mayor trabajo por nuestra parte. Suponiendo que quisiéramos recoger el nombre del usuario del programa, partiendo del fragmento de código previo, tendríamos que recurrir a un bucle en el que fuésemos leyendo carácter

a carácter, comprobando si es un carácter válido y guardándolo. Es lo que hace el código siguiente:

```
string Nombre = "";
ConsoleKeyInfo Pulsacion;
do
{
    Pulsacion = Console.ReadKey(true);
    if (char.IsLetter(Pulsacion.KeyChar))
    {
        Nombre += Pulsacion.KeyChar;
        Console.Write(Pulsacion.KeyChar);
    }
} while (Pulsacion.Key != ConsoleKey.Enter);

Console.SetCursorPosition(5, 10);
Console.WriteLine("El nombre introducido ha sido {0}", Nombre);
```

Este código podría introducirse en un método propio, que aceptase como argumento la lista de caracteres permitidos o prohibidos, la longitud máxima, etc., y devolviese como resultado el dato recogido. Así se evitaría su repetición cada vez que hubiese que facilitar alguna información.

Resumen

En este capítulo hemos aprendido a crear con *Visual C# 2005 Express Edition* aplicaciones que se ejecutan en una ventana de consola del sistema, aplicaciones cuya interfaz esta basada únicamente en el uso de texto y la interactuación con el teclado, prescindiendo del dispositivo por excelencia en la actualidad que es el ratón. Este tipo de programas son adecuados precisamente cuando la interfaz de usuario no es importante, ya que nos permiten concentrarnos en la codificación directa de un proceso.

También hemos conocido algunas de las características avanzadas de la clase `Console` en la versión 2.0 de la plataforma .NET, con propiedades y métodos que no están presentes en versiones previas.

4. Cómo crear aplicaciones con ventanas

Cuando un programa está pensado para ser empleado directamente por un usuario, y no para ejecutarse como un servicio o aplicación de servidor, lo habitual es que cuente con una interfaz basada en ventanas. Las interfaces gráficas de usuario o GUI (*Graphics User Interface*) se han convertido en un estándar, simplificando la interacción con las personas a través del uso de metáforas como los botones, las listas o los menús, con las que se representan conceptos bien conocidos por todos.

La plataforma .NET cuenta con un sofisticado conjunto de componentes dirigidos a facilitar la construcción de este tipo de aplicaciones, conocidos genéricamente como componentes para formularios Windows, de tal forma que el programador ya tiene a su disposición los elementos antes citados: botones, listas, menús, cuadrículas, etc. Solamente tiene que crear los componentes, configurarlos a través de sus propiedades y métodos y responder a los eventos que generan.

En esta última fase es donde entra en escena el diseñador de formularios integrados en el entorno de *Visual C# 2005 Express Edition*, una herramienta que permite al programador arrastrar los componentes hasta el formulario y editar sus propiedades generando automáticamente el código que, de otra manera, se tendría que haber escrito manualmente. El resultado final es que el diseño de la interfaz de una aplicación basada en formularios Windows se convierte, gracias al diseñador y los componentes prefabricados, en un proceso realmente sencillo y eficiente.

El objetivo de este capítulo, partiendo de que sabemos cómo arrastrar los componentes desde el **Cuadro de herramientas** hasta el formulario y modificar sus propiedades en la ventana **Propiedades** al ser tareas que aprendimos en un capítulo previo, será conocer más detalles acerca del propio formulario y también algunos de los controles de uso más habitual.

4.1. Controles más usuales y sus características

La lista de controles que pueden utilizarse para confeccionar una interfaz de usuario basada en formularios Windows es muy extensa, pero existe un grupo cuyo uso resulta mucho más frecuente. Forman parte de este grupo los botones, cuadros de texto, listas, botones de radio y etiquetas de texto, entre otros. En general, los controles de uso más común los encontraremos en la sección **Controles comunes** del **Cuadro de herramientas** (véase la figura 4.1 en la página siguiente). Son éstos en los que vamos a concentrarnos en un principio.

A la hora de insertar un control en el formulario tenemos varias opciones: hacer doble clic sobre el control en el **Cuadro de herramientas**, hacer clic sobre el control y después en el punto del formulario donde se desea colocar o bien hacer clic sobre el

control y a continuación arrastrar y soltar sobre el formulario a fin de delimitar el área que deseamos asignar como tamaño inicial.

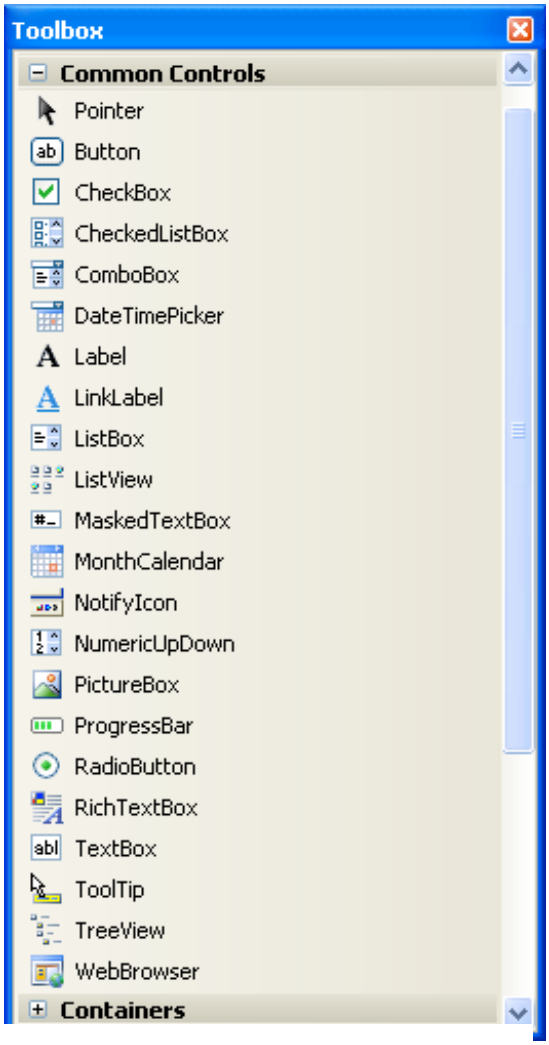

Figura 4.1. Controles de uso común.

Las dos primeras técnicas insertan el control usando las dimensiones por defecto, si bien podemos ajustarlas con posterioridad. De optar por el doble clic en el icono del control, también la posición inicial en el formulario será la usada por defecto por el diseñador, mientras que el segundo procedimiento nos permite ponerlo en el lugar que deseemos. En cualquier caso, una vez insertado el control puede arrastrarse y colocarse donde deseemos.

A medida que arrastremos los controles dentro del formulario, para alterar su posición, el diseñador nos irá mostrando unas líneas guía que facilitan la alineación respecto a los demás que haya alrededor. También al cambiar el tamaño aparecerán dichas líneas, haciendo que sea más fácil conseguir controles con el mismo ancho o alto sin necesidad de tener que editar directamente las propiedades `Width` y `Height` en la ventana **Propiedades**.

Además de los recuadros que aparecen en el contorno del control, una vez insertado, y que sirven para ajustar su tamaño, ciertos controles mostrarán en la parte superior derecha un botón con el icono de una flecha en su interior. Dicho botón da paso al menú de tareas del control, enumerando las opciones de configuración que se utilizan de forma más habitual en cada caso. La lista de tareas de un `TextBox`, por ejemplo, permite activar o desactivar la propiedad `MultiLine` para permitir la introducción de varias líneas de texto, mientras que en el control `ListBox` encontramos opciones para editar los elementos mostrados en la lista o vincularla a un origen de datos.

4.1.1. Propiedades, métodos y eventos comunes

A pesar de las patentes diferencias que existen entre un botón, un recuadro de texto y una lista, lo cierto es que todos ellos son controles Windows y, como tales, tienen un ascendiente común: la clase `Control` definida en `System.Windows.Forms`. Esto explica que compartan un mismo conjunto de miembros, entre ellos una serie de propiedades y eventos comunes, a los que se añaden otros de carácter específico. El conocimiento de los miembros comunes, por tanto, nos será de utilidad general, con independencia de los controles concretos que vayamos a usar.

Todos los controles se colocan en el interior de un contenedor, normalmente un formulario Windows, por lo que necesitan por igual contar con unas coordenadas y unas dimensiones. Heredados de la clase `Control` disponen de los siguientes elementos:

- `Left`, `Top`, `Width` y `Height`: Estas cuatro propiedades contienen las coordenadas de la esquina superior izquierda del control (columna y fila), el ancho y alto, respectivamente. Aunque normalmente se establecen en la fase de diseño mediante operaciones de arrastrar y soltar, podemos editar directamente cualquiera de estas propiedades tanto en la ventana **Propiedades** como a través de una sentencia de asignación en el código del programa.

- `Location`: Es una estructura de tipo `Point` que contiene las coordenadas (`X` e `Y`) donde está colocado el control. Esta estructura dispone de métodos que facilitan el movimiento del control.

- `Bounds`: Esta propiedad también es una estructura, pero en este caso de tipo `Rectangle` y tiene cuatro miembros: `X`, `Y`, `Width` y `Height`, por lo que representa una vía de acceso las coordenadas y dimensiones del control. Al igual que la anterior cuenta con métodos para manipular posición y tamaño.

- `SetLocation` y `SetBounds`: Mediante estos dos métodos puede fijarse la posición o la posición y dimensiones en un solo paso, sin necesidad de realizar varias asignaciones a las propiedades antes citadas. Basta con entregar la lista de parámetros adecuada: coordenadas de la esquina superior izquierda a `SetLocation` y coordenadas y dimensiones a `SetBounds`.

Muchos de los controles de una interfaz basada en ventanas pueden tomar el foco de entrada, estado en el cual las pulsaciones de teclado van dirigidas a ese control y no a los demás que hubiese en la ventana. El foco de entrada puede desplazarse de un control a otro, siendo el más habitual el uso de la tecla **Tab** para ir avanzando según un orden preestablecido. También puede otorgarse de forma directa a un cierto control haciendo clic sobre él con el puntero del ratón. Los miembros relacionados con el foco de entrada y el orden en que lo toman los controles son los siguientes:

- `TabStop`: Es una propiedad de tipo `bool` que indica si un control participa o no de la toma del foco de entrada mediante la tecla **Tab**. Por defecto siempre toma el valor `true` para los controles que pueden tomar el foco. Dándole el valor `false` conseguiremos que nunca se llegue a este control mediante la tecla **Tab**, pero no podremos impedir que tome el foco de entrada a través de un clic de ratón.

- `TabIndex`: Con esta propiedad se establece el orden de acceso a los controles mediante la tecla **Tab**. Contendrá un valor entero a partir de 0, siendo el control que tenga dicho valor el primero en tomar el foco de entrada. En lugar de editar manualmente la propiedad `TabIndex` en la ventana **Propiedades**, puesto que en principio el valor que tomará será el del orden en que se hayan insertado los controles, podemos recurrir a la opción **Ver>Orden de tabulación**. Al activarla, cada control mostrará en un recuadro el orden que tiene asignado, bastando un clic sobre ellos para fijar el nuevo orden que nos interese (véase la figura 4.2).

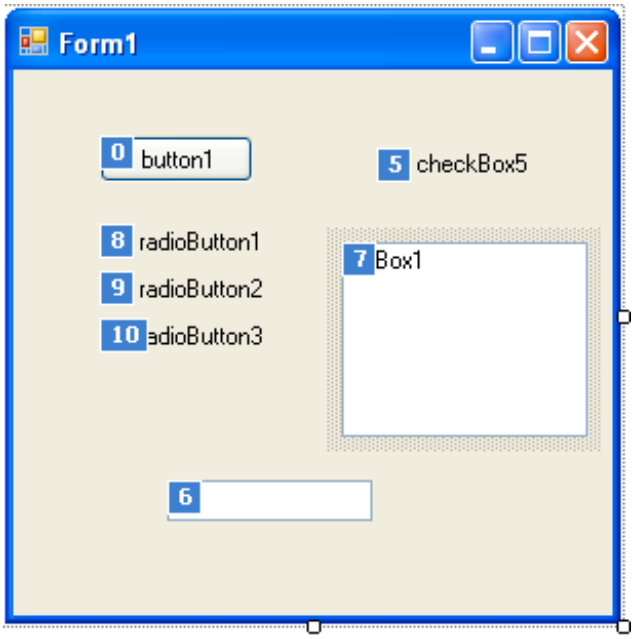

Figura 4.2. Establecemos el orden de acceso a los controles mediante el tabulador.

- `CanFocus` y `Focused`: Son dos propiedades de tipo `bool` y están presentes en todos los controles. La primera indica si el control puede o no tomar el foco de entrada. Un control `Label`, por ejemplo, no puede tomarlo, pero sí un control `Button`. La segunda contendrá el valor `true` si el control en cuestión tiene el foco de entrada en ese momento o el valor `false` en caso contrario.

- `Focus`: Este método permite dar el foco de entrada directamente al control que nos interese, sencillamente introduciendo una sentencia del tipo `Control.Focus();` en el código.

Teniendo el foco de entrada, a medida que se detecten pulsaciones de tecla éstas se transferirán al control mediante una serie de eventos. El proceso será similar para las acciones desencadenadas mediante el ratón, generalmente a través de un clic de botón. La serie de eventos generados en un caso y otro es la siguiente:

- `KeyDown`, `KeyUp` y `KeyPress`: Se producen al pulsar una tecla, liberarla y en caso de que esta tecla tenga asociado un carácter, es decir, no sea una tecla de función o edición. Los dos primeros permiten interceptar cualquier pulsación de tecla, obteniendo información de ella a través del parámetro `KeyEventArgs` que recibirá el método asociado al evento. El último recibe un parámetro de tipo `KeyPressEventArgs` con el carácter pulsado en el miembro `KeyChar`.

- `MouseDown`, `MouseUp` y `MouseMove`: Cuando pulsamos un botón del ratón se genera el primer evento, seguido del segundo en el momento en que lo soltamos. El tercero notifica el desplazamiento del puntero del ratón sobre el control. Los tres eventos aportan los parámetros necesarios para saber cuál es el botón pulsado y dónde está el puntero del ratón en ese instante.

- `Click` y `DblClick`: Estos dos eventos indican que se ha hecho clic o doble clic sobre un control. A pesar de que lo habitual es que se produzcan por una actuación del ratón, lo cierto es que ciertos atajos de teclado y la misma tecla **Intro** sobre un botón se reflejarán también como un evento `Click`.

Además de servirnos de estos eventos, para detectar las pulsaciones de tecla o botones del ratón, también podemos recurrir a las propiedades `ModifierKeys`, `MouseButtons` y `MousePosition`, que todos los controles heredan de la clase `Control`, para conocer el estado de las teclas muertas, los botones del ratón y la posición del puntero de dicho dispositivo.

4.1.2. Recuadros de texto

Una vez que conocemos los aspectos generales aplicables a todos los controles, vamos a ir conociendo los utilizados con mayor frecuencia agrupados según su finalidad. El primer grupo será el formado por los recuadros de texto, áreas delimitadas por un contorno en las que el usuario puede introducir una o más líneas de texto sin limitación inicial alguna. Hay tres controles disponibles de este tipo:

- `TextBox`: Es el más simple y utilizado. Sirve para pedir una o varias líneas de texto sin más control intrínseco que el de poder limitar la longitud, el número máximo de caracteres a introducir.

- `MaskedTextBox`: Similar al anterior, añade las propiedades necesarias para conseguir que la información introducida se ajuste a una cierta máscara o patrón, como puede ser la de un teléfono, NIF, código postal, etc.

- `RichTextBox`: Se utiliza para facilitar la introducción o visualización de texto con formato, usando estilos de párrafo y carácter tales como la alineación, sangrado, tipos y tamaños de letra, bolos, etc.

En los tres casos la propiedad `Text` nos permite tanto recuperar el texto introducido en ejecución como modificarlo, mediante una simple asignación. Los controles `TextBox` y `RichTextBox` cuentan también con la propiedad `Lines`, una matriz de tipo `string` que hace posible el acceso individual a cada una de las líneas de texto en lugar de al contenido completo.

*El control `TextBox` no permitirá la introducción de más de una línea de texto a menos que demos el valor `true` a la propiedad `MultiLine`. Podemos hacerlo sin necesidad de acceder a la ventana **Propiedades**, sencillamente abriendo la lista de tareas del control y marcando la opción que aparece.*

Tanto `TextBox` como `RichTextBox` disponen de una propiedad llamada `MaxLength` con la que puede limitarse el número de caracteres que es posible introducir en el control, teniendo por defecto el valor 32767 en el caso de `TextBox` y 2147483647 en el de `RichTextBox`. En el control `MaskedTextBox` la longitud del dato viene determinada por la máscara asignada a la propiedad `Mask`, ya sea escri-

biéndola de forma manual o bien sirviéndonos del cuadro de diálogo específico para esta tarea (véase la figura 4.3), en el que podemos elegir entre una serie de máscaras predefinidas, escribir la nuestra propia y observar una vista previa en la parte inferior.

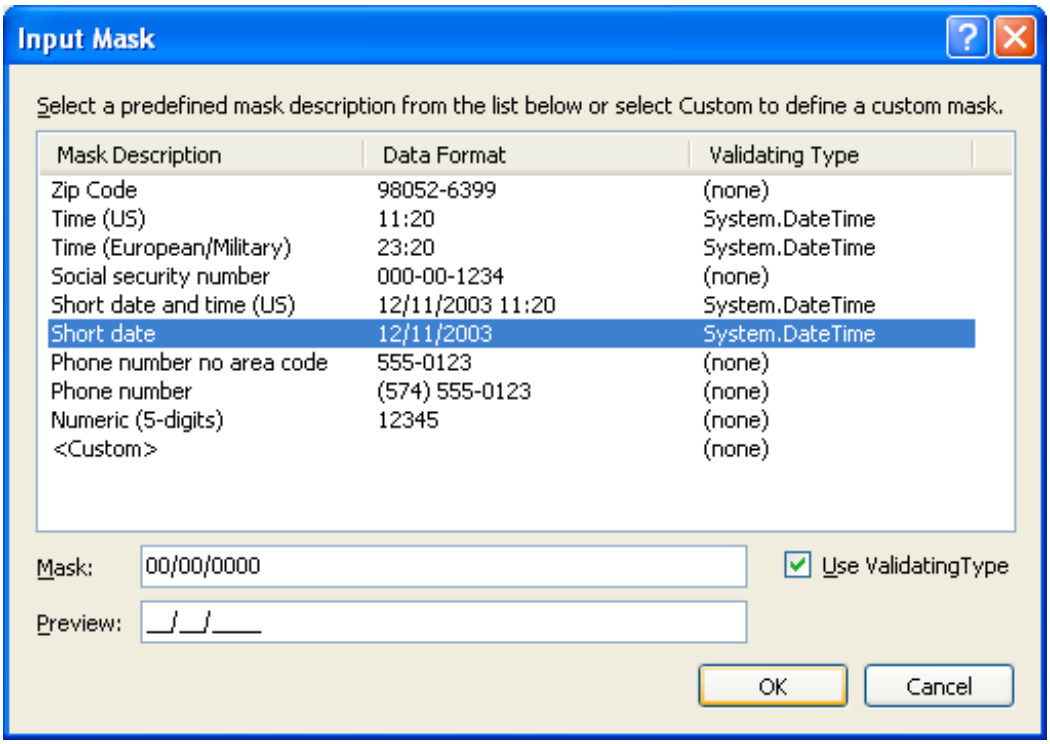

Figura 4.3. Edición de la propiedad `Mask` de un control `MaskedTextBox`.

El control `RichTextBox` no dispone de elemento de interfaz alguno que permita cambiar los atributos de párrafo o carácter, sino que estas tareas se llevan a cabo mediante código.

4.1.3. Botones de radio y de selección

A veces no es necesario que el usuario introduzca un dato escribiéndolo mediante teclado, sino que puede elegirlo entre varias opciones exclusivas entre sí o, sencillamente, marcar o desmarcar una cierta posibilidad. En estos casos se usan dos tipos de botones: `RadioButton` y `CheckBox`. Ambos tienen en común los siguientes miembros fundamentales:

- `Text`: Contendrá el texto descriptivo de la opción y que aparecerá junto al botón.
- `Checked`: Esta propiedad de tipo `bool` determina si el botón está marcado o no.
- `Click`: El evento que se genera al hacer clic sobre uno de estos botones.
- `CheckedChanged`: Evento que notifica un cambio de la propiedad `Checked`.

Funcionalmente, sin embargo, se comportan de manera diferente. Un `CheckBox` cambia de estado cada vez que se hace clic sobre él, invirtiéndose el valor de la propiedad `Checked`, sin afectar para nada al resto de controles del mismo tipo que existan en el contenedor. Es el control adecuado cuando se quiere dar una opción que el usuario puede marcar o desmarcar, del tipo sí/no, verdadero/falso.

Figura 4.4. Dos contenedores `GroupBox` con cuatro `RadioButton` cada uno.

Los controles `RadioButton`, por el contrario, operan de manera conjunta, sirviendo para ofrecer varias opciones exclusivas entre sí, es decir, de las que solamente puede elegirse una. Por ello cuando se marca un botón de este tipo el resto de los existentes en el mismo grupo se desmarcan automáticamente. Para crear varios grupos de estos controles es necesario introducirlos en contenedores separados. En la sección **Contenedores** del **Cuadro de herramientas** encontraremos todos los disponibles, entre ellos `Panel` y `GroupBox` que resultan adecuados para esta tarea.

4.1.4. Listas de distintos tipos

El siguiente grupo de componentes más utilizado es el de las listas, ya sean simples, desplegables o de elementos que pueden marcarse y desmarcarse, los tres tipos representados por los controles `ListBox`, `ComboBox` y `CheckedListBox`, respectivamente.

A pesar de las lógicas diferencias en apariencia y funcionalidad, estos tres componentes comparten una serie de aspectos comunes. El más importante de ellos es que almacenan los elementos que muestran en forma de lista en una propiedad llamada `Items`, una colección de objetos que puede editarse tanto durante la fase de diseño, con el editor de colecciones integrado, como en ejecución. Para abrir el editor basta con hacer clic sobre el botón que aparece junto a la propiedad `Items`, en la ventana **Propiedades**.

Como todas las colecciones, `Items` dispone de una serie de miembros que nos permiten saber cuántos elementos hay en la lista (propiedad `Count`), acceder a cada uno de ellos (propiedad `Item`), añadir uno o varios nuevos elementos (métodos `Add` y `AddRange`), eliminar uno o todos los elementos (métodos `Remove` y `Clear`), etc. Asimismo es posible enumerar el contenido de la lista con la instrucción `foreach`.

La colección a la que apunta la propiedad `Items` de estos controles implementa las interfaces `IList`, `ICollection` e `IEnumerable`, por lo que se comportan según el estándar de las colecciones .NET.

La única diferencia entre un `ListBox` y un `ComboBox` es que este último aparece como un recuadro de texto con una lista desplegable asociada, siendo posible, según el estilo empleado, escribir en ese recuadro de texto en lugar de elegir un valor de la lista. El control `ListBox`, por el contrario, ocupa un área mayor y muestra un cierto número de elementos de la lista, así como unas barras de desplazamiento en caso de que fuesen necesarias. En ambos casos las propiedades `SelectedIndex` y `SelectedItem` permiten obtener/modificar el índice del elemento elegido o directamente el elemento.

Por su parte, la particularidad del control `CheckedListBox` está en que cada elemento aparece como un `CheckBox`, de tal forma que es posible marcarlo y desmarcarlo. Mediante la propiedad `CheckedItems` se obtiene la colección con los elementos que haya marcados en cada momento, mientras que `Items`, como en el caso de las otras listas, contendría todos los elementos, marcados y no marcados.

4.1.5. Otros controles de uso habitual

Aparte de los ya citados en los puntos previos, hay dos controles más cuyo uso es reiterado en toda aplicación Windows: `Label` y `Button`, las etiquetas de texto y los botones. La finalidad del primero es sencillamente mostrar un texto, el que se asigne a la propiedad `Text`, para servir como título o cabecera de listas, secciones y, en general, introducir cualquier tipo de aclaración o descripción en la ventana. Como la mayoría de los controles, `Label` dispone de propiedades como `Font`, `ForeColor` y `BackColor` que permiten configurar el tipo de letra y los colores de tinta y fondo.

El objetivo del control `Button` es desencadenar algún tipo de acción: aceptar o rechazar los datos de un formulario, acceder a una ventana auxiliar para seleccionar un archivo, etc. También cuenta con la propiedad `Text`, a la que asignaremos el título que deba aparecer en el botón, si bien su miembro más importante es el evento `Click` ya que será el que ponga en marcha la acción asociada.

Usando estos dos controles y parte de los que se han descrito en los puntos previos, vamos a preparar una interfaz de usuario como la que aparece en la figura 4.5 (página siguiente). En la parte superior tenemos dos grupos de controles `RadioButton`, supuestamente para elegir el tipo de procesador y la cantidad de memoria. Debajo, a la izquierda, se han colocado dos `CheckBox` y un `ComboBox`. Este último control tiene una lista de posibles configuraciones de disco. A la derecha un control

`CheckedListBox` con otros dispositivos que el usuario puede elegir. Debajo de éste un botón que guardará la configuración en la lista que hay en la parte inferior.

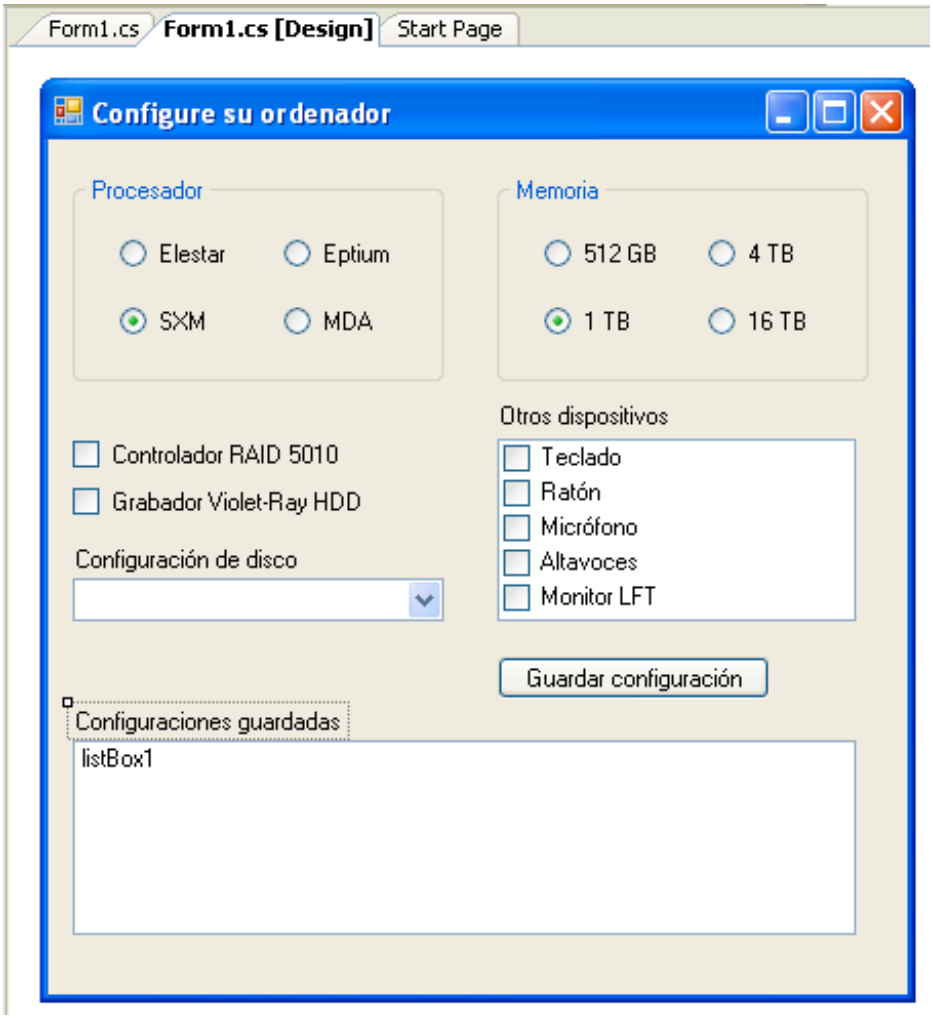

Figura 4.5. Aspecto de la interfaz de usuario a diseñar.

Diseñada la interfaz, haremos doble clic sobre el botón para abrir el método asociado a su evento `Click`, introduciendo en él las sentencias siguientes:

```
// Cadena de caracteres con contenido inicial
string Descripcion = "Procesador ";

// añadimos el texto del RadioButton elegido en el primer grupo
Descripcion += radioButton4.Checked ? radioButton4.Text :
    radioButton5.Checked ? radioButton5.Text :
    radioButton6.Checked ? radioButton6.Text : radioButton7.Text;

Descripcion += ", "; // una coma de separación

// Agregamos el texto del RadioButton elegido en el segundo grupo
Descripcion += radioButton8.Checked ? radioButton8.Text :
    radioButton9.Checked ? radioButton9.Text :
    radioButton10.Checked ? radioButton10.Text : radioButton11.Text;

// indicando que se trata de memoria RAM
Descripcion += " de RAM, ";
```

```
// Se está marcado el primer CheckBox
if (checkBox1.Checked)
    // añadimos que el sistema integra RAID
    Descripcion += "RAID integrado, ";

// Lo mismo para el segundo CheckBox
if (checkBox2.Checked)
    Descripcion += "unidad HDD, ";

// Recorremos los elementos elegidos en el CheckedListBox
foreach (string Dispositivo in checkedListBox1.CheckedItems)
    // agregando el texto de cada uno de ellos a la cadena
    Descripcion += Dispositivo + ", ";

// Finalmente añadimos la configuración de disco elegida de
// la lista desplegable
Descripcion += comboBox1.SelectedItem;

// e insertamos el texto como un nuevo elemento del ListBox
listBox1.Items.Add(Descripcion);
```

Lo que hacemos es ir comprobando el estado de cada `RadioButton`, `CheckBox` y lista guardando en una variable de tipo `string` el texto que, finalmente, añadiremos como nuevo elemento a la lista, usando para ello el método `Add` de la colección `Items`. En la figura 4.6 (página siguiente) puede verse el programa en funcionamiento.

4.2. Cuadros de diálogo de uso común

Determinadas tareas que resultan habituales en muchas aplicaciones, como puede ser la elección de un archivo a abrir o guardar, la selección de un color o de un tipo de letra, suelen requerir una ventana independiente específica, un cuadro de diálogo que no interfiere con la ventana principal y que tiene únicamente esa finalidad concreta. Aunque podemos diseñar nosotros mismos esos cuadros de diálogo, apoyándonos en los servicios de la plataforma .NET que permiten acceder al sistema de archivos, obtener los tipos de letra disponibles, etc., nos resultará mucho más cómodo usar los componentes del grupo **Diálogos** del **Cuadro de herramientas**.

A diferencia de los controles que hemos conocido anteriormente, estos componentes no forman parte de la interfaz y, por ello, al insertarlos en el formulario no quedan en él sino que aparecen en la parte inferior del diseñador, una zona reservada para componentes no visuales. Su propia naturaleza, al no ser controles y, por tanto, no estar derivados de la clase `Control`, hace que carezcan de todas las propiedades, eventos y métodos que se han explicado antes y cuya finalidad es fijar la posición, dimensiones, color, etc.

Salvo por el hecho de contar todos ellos con un método llamado `ShowDialog`, que se encarga de abrir el cuadro de diálogo de acorde a la configuración establecida, lo cierto es que hay pocos elementos comunes entre los cinco componentes salvo si exceptuamos a `SaveFileDialog` y `OpenFileDialog` que comparten un cierto grupo de propiedades.

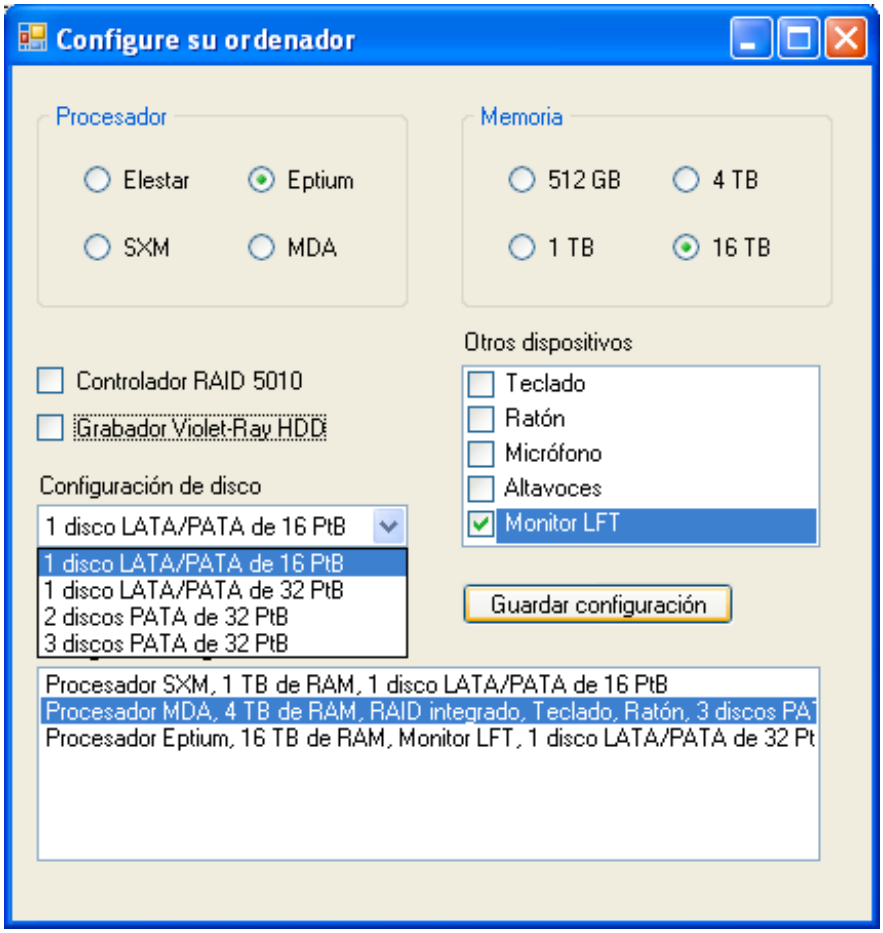

Figura 4.6. El programa en funcionamiento.

Los cinco componentes disponibles son los siguientes:

- `ColorDialog`: Muestra un cuadro de diálogo que permite seleccionar un color. El comportamiento dependerá del valor dado a propiedades como `FullOpen`, `AllowFullOpen`, `SolidColorOnly` y `AnyColor`, que establecen si el cuadro de diálogo muestra todos los paneles de selección abiertos, o puede mostrarlos, si permite únicamente colores sólidos o cualquier color. El color elegido finalmente se recupera de la propiedad `Color`.

- `FontDialog`: Este componente abre una cuadro de diálogo en el que puede seleccionarse un tipo de letra y establecerse sus atributos. Hay disponibles múltiples propiedades de configuración para limitar los tamaños que pueden elegirse (`MinSize` y `MaxSize`), permitir o denegar la selección de ciertos tipos de fuentes de letra (`AllowVectorFonts`, `FixedPitchOnly` y `AllowVerticalFonts`) o habilitar y deshabilitar la configuración del color (`ShowColor`) y efectos de estilo (`ShowEffects`). La fuente puede recuperarse de la propiedad `Font`.

- `OpenFileDialog`: Su finalidad es permitir al usuario seleccionar un archivo existente para abrirlo o usarlo con algún fin en la aplicación. Pueden configurarse filtros de selección de archivos, con las propiedades `Filter` y `FilterIndex`; establecerse un directorio inicial con `InitialDirectory` y

habilitar la selección de múltiples archivos dando el valor `true` a `MultiSelect`. Si queremos asegurarnos de que el camino y archivo indicados existen, no tenemos más que dar ese mismo valor a `CheckPathExists` y `CheckFileExists`. Finalmente, recuperaremos el nombre del archivo elegido de la propiedad `FileName` o, en caso de haberse elegido varios, de `FileNames`.

- `SaveFileDialog`: Es similar al anterior, con la diferencia de que el usuario va a introducir el camino y nombre de un archivo para guardar una información. Por eso además de las propiedades de `OpenFileDialog` tenemos otras específicas, como `OverwritePrompt` que se encarga de avisar del peligro de sobrescribir un archivo existente. La propiedad `FileName` contendrá el nombre del archivo elegido.

- `FolderBrowserDialog`: Con este componente se facilita la selección de una carpeta del sistema de archivos. La carpeta raíz inicial será la que se asigne a `RootFolder`, mientras que la elegida por el usuario la recuperaremos de `SelectedPath`. Si damos el valor `true` a la propiedad `ShowNewFolderButton` el usuario tendrá opción a crear nuevas carpetas.

Para usar cualquiera de estos componentes, una vez insertados en el diseñador y configurados mediante la ventana **Propiedades**, llamaremos al método `ShowDialog` y comprobaremos si éste devuelve el valor `DialogResult.OK`, lo cual significará que el usuario aceptó la selección. En este caso podemos recuperar el color, fuente, nombre de archivo o camino de carpeta y usarlos como nos interese. Por ejemplo, asumiendo que tenemos un componente `ColorDialog` en un formulario en el que hemos insertado también un botón, bastará con hacer doble clic sobre éste e introducir el código siguiente para poder cambiar el color de fondo del formulario estableciendo cualquier otro:

```
if (colorDialog1.ShowDialog() == DialogResult.OK)
    this.BackColor=colorDialog1.Color;
```

4.3. Aplicaciones MDI

En ocasiones las ventanas que necesita abrir una aplicación no son cuadros de diálogo independientes, como los que acabamos de conocer, sino documentos adicionales dentro del mismo entorno de la ventana principal. Con este fin puede utilizarse la técnica conocida como MDI (*Multiple Document Interface*), consistente en tener una ventana que actúa como *madre* de varias ventanas *hija*. Cada una de éstas puede maximizarse, minimizarse y abrirse de manera simultánea al resto de hijas.

Para crear una aplicación MDI lo primero que hemos de hacer es dar el valor `true` a la propiedad `IsMDIContainer` de la ventana principal, usualmente la que aparece por defecto al crear el proyecto. El fondo de dicha ventana cambiará de color, haciéndose más oscura, lo cual la distingue del resto de ventanas. Además en esta ventana no se suelen insertar directamente controles como los que ya conocemos (listas, botones, recuadros de texto), sino que toda su área queda, en principio, disponible para que la ocupen las hijas.

Ciertas partes de la ventana principal, no obstante, pueden ser ocupadas a fin de mostrar menús de opciones, barras de botones y herramientas. El caso más típico es el del menú principal que aparece en la parte superior. Para insertarlo no tenemos más que hacer doble clic sobre el componente `MenuStrip`, en el **Cuadro de herramientas**, y a continuación configurar el título y resto de propiedades de cada opción. El diseñador de menús resulta muy intuitivo: podemos escribir los títulos de las opciones directamente donde aparecerán, usando las teclas de desplazamiento del cursor para agregar nuevas opciones, menús y submenús. En la ventana **Propiedades** podemos configurar aspectos como el atajo de teclado o el icono asociado a cada opción. En la figura 4.7 puede verse la edición de un menú sencillo.

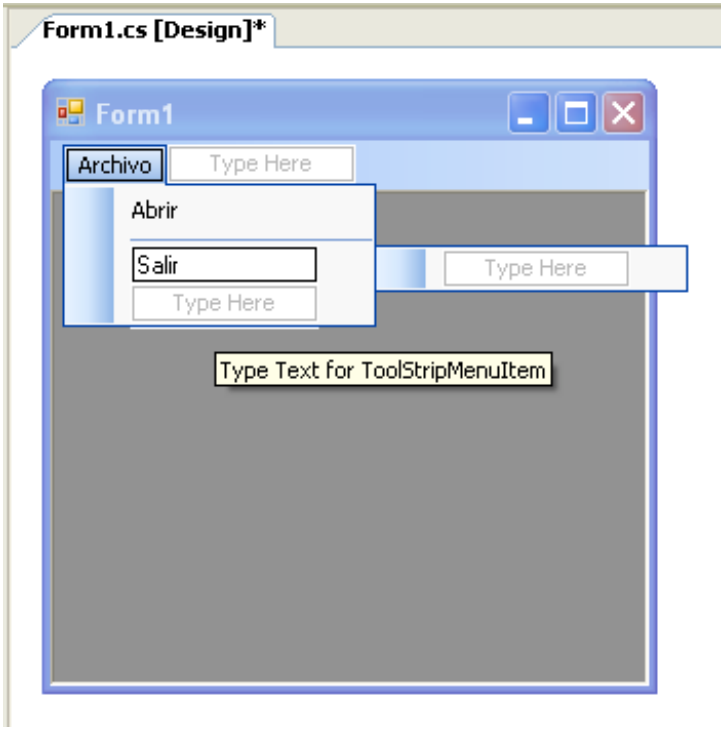

Figura 4.7. Introducimos las opciones del menú.

Cada opción del menú se comporta como un botón, generando el evento `Click` en el momento en que es seleccionada. Encontrándonos en el diseñador, haremos doble clic sobre las opciones para acceder al método de gestión de dicho evento, introduciendo las sentencias que deban ejecutarse en cada caso.

En cuanto a las ventanas hija, son formularios simples con la única particularidad de que su propiedad `MDIParent` contiene una referencia al formulario principal. Dado que en una aplicación MDI pueden abrirse tantos documentos como se necesiten, las ventanas hija van creándose a demanda del usuario, normalmente al elegir una cierta opción.

Suponiendo que tenemos un proyecto con el formulario inicial, a cuya propiedad `IsMDIContainer` ya hemos dado el valor `true`, vamos a agregar un nuevo formulario, eligiendo para ello la opción **Proyecto>Agregar Windows Forms** (véase la figura 4.8, en la pagina siguiente). Daremos a este formulario el nombre `VentanaHija`, identificador que además de actuar como nombre de archivo nos servirá para referirnos al formulario desde el código del programa.

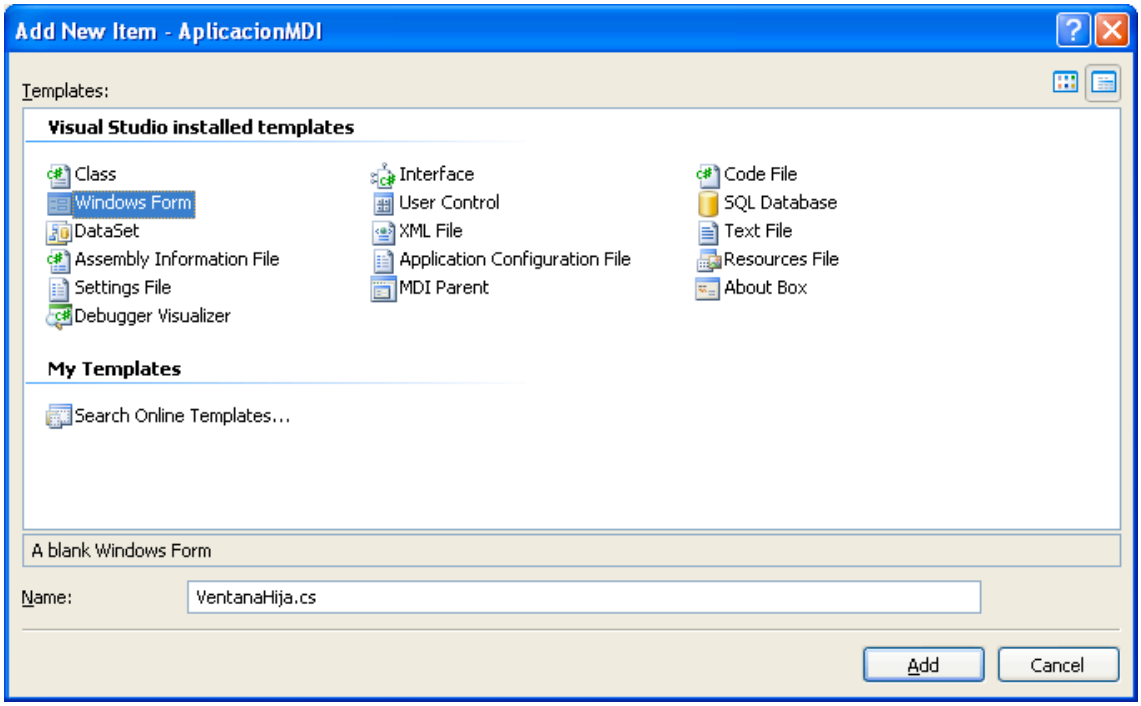

Figura 4.8. Añadimos al proyecto un nuevo formulario Windows.

Insertaremos en el nuevo formulario un recuadro de texto, ya sea un `TextBox` o un `RichTextBox`, ocupando todo el área disponible. El aspecto y diseño de este formulario podría ser cualquiera, usando los controles básicos que ya conocemos.

Puede insertar un control `RichTextBox`, abrir el menú de tareas haciendo clic en el botón que aparece en la esquina superior derecha del control en el diseñador y elegir la opción ***Acoplar en contenedor principal*** *para conseguir que ocupe todo el espacio disponible en la ventana.*

A continuación volveremos al formulario principal, haremos doble clic sobre una de las opciones del menú diseñado previamente e introduciremos las sentencias siguientes en el editor de código:

```
VentanaHija UnaHija = new VentanaHija();

UnaHija.MdiParent = this;
UnaHija.Show();
```

El formulario `VentanaHija` es una clase, como otra cualquiera, y la empleamos para crear un objeto: una ventana hija. Asignamos a la propiedad `MdiParent` de ésta una referencia al formulario principal, representado por la palabra clave `this` ya que estas sentencias están ejecutándose en ese formulario. Finalmente usamos el método `Show` para hacer visible la ventana.

Al ejecutar el programa comprobaremos que es posible usar la opción tantas veces como necesitemos, abriendo nuevas ventanas hija en el entorno de la ventana principal. Cada una de esas ventanas puede ajustarse en tamaño, minimizarse, maximizarse, etc. Recurriendo a los métodos de la ventana principal podríamos cam-

biar la disposición de las ventanas hija, cerrarlas todas, etc. Estas opciones suelen disponerse en un menú **Ventana**.

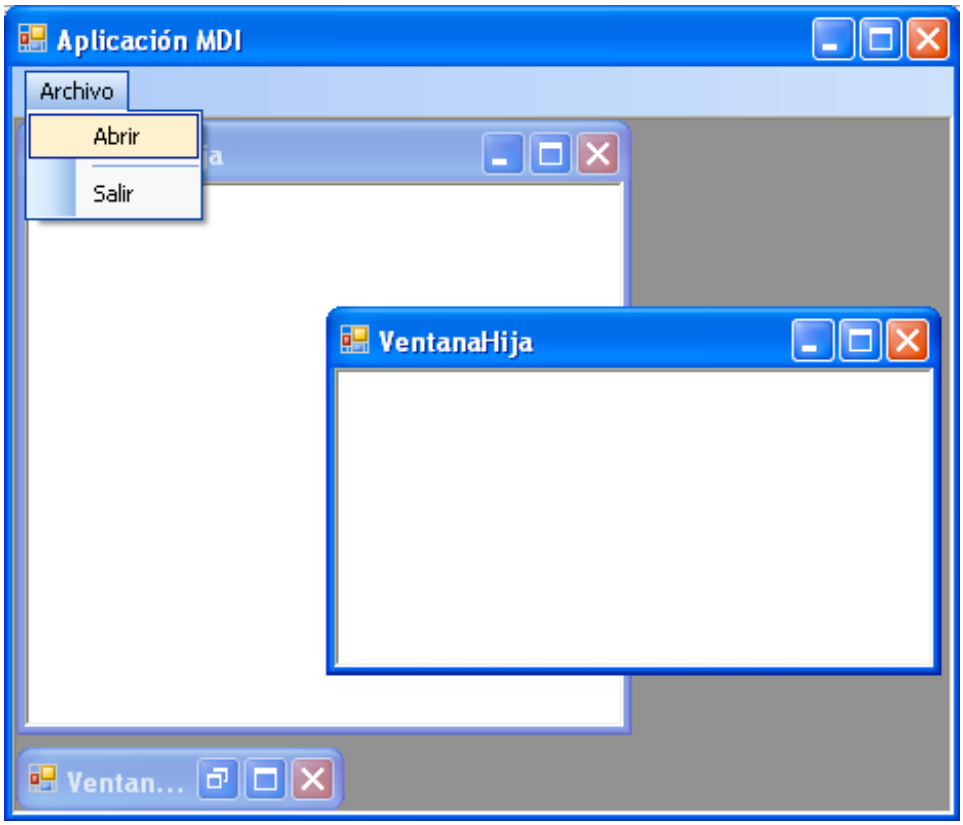

Figura 4.9. Aspecto de la aplicación MDI en funcionamiento.

Resumen

Al concluir el estudio de este capítulo ya sabemos cómo crear aplicaciones Windows con una interfaz gráfica basada en formularios, ventanas en las que es posible insertar controles de distintos tipos y, como hemos visto en el último ejemplo, incluso otras ventanas. Hemos conocido los controles utilizados con más frecuencia, como los recuadros de texto, distintos tipos de botones y listas o menús. Lógicamente hay muchos otros, la lista del **Cuadro de herramientas** es muy extensa y pueden agregarse más. En el sexto capítulo tendremos la oportunidad de conocer algunos de ellos.

Una interfaz de usuario, formada por elementos como los que hemos tratado en este capítulo, no es demasiado útil si detrás no hay una lógica que se encargue de procesar la información. En el primer ejemplo usábamos una serie de sentencias C# para recoger la configuración elegida de un hipotético equipo informático y añadirla a una lista. En el momento en que salgamos del programa, sin embargo, esa información se perderá. Lo habitual es que los datos que deseamos conservar se guarden en un medio persistente, por ejemplo una archivo en disco o una base de datos.

En el capítulo siguiente aprenderemos a trabajar con archivos en disco, guardando y recuperando información. El acceso a bases de datos lo dejaremos para más adelante.

5. Almacenar y recuperar datos de archivos

La plataforma .NET nos ofrece una extensa biblioteca de clases con todo tipo de servicios, clases que residen en unos determinados espacios de nombres que importaremos con sentencias `using`, tras lo cual procederemos a crear objetos y usarlos, sin más. Entre todos esos servicios se encuentran los encargados de facilitar el acceso al sistema de archivos, cuyas clases están alojadas en su mayor parte en el espacio de nombres `System.IO`.

Utilizando estos servicios podremos almacenar la información recogida del usuario en un archivo o bien leer el contenido de un archivo existente y usarlo en nuestro programa. Normalmente estas operaciones van enlazadas: el programa al iniciar su ejecución recupera los datos guardados la última vez.

Mediante estos servicios también resulta posible obtener información procedente del sistema de archivos: unidades de almacenamiento disponibles, espacio libre que hay en ella, listas de directorios y archivos existentes, atributos de cada archivo, etc. En la mayoría de las aplicaciones estos servicios no son precisos, bastando con un componente `OpenFileDialog` para solicitar al usuario el nombre de un archivo a abrir y un `SaveFileDialog` a la hora de guardar información.

En este capítulo nuestro objetivo es aprender a usar parte de las clases existentes en `System.IO`, clases que utilizaremos para abrir archivos, que pueden ser locales o residir en una unidad compartida de red; almacenar y recuperar datos en distintos formatos (textual y binario) y obtener información general sobre los archivos como su tamaño y atributos.

5.1. Primeros pasos

Antes de comenzar a leer o escribir datos de un archivo necesitamos dar una serie de pasos previos, de los cuales el más importante es la apertura del archivo en el modo adecuado. Además necesitaremos conocer los métodos de lectura y escritura que nos ofrecen las distintas clases para, así, recurrir a la más adecuada dependiendo de que vayamos a trabajar principalmente con texto u otros tipos de datos. El último paso será siempre el cierre del archivo.

La clase fundamental para trabajar con archivos en disco es `FileStream`, con métodos generales de lectura y escritura. Ésta puede utilizarse de forma aislada o bien conjuntamente con otras como `StreamReader`/`StreamWriter`, para la lectura/escritura de textos, y `BinaryReader`/`BinaryWriter`, cuando es necesario operar con datos binarios.

5.1.1. Apertura del archivo

El primer paso que debemos dar, previo a la lectura o escritura de datos, es la apertura del archivo, operación para la cual precisaremos conocer el camino y nombre de éste, así como el modo en que queremos abrirlo. Estos argumentos los entregaremos como parámetros al constructor de la clase `FileStream` que, a la postre, será la encargada de efectuar la localización física del archivo, su creación si fuese necesaria, y la apertura. La sentencia a usar sería similar a la siguiente:

```
FileStream Archivo = new FileStream("Datos.txt", modo);
```

El modo determina si debe buscarse un archivo que ya existe, para abrirlo o eliminarlo; si debe crearse un nuevo archivo, etc. Los valores posibles son los de la enumeración `FileMode` (véase la tabla 5.1).

Valores de la enumeración `FileMode`

Constante	Cómo se abre el archivo
`Open`	El archivo debe existir
`CreateNew`	Creándolo únicamente si no existe
`Create`	Creándolo nuevo incluso si ya existe
`OpenOrCreate`	Abriéndolo si existe o creándolo si no existe
`Truncate`	El archivo debe existir y su contenido actual se perderá
`Append`	El archivo debe existir y se añadirán nuevos datos

Tabla 5.1. Valores de numeración `FileMode` para el segundo parámetro de `Filestream`.

Por defecto el archivo, una vez abierto o creado según los casos, estará preparado para admitir tanto operaciones de lectura como de escritura. En caso necesario puede entregarse al constructor de `FileStream` un tercer argumento, uno de los valores de la enumeración `FileAccess`, indicando el acceso que vamos a necesitar. Dichos valores son tres: `Read`, `Write` y `ReadWrite`, según necesitemos solamente leer, solamente escribir o leer y escribir, respectivamente.

El camino entregado como primer argumento al constructor de `FileStream` puede corresponder a una ruta UNC, lo cual nos permite abrir archivos en unidades compartidas de red a las que se tenga acceso.

Un fallo en la apertura del archivo provocará una excepción, no un error durante la compilación, al tratarse de un problema que no es posible prever durante el proceso de desarrollo. Por ello las operaciones con archivos en disco, y otros recursos externos, suelen introducirse en un bloque `try/catch` como se hace en el siguiente fragmento:

```
try {
  Archivo = new FileStream("Datos.txt", Open);
  // otras operaciones con el archivo
} catch {
  // fallo en la apertura del archivo
}
```

5.1.2. Consulta de las operaciones permitidas

Contando con un objeto `FileStream` asociado a un archivo abierto, podemos saber las operaciones que están permitidas antes de intentar ejecutarlas consultando las siguientes propiedades, todas ellas de tipo `bool`:

- `CanRead`: Contendrá el valor `true` si puede leerse del archivo o el valor `false` en caso contrario.
- `CanWrite`: Contendrá el valor `true` si puede escribirse en el archivo o el valor `false` en caso contrario.
- `CanSeek`: Contendrá el valor `true` si es posible desplazar el puntero de lectura o escritura.

En cualquier momento podemos recurrir a la propiedad `Length` de `FileStream` para saber el número de bytes que hay en el archivo, por ejemplo justo después de abrirlo para así leer todo su contenido.

5.1.3. Lectura y escritura de datos

La clase `FileStream` no tiene especialización alguna en cuanto al tipo de información que va a leerse o escribirse en los archivos, razón por la cual sus métodos se limitan a escribir y leer bytes, ya sea de forma individual, uno a uno, o en bloques. Los métodos de que disponemos son los siguientes:

- `ReadByte`: Lee un byte del archivo y lo devuelve como un `int`. En caso de que no haya más datos que leer el valor devuelto será -1.
- `WriteByte`: Toma como argumento un valor de tipo `byte` y lo escribe en el archivo.
- `Read`: Lee una secuencia de bytes del archivo, para lo cual precisa tres argumentos: una matriz de tipo `byte` con el espacio suficiente, un entero indicando la posición de la matriz a partir de la cual puede almacenarse información y otro entero estableciendo el numero de bytes a leer. El método devolverá un entero comunicando el número de bytes leídos, un número que puede ser inferior al solicitado o incluso cero si ya se estaba al final del contenido.
- `Write`: Escribe una secuencia de bytes en el archivo. Los parámetros necesarios son los mismos tres explicados para `Read`, con la diferencia de que

se toman los datos de la matriz y se envían al archivo en lugar de leer del archivo y almacenar en la matriz. Este método no devuelve resultado alguno.

Todos estos métodos operan de manera síncrona. Esto significa que mientras se está llevando a cabo la lectura o escritura de datos nuestro programa quedará a la espera, bloqueando incluso la interfaz de usuario si el bloque de información a transferir es muy grande y la unidad no es local sino remota.

Dado que siempre se opera con bytes individuales o secuencias de bytes, cualquier información que necesitemos escribir o leer deberá ser convertida adecuadamente en un sentido u otro: del tipo original a bytes al escribir y de bytes de vuelta al tipo original al leer. Esas conversiones representan uno de los inconvenientes de usar directamente la clase `FileStream` para leer y escribir, a menos, claro está, que necesitemos precisamente trabajar con secuencias de bytes.

Para convertir una matriz de bytes en una cadena de caracteres, o viceversa, se utilizan las clases `UnicodeEncoding`, `ASCIIEncoding`, `UTF8Encoding` y similares del espacio de nombres `System.Text`. Cada una de ellas efectúa la conversión según una codificación concreta: UNICODE, ASCII, UTF-8, etc.

5.1.4. Posición en el archivo

Cuando se abre un archivo la posición del puntero de lectura/escritura estará siempre al principio salvo si hemos usado el modo `FileMode.Append`, caso éste en que se encontrará al final preparado para añadir información. Esta posición va actualizándose a medida que escribimos o leemos datos, avanzando según el número de bytes leídos o escritos.

Consultando la propiedad `Position` podemos saber en cualquier momento la posición absoluta en que nos encontramos, contada como número de bytes desde el principio del archivo. También podemos efectuar una asignación directa a `Position` para mover el puntero a un posición absoluta.

El desplazamiento por el archivo puede llevarse a cabo también con el método `Seek` de `FileStream`, al que debemos entregar dos argumentos: un entero con el número de bytes a desplazarse y uno de los valores de la enumeración `SeekOrigin` estableciendo el punto de referencia:

- `Begin`: El número de bytes se contará desde el inicio del archivo, por lo que será un desplazamiento absoluto.
- `Current`: El entero facilitado como primer argumento se sumará a la posición actual indicada por `Position`, generando así un desplazamiento relativo del puntero.
- `End`: El número de bytes se contará hacia atrás desde el final del archivo.

Como resultado, el método `Seek` devolverá un número entero que debemos interpretar como la nueva posición en el archivo, posterior a la ejecución del desplazamiento.

5.1.5. Un primer ejemplo

Antes de conocer otras clases relacionadas con la entrada/salida de información a archivos en disco, vamos a usar lo que conocemos de `FileStream` para escribir un sencillo ejemplo. La finalidad de éste será permitir al usuario elegir un archivo de texto, recuperar su contenido y mostrarlo en la ventana en el interior de un recuadro de texto.

Insertaremos en un formulario un control `RichTextBox` y lo ajustaremos para que ocupe todo el espacio disponible, así como un `OpenFileDialog`. Cambiaremos la propiedad `Filter` de éste asignándole el valor `Archivos de texto|*txt`, para que en el cuadro de diálogo aparezcan solamente archivos de texto. Finalmente insertaremos también un `MenuStrip` con una única opción que será la que ejecute las sentencias siguientes:

```
if (openFileDialog1.ShowDialog() == DialogResult.OK) {
    FileStream Archivo = new FileStream(
        openFileDialog1.FileName, FileMode.Open, FileAccess.Read);
    byte[] Contenido;

    Contenido = new byte[Archivo.Length];
    Archivo.Read(Contenido, 0, (int )Archivo.Length);

    richTextBox1.Text = new ASCIIEncoding().GetString(Contenido);
}
```

En caso de que se elija un nombre de archivo, procedemos a abrirlo para lectura, preparamos una matriz con tantos bytes como longitud tenga el archivo, lo leemos completo, convertimos la matriz `byte` en una cadena asumiendo que los archivos usan la codificación ASCII y finalmente asignamos la cadena al recuadro de texto. No es precisamente espectacular, pero al ejecutar el programa podremos acceder a cualquier archivo de texto y ver su contenido.

5.2. Clases especializadas de lectura y escritura de datos

Salvo que prefiramos estar efectuando conversiones con cada escritura y lectura, como en el ejemplo anterior, siempre nos resultará más cómodo usar alguna de las clases especializadas de lectura y escritura en lugar de `FileStream`. En realidad dichas clases trabajan conjuntamente con esta última, por lo que la apertura la seguiríamos realizando igual en un principio. Después facilitaríamos el objeto `FileStream` como parámetro al constructor de una de la siguientes clases:

- `StreamReader`: Lee caracteres y cadenas de caracteres de un flujo de datos.

- `StreamWriter`: Escribe caracteres y cadenas de caracteres en un flujo de datos.

- `BinaryReader`: Lee datos binarios en distintos formatos de un flujo de datos.

- `BinaryWriter`: Escribe datos binarios en distintos formatos en un flujo de datos.

En nuestro caso el flujo de datos sería el objeto `FileStream`, un flujo conectado a un archivo en disco, pero existen otros tipos de flujos de datos: en memoria, cifrados, comprimidos, a través de red, etc.

5.2.1. Trabajar con texto

En caso de que vayamos a operar principalmente con archivos de texto, todo lo que necesitamos son las clases `StreamReader` y `StreamWriter`. Incluso podemos prescindir de `FileStream` si a los constructores de dichas clases entregamos, en lugar de una referencia a un objeto `FileStream`, directamente el nombre del archivo que necesitamos leer o abrir para escritura. Opcionalmente puede indicarse también la codificación de caracteres que debe emplearse, usándose por defecto UTF-8.

Para leer texto tenemos a nuestra disposición los métodos siguientes en la clase `StreamReader`:

- `Read`: Lee un carácter del archivo si se usa sin parámetros. Una segunda versión acepta los mismos parámetros que el método `Read` de `FileStream`, leyendo un bloque de caracteres y almacenándolos en una matriz.

- `ReadLine`: Lee una secuencia de caracteres hasta encontrar una marca de fin de línea, devolviendo un `string` como resultado.

- `ReadToEnd`: Lee desde la posición actual en el archivo hasta el final, devolviendo un `string` con esa parte del contenido. Si acabamos de abrir el archivo este método lo leerá completo.

- `ReadBlock`: Lee un bloque de caracteres y es análogo a la segunda versión del método `Read` tomando los mismos parámetros.

Para convertir cualquier secuencia de caracteres leída de un archivo a otro tipo, por ejemplo `int`, `decimal` o `float`, siempre podemos recurrir al método `Parse` presente en todos esos tipos y que conocimos en un capítulo previo.

A la hora de escribir datos en un archivo usaremos los métodos `Write` y `WriteLine` de `StreamWriter`, funcionalmente equivalentes a los métodos del mismo nombre de la clase `Console` que conocimos en un capítulo previo. Estos métodos, por tanto, cuentan con múltiples versiones que aceptan datos de distintos tipos: cade-

nas de caracteres, números enteros y decimales, matrices, etc., incluyendo la versión que toma como primer argumento una cadena de formato y a continuación una secuencia de parámetros a incluir en ella.

Lo que hemos de tener en cuenta es que lo que se escribirá en el archivo será siempre la representación textual de cada dato, no éste tal cual se almacena en memoria. El contenido de una variable `int`, por tanto, se escribirá como una sucesión de caracteres (dígitos numéricos) y no como cuatro bytes que deben evaluarse en un cierto orden (mayor peso/menor peso) para obtener el número original.

5.2.2. Trabajar con datos binarios

Almacenar los datos en los archivos usando su representación textual tiene ventajas (el archivo puede leerse desde cualquier programa sin problemas) y desventajas (el tamaño es superior y son necesarias conversiones). En ocasiones, cuando las ventajas no son tan importantes, puede ser preferible emplear un formato de almacenamiento binario. Éste se caracteriza por ser bastante más eficiente en el espacio usado, especialmente si se usan datos numéricos, si bien a cambio los archivos no son legibles directamente para el usuario, siendo preciso el uso de un programa para su interpretación.

La clase `BinaryReader` dispone de un amplio abanico de métodos especializados en la recuperación de datos de diferentes tipos: `ReadByte`, `ReadBoolean`, `ReadInt16`, `ReadInt32`, `ReadDouble`, `ReadString`, `ReadChar`, etc. Cada uno de ellos lee una sucesión de bytes del archivo, los interpreta según el tipo de dato que representa y lo devuelve con dicho tipo. Así, el método `ReadInt32` por ejemplo no lee una sucesión de caracteres interpretándolos como dígitos numéricos, sino que extraerá exactamente 4 bytes del archivo, los que ocupa un entero de 32 bits, y los tratará como la representación binaria de dicho entero.

Por su parte, la clase `BinaryWriter` cuenta con múltiples versiones de un mismo método, `Write`, aceptando cada una de ellas un parámetro de un cierto tipo. Es, por tanto, similar al método `Write` de `StreamReader` en este sentido, si bien los datos no se convierten a secuencias de caracteres para ser escritos en el archivo sino que se almacenan según su representación en memoria.

Para comprobar personalmente la diferencia entre los datos textuales y en formato binario puede crear una aplicación de consola e introducir las sentencias siguientes en el método `Main`, ejecutando a continuación el programa:

```
short agno = 2005;
decimal ingresos = 46762993;
bool anotado = true;

StreamWriter Texto = new StreamWriter("Archivo.txt");
Texto.Write(agno);
Texto.Write(ingresos);
Texto.Write(anotado);
Texto.Close();

FileStream Archivo = new FileStream("Archivo.txt", FileMode.Append);
BinaryWriter Binario = new BinaryWriter(Archivo);
```

```
Binario.Write(agno);
Binario.Write(ingresos);
Binario.Write(anotado);
Archivo.Close();
```

Lo que hacemos es crear un archivo, llamado Archivo.txt, escribiendo en él los mismos datos en formato texto y binario, para lo cual usamos primero un `StreamWriter`, a continuación cerramos el archivo y volvemos a abrirlo para añadir datos con un `BinaryWriter`. El archivo contendrá, como se aprecia en la figura 5.1 correspondiente al **Bloc de notas**, una primera parte con los datos perfectamente inteligibles y una segunda, destacada en dicha imagen, con los mismos datos pero en binario.

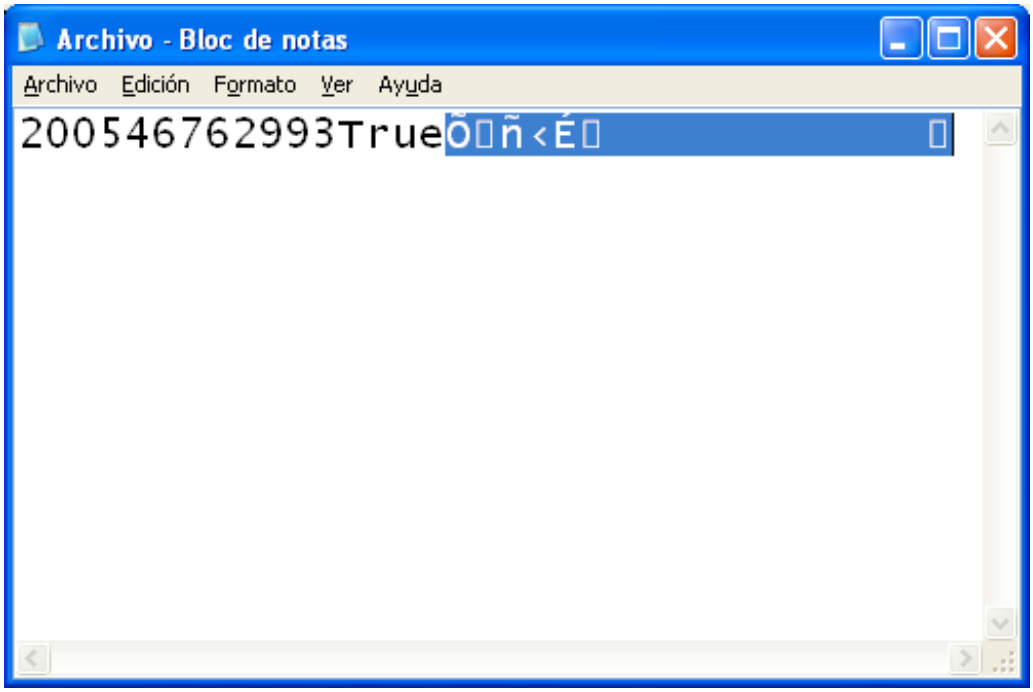

Figura 5.1. Aspecto de los datos en formato textual y binario escritos en el archivo.

5.3. Otras operaciones con archivos

Hasta ahora nos hemos centrado en el acceso al contenido de los archivos, pero éstos son objetos que residen en una unidad de almacenamiento y que pueden manipularse como un todo, por ejemplo al copiarlos o eliminarlos, además de contar con una serie de propiedades como son la longitud, hora de creación y los atributos que determinan si es un archivo de sistema, solamente de lectura, oculto, protegido, etc.

Para trabajar con un archivo podemos recurrir a dos clases distintas: `File` y `FileInfo`. La primera dispone de una serie de métodos estáticos que aportan información sobre el archivo y ejecutan operaciones sobre él, todo sin necesidad de crear un objeto `File` por lo que es preciso aportar el camino y nombre a cada método. `FileInfo`, por el contrario, es una clase pensada para crear objetos a partir de ella, facilitando al constructor el camino y nombre del archivo. A partir de ese momento el objeto está asociado al archivo, pudiendo utilizar sus propiedades y métodos para operar sobre él.

¿Cuándo debemos usar una u otra clase? La respuesta depende de las operaciones que queramos llevar a cabo y, sobre todo, de que pretendamos operar repetidamente con un mismo archivo. Para leer todo el contenido de un archivo de texto, por ejemplo, bastaría con una sentencia como la siguiente, sin necesidad de crear objetos, abrir archivo, etc.:

```
string Contenido = File.ReadAllText("Archivo.txt");
```

El método estático `ReadAllText` de la clase `File` toma como argumento el nombre del archivo, recupera su contenido y lo devuelve como una cadena, todo ello en un solo paso. De manera análoga podríamos usar los métodos `Move`, `Copy` y `Delete` para renombrar el archivo o trasladarlo, copiarlo y eliminarlo; o los métodos `AppendAllText` y `WriteAllText` para añadir un texto a un archivo existente o crear un archivo y escribir un texto en él, todo ello en un único paso.

La clase `FileInfo` resulta más cómoda en el momento que precisemos trabajar con el mismo archivo repetidamente, por ejemplo para obtener información sobre él, leer su contenido y volver a escribirlo. En lugar de usar los métodos estáticos de `File`, facilitando el camino y nombre a cada uno de ellos, procederíamos así:

```
FileInfo Archivo = new FileInfo("Archivo.txt");

string Contenido = Archivo.OpenText().ReadToEnd();
// manipulación del contenido
Archivo.AppendText().Write(Contenido);
```

En la clase `FileInfo` encontramos métodos como `MoveTo`, `CopyTo` y `Delete`, equivalentes a los métodos `Move`, `Copy` y `Delete` de `File`, pero no los métodos capaces de leer o escribir todo el contenido de un archivo. `OpenText` y `AppendText` lo que hacen es crear un `StreamReader` o `StreamWriter` asociado al archivo representado por el objeto `FileInfo`, por lo que tendríamos que recurrir a los métodos que ya conocemos para leer y escribir en un flujo de datos.

A cambio la clase `FileInfo` hace más fácil la recuperación de datos sobre el archivo, ya que expone una serie de propiedades que no existen en `File`. Algunas de ellas son `Length` (tamaño del archivo), `CreationTime` (hora de creación), `LastAccessTime` (hora de último acceso) y `Attributes` (atributos de archivo).

Resumen

En este capítulo hemos conocido las clases fundamentales para poder acceder al sistema de archivos, principalmente con el objetivo de almacenar y recuperar datos de un archivo concreto. Gracias a clases como `StreamReader`, `StreamWriter`, `BinaryReader` y `BinaryWriter` el procedimiento para leer y escribir tanto datos textuales como en formato binario resulta muy sencillo, bastando la creación de un objeto, que representa al archivo en nuestro programa, e invocación a métodos del tipo `Read` y `Write`.

Las clases `File` y `FileInfo` abordadas en el último punto, y que permiten manipular el archivo como un objeto único, se complementan con otras similares, como

`Directory` y `DirectoryInfo`, que actúan sobre directorios en lugar de sobre archivos. No obstante, y según se apuntaba al inicio del capítulo, la aplicación de estas clases en programas corrientes es más limitada, ya que un simple `SaveFileDialog` es suficiente para que el usuario elija el directorio donde quiere guardar un archivo, incluso creándolo si lo necesita.

6. Mejoramos las interfaces de usuario

En el cuarto capítulo conocimos los elementos básicos que permiten crear aplicaciones basadas en formularios Windows, empleando los controles más usuales y sus propiedades fundamentales. Gracias a ellos podemos solicitar información al usuario y mostrársela, ya sea mediante recuadros de texto, listas, etiquetas, botones de radio, etc.

Las interfaces de usuario además de ser funcionales, y servir para la tarea a que están destinadas, también deberían ser lo más cómodas posible. Esto implica que la distribución de los componentes debe resultar siempre clara y consistente, para lo cual es importante seguir unos ajustes de colocación, separación y agrupación homogéneos y lógicos.

Además de esto, las interfaces de usuario deberían ajustarse en la medida de lo posible a las preferencias del usuario final. Si éste modifica el tamaño de la ventana, por ejemplo, el nuevo espacio debe redistribuirse adecuadamente. También puede resultar útil ofrecer paneles o áreas redimensionables.

En este capítulo vamos a conocer algunas propiedades más que la mayoría de controles Windows heredan de la clase `Control`, como las que se encargan de controlar los márgenes entre controles o su anclaje en el contenedor. También trataremos algunos controles nuevos, especialmente los contenedores que se encargan automáticamente de efectuar la distribución de los controles alojados en ellos.

6.1. Posición y separación de los controles

Ya sabemos que la posición de los controles se almacena en las propiedades `Top` y `Left` que todos ellos heredan de `Control`, propiedades que normalmente establecemos de forma indirecta a medida que colocamos los controles en su lugar mediante el diseñador de formularios. La asignación de una posición y unas dimensiones fijas a cada control no siempre resulta el método más adecuado, especialmente si la aplicación va a tener que funcionar en diferentes resoluciones y ajustarse a distintos tamaños de ventana.

Para promover el diseño de interfaces *fluyentes* (que se ajustan automáticamente al espacio disponible) la versión 2.0 de la plataforma .NET incorpora como novedad los componentes `TableLayoutPanel` y `FlowLayoutPanel`. La finalidad de éstos es distribuir los controles en forma de tabla o de flujo (de izquierda a derecha y arriba a abajo, por ejemplo), sin importar en un principio propiedades como `Top`, `Left`, `Width` y `Height`.

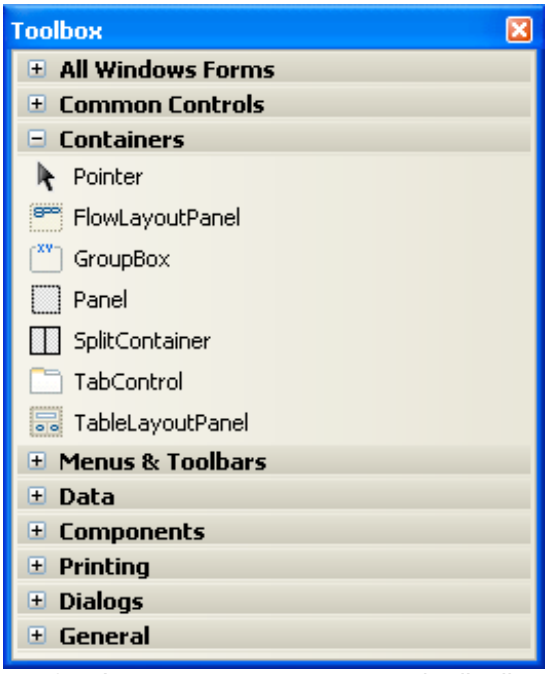

Figura 6.1. Los nuevos componentes de distribución de controles los encontramos en la sección **Contenedores** del **Cuadro de herramientas**.

Conseguir que una interfaz fluyente funcione adecuadamente requiere que los controles se comporten de manera ligeramente distinta a la que hemos estado acostumbrados en versiones previas. En lugar de un tamaño absoluto deberían comunicar cuáles son las dimensiones mínimas y el tamaño preferente. En lugar de una posición tendrían que establecer una separación, unos márgenes, respecto a los demás elementos de la interfaz. Con este fin se han agregado a la clase `Control` una serie de propiedades y métodos nuevos, heredados por todos los controles Windows.

6.1.1. Configuración de los márgenes

Cada control dispone de una configuración de margen exterior, de separación respecto a los controles que haya a su alrededor, y otra de margen interior, de separación entre su propio contorno y el contenido que vaya a mostrar. El primero se almacena en la propiedad `Margin` y el segundo en la propiedad `Padding`. Ambas propiedades son estructuras de tipo `Padding`, cuyo constructor cuenta con dos versiones:

- `Padding(int margen)`: Usa el mismo margen para todos los lados: izquierdo, derecho, superior e inferior.
- `Padding(int izq, int sup, int der, int inf)`: Establece los márgenes indicados para cada lado.

También podemos acceder a cada miembro por separado, a través de las propiedades `Left`, `Top`, `Right` y `Bottom`, así como por grupos, con las propiedades `Horizontal` y `Vertical`. Leyendo la propiedad `Horizontal`, por ejemplo, sabríamos cuál es el margen horizontal total, es decir, la suma de los márgenes izquierdo y derecho.

Por defecto la mayoría de los controles cuentan con un margen interno de 0 puntos y externo de 3 puntos. Podemos comprobarlo insertando en un formulario un control cualquiera, por ejemplo un botón, y examinando sus propiedades `Margin` y `Padding` en la ventana **Propiedades**. La primera contendrá el valor 3 en todos sus miembros, mientras que la segunda tendrá el valor 0.

*Todas las propiedades relacionadas con la posición, dimensiones, márgenes y anclaje de los controles se agrupan en la categoría **Diseño** de la ventana **Propiedades**, lo cual facilita su edición conjunta.*

6.1.2. Ajuste de las dimensiones

En principio las dimensiones de un control serán las que se asignen a las propiedades `Width` y `Height`, estableciendo un ancho y un alto constantes. Ese tamaño puede ajustarse automáticamente, en ejecución, si damos el valor `true` a la propiedad `AutoSize`, que por defecto tiene el valor `false`. El cambio de tamaño puede venir provocado por distintas causas: un cambio de resolución o de tipo de letra del sistema, un ajuste de las dimensiones de la propia ventana o la modificación del contenido del control.

La propiedad `AutoSizeMode` determina cómo cambiará el tamaño del control. Su valor por defecto (`GrowOnly`) indica que solamente debe modificarse para incrementarlo, pero no para reducirlo. Podemos usar el valor `GrowAndShrink` para conseguir que el tamaño del control se ajuste libremente, aumentándolo o reduciéndolo, a fin de alcanzar el resultado óptimo.

Si optamos porque las dimensiones de los controles puedan alterarse durante la ejecución, ajustándose según convenga, aún mantenemos un cierto control sobre el tamaño que tendrá a través de las propiedades `MinimumSize` y `MaximumSize`. Ambas son estructuras de tipo `Size` y, por tanto, cuentan con los miembros `Width` y `Height`. Independientemente del contenido del control y otros agentes externos, su tamaño nunca será superior a lo indicado por `MaximumSize` ni inferior a lo establecido en `MinimumSize`.

Existe una serie de eventos que pueden resultarnos de interés a la hora de gestionar una interfaz en la que la posición y el tamaño de los controles se ajustan dinámicamente. Algunos de ellos son:

- `Move`: Notifica que el control ha cambiado de posición.
- `Resize`: Comunica que el tamaño del control ha sido modificado.
- `Layout`: Este evento indica al control que debe ajustar su contenido.

Preferentemente optaremos por usar el evento `Layout` para verificar los ajustes hechos al control y, en caso necesario, aplicar cualquier modificación.

6.1.3. Anclajes y acoplamiento

Otras dos propiedades que nos serán de suma utilidad, a la hora de mejorar el diseño de nuestras interfaces de usuario, son `Anchor` y `Dock`, encargadas de controlar los anclajes del control en su contenedor y el acoplamiento a uno de los márgenes de éste.

Por defecto los controles están siempre anclados respecto a la esquina superior izquierda del contenedor en que están alojados. Por ello se desplazan solidariamente cuando dicha esquina cambia de posición, pero no cuando lo hace la esquina opuesta. Este comportamiento puede modificarse a través de la propiedad `Anchor`, a la que pueden asignarse, mediante combinación lógica, uno o más de los valores de la enumeración `AnchorStyles`:

- `None`: El control no está anclado a ningún margen del contenedor.
- `Left`: Anclado al margen izquierdo.
- `Right`: Anclado al margen derecho.
- `Top`: Anclado al margen superior.
- `Bottom`: Anclado al margen inferior.

Si queremos que un control conserve su posición relativa a la esquina inferior derecha, por ejemplo un botón situado en ese punto que queremos que permanezca ahí incluso si el formulario cambia de tamaño, usaríamos conjuntamente los valores `Right` y `Bottom`, en lugar de `Left` y `Top` que son los empleados por defecto. En caso de asignar a `Anchor` dos de los valores opuestos, por ejemplo `Left` y `Right`, se conseguirá anclar el control a ambos lados y, por tanto, *estirarlo* aumentando o reduciendo sus dimensiones según cambien las del contenedor. Lo mismo es aplicable en sentido vertical. Incluso es posible usar los cuatro valores de manera simultánea, obteniendo un control que refleja proporcionalmente los cambios de posición y tamaño del contenedor.

En la fase de diseño configurar la propiedad `Anchor` nos resultará muy sencillo gracias al editor específico con que cuenta dicha propiedad (véase la figura 6.2). Solamente tenemos que marcar los márgenes a los que queremos anclar el control.

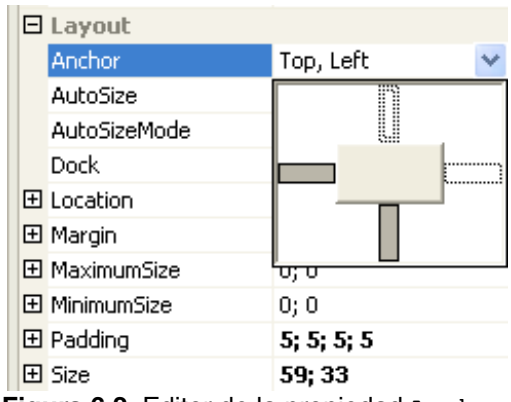

Figura 6.2. Editor de la propiedad `Anchor`.

La propiedad `Dock` representa otra forma de mantener el control en una cierta posición y con un cierto tamaño, siempre respecto al contenedor donde esté insertado. Por defecto esta propiedad contiene el valor `DockStyle.None`, por lo que el control se mantiene en la posición y con el tamaño que indican el resto de propiedades. Al usar cualquier otro de los valores de la enumeración `DockStyle`, sin embargo, el control pasará a estar acoplado a uno de los márgenes del contenedor, ocupándolo de extremo a extremo. Esos valores son `Top`, `Bottom`, `Left`, `Right`, `Bottom` y `Fill`. Los cuatro primeros acoplan el control al margen superior, inferior, izquierdo y derecho, mientras que el último ajustaría su tamaño para que ocupe todo el espacio libre en el contenedor.

También la propiedad `Dock` dispone, como se aprecia en la figura 6.3, de un editor a medida durante la fase de diseño. Basta con hacer clic en el botón que representa el margen a donde quiere acoplarse el control.

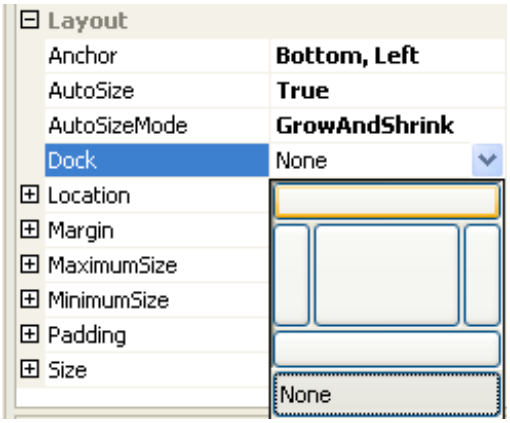

Figura 6.3. Editor de la propiedad `Dock`.

Aunque podemos acoplar cualquier control a un margen del formulario, lo habitual es que lo que se acople sea un contenedor, por ejemplo un `Panel` o un `GroupBox`, usando éste como recipiente para el resto de controles: botones, listas, recuadros de texto, etc.

6.2. Distribución automática de los controles

Dependiendo de las características de la interfaz de usuario que estemos diseñando, es posible que con lo que conocemos hasta el momento tengamos más que suficiente. Al componer un cuadro de diálogo, por ejemplo, normalmente la ventana no será redimensionable por el usuario y, por tanto, los controles mantendrán una posición fija.

En diseños más complejos, podemos tomar como referencia el propio entorno de *Visual C# 2005 Express Edition*, las ventanas cambian de tamaño y tienen en su interior distintos paneles con controles y otras ventanas. En algunos casos los controles fluyen en forma de tabla, por ejemplo en la ventana **Propiedades**, mientras que en otros se distribuyen de izquierda a derecha o de arriba a abajo, por ejemplo en las barras de botones y herramientas que están en la parte superior pero podemos colocar en cualquier margen.

Crear una interfaz de este tipo no tiene necesariamente que requerir mucho trabajo por nuestra parte, solamente necesitamos conocer algunos componentes más y saber cómo usarlos. Los más importantes son `TableLayoutPanel`, `FlowLayoutPanel`, `SplitContainer` y `TabControl`.

6.2.1. Distribución en forma de tabla

Cuando un cierto grupo de controles van a organizarse en una serie de filas y columnas, incluso si no es totalmente regular y hay filas con más o menos columnas, el contenedor al que debemos recurrir es `TableLayoutPanel`. Recién insertado éste tendrá dos filas con dos columnas cada una de ellas, pudiendo alojarse en cada celda únicamente un control.

A través de las opciones del menú emergente del control `TableLayoutPanel` podemos añadir, insertar y eliminar tanto filas como columnas, así como modificar el ajuste de cada una de ellas que puede tener unas dimensiones fijas (en puntos), relativas (en tanto por ciento) o dinámicas (ajustándose al contenido). También hay disponible una ventana de estilos de filas y columnas (véase la figura 6.4) que permite configurar con más comodidad este aspecto.

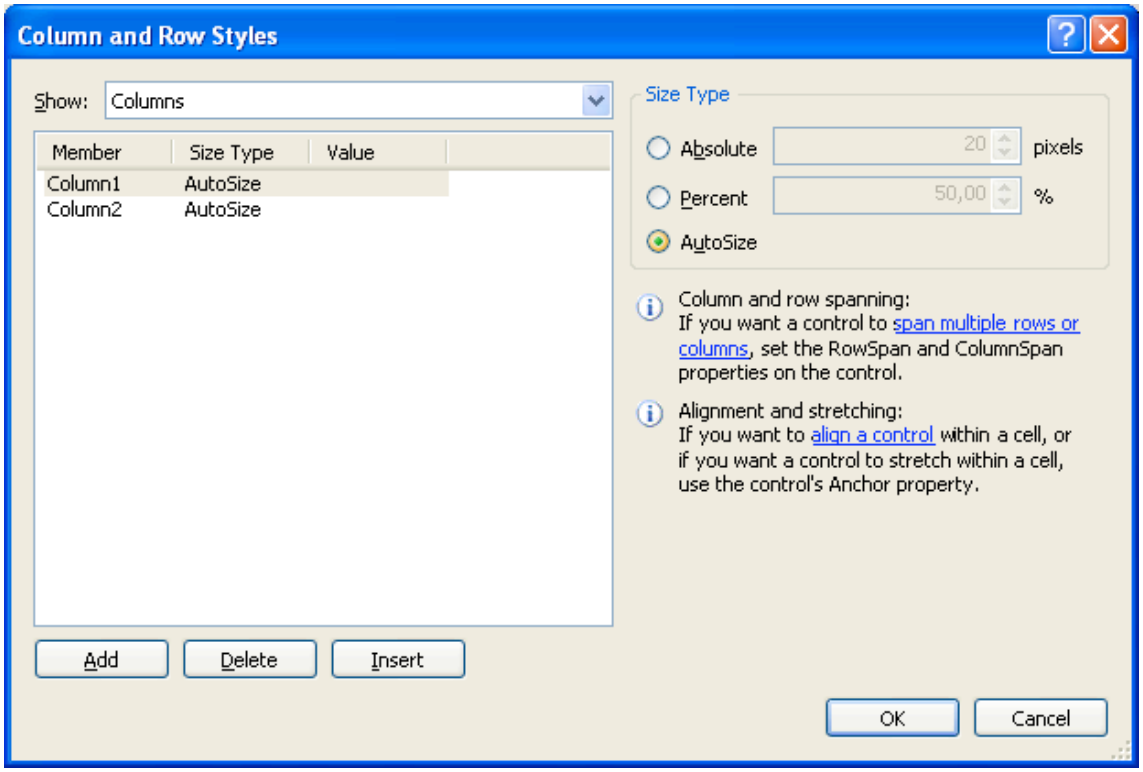

Figura 6.4. Ventana para ajustar el tamaño de filas y columnas, añadirlas y eliminarlas.

Los controles insertados en las celdas de un `TableLayout` adquieren una serie de propiedades de diseño con las que normalmente no cuentan, como `Cell`, `Column`, `Row`, `ColumnSpan` y `RowSpan`. Las tres primeras establecen la celdilla, columna y fila en la que aparecerá el control dentro de la tabla, la cuarta indica el número de columnas que abarca el control y la última el número de filas que ocupa. De esta forma es

posible introducir controles que ocupen más de una columna o fila si ello fuese necesario, dando lugar a tablas que no siguen una distribución completamente regular.

Las columnas y filas pueden manipularse en ejecución, por ejemplo añadiendo otras nuevas, a través de las propiedades `Columns` y `Rows` de `TableLayoutPanel`, dos colecciones en las que cada columna o fila aparece como un objeto. Otras propiedades interesantes de este control son `CellBorderStyle`, que configura el tipo de borde entre celdas de la tabla; `GrowStyle`, que determina si la tabla es de dimensiones fijas, crecerá añadiendo nuevas filas o nuevas columnas, o `AutoScrollMargin` y `AutoScrollMinSize`, encargadas de la configuración de barras de desplazamiento en caso de que sean precisas.

En la figura 6.5 puede ver cómo se ha insertado en un formulario un control `TableLayoutPanel` con dos columnas y una serie de filas. La primera fila tiene un control `Label` a cuya propiedad `ColumnSpan` se ha dado el valor 2, para que ocupe ambas columnas. Las demás tienen distintos controles. El propio `TableLayoutPanel` se ha acoplado al margen derecho del formulario, modificando la propiedad `Dock`. También se ha dado el valor `true` a la propiedad `AutoSize` de todos los controles, a fin de que se ajusten automáticamente lo mejor posible.

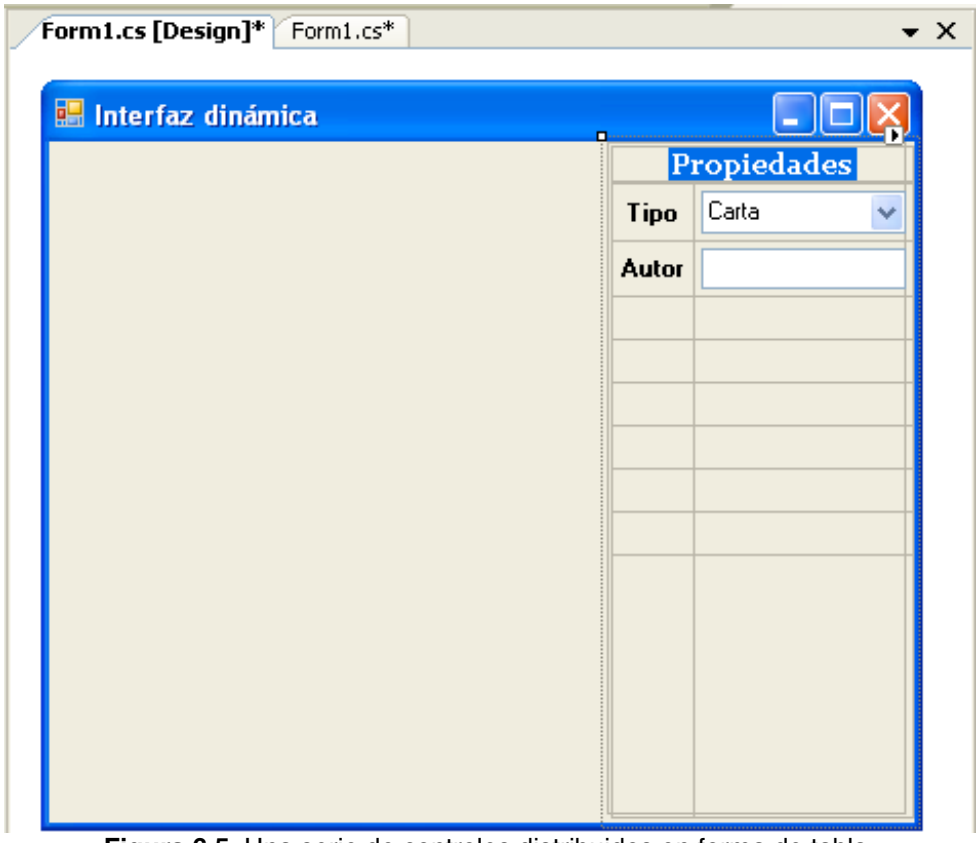

Figura 6.5. Una serie de controles distribuidos en forma de tabla.

6.2.2. Distribución como flujo

Si lo que tenemos que colocar en la interfaz de usuario es una secuencia de controles relacionados entre sí pero sin una estructura de tabla, podemos recurrir al control

`FlowLayoutPanel`. Este contenedor es más simple que `TableLayoutPanel` ya que no hay columnas, filas, bordes ni nada parecido, simplemente una sucesión de controles que van a distribuirse como un flujo de objetos conforme a lo que dicten sus propiedades de separación y tamaño.

La dirección, vertical u horizontal, y el sentido, de izquierda a derecha, de derecha a izquierda, de arriba a abajo o de abajo a arriba, en que se distribuyan los controles insertados en el `FlowLayoutPanel` dependerán del valor que asignemos a la propiedad `FlowDirection`. El tomado por defecto es `LeftToRight`, colocando los componentes de izquierda a derecha, existiendo también `RightToLeft`, inverso al anterior; `TopDown`, de arriba a abajo, y `BottomUp`, inverso a `TopDown`.

Al insertar un `FlowLayoutPanel` en el formulario lo único que veremos será un contorno. A medida que insertemos controles apreciaremos cómo éstos se distribuyen según la opción elegida en `FlowDirection`. En caso de que haya más controles que espacio disponible podemos encontrarnos en dos situaciones posibles:

- `AutoScroll`: Si hemos dado el valor `true` a esta propiedad, aparecerán automáticamente unas barras de desplazamiento cuando sean precisas para poder acceder a todo el contenido del `FlowLayoutPanel`.

- `WrapContents`: Esta propiedad por defecto tiene el valor `true`, de tal forma que los controles van saltando de línea cuando no queda más espacio disponible. Si además damos el valor `true` a la propiedad `AutoSize` conseguiremos que el `FlowLayoutPanel` ajuste su tamaño a fin de mostrar siempre todo su contenido.

En la figura 6.6. (en la página siguiente) puede verse cómo hemos añadido al mismo formulario anterior un `FlowLayoutPanel`, usando su propiedad `Dock` para adosarlo a la parte superior. A continuación hemos dado el valor `true` a la propiedad `AutoSize` y dejado ese mismo valor en `WrapContents`. Finalmente hemos insertado una serie de controles que teóricamente conformarían la barra de herramientas de la aplicación. Observe que el último de ellos, una lista desplegable, aparece en una segunda línea, al no haber espacio para él en la primera.

6.2.3. Distribución en múltiples páginas

Una de las técnicas más eficientes a la hora de economizar espacio, en el diseño de una interfaz de usuario, consiste en repartir los elementos en varias páginas accesibles mediante pestañas. Es lo que hace *Visual C# 2005 Express Edition*, por ejemplo, para facilitar el acceso a los distintos documentos que podemos tener abiertos de forma simultánea en el entorno.

El control `TabControl` es un contenedor capaz de mostrar varias páginas, cada una de ellas identificada por una pestaña y con el conjunto de controles que nos interesen. Las páginas son accesibles a través de la propiedad `TabPages`, una colección de objetos `TabPage` cada uno de los cuales representa a una página. Como todas las colecciones, ésta dispone de los métodos habituales que hacen posible agregar, eliminar y, en general, manipular los elementos de la colección.

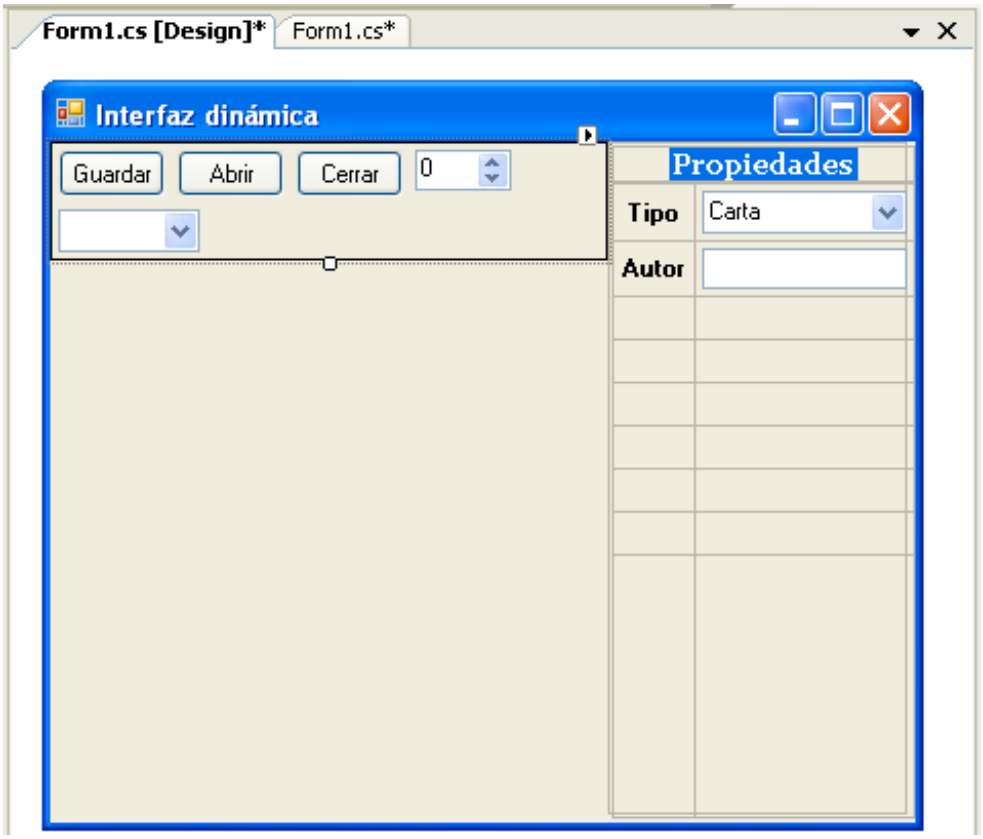

Figura 6.6. Aspecto del formulario tras insertar el control `FlowLayoutPanel`.

En la fase de diseño es fácil añadir y eliminar páginas mediante las opciones a medida que aparecen en el menú contextual (véase la figura 6.7, en la página siguiente). Cada página dispone de una serie de propiedades que podemos editar en la ventana **Propiedades**, como es habitual, estableciendo el texto que aparecerá en la pestaña (propiedad `Text`), el color de fondo (propiedad `BackColor`), borde (propiedad `BorderStyle`), tipo de letra (propiedad `Font`), etc.

*Para seleccionar en el diseñador el control `TabControl`, a fin de acceder a sus propiedades, haga clic sobre cualquiera de las pestañas. Si quiere seleccionar una de las páginas, el objeto `TabPage`, haga clic en el fondo de una página. También puede desplegar la lista de la ventana **Propiedades** y escoger directamente el componente sobre el que desea operar.*

La propiedad `TabPages` dispone de un editor específico (véase la figura 6.8, también en la página siguiente) que facilita la edición, eliminación y reordenación de las páginas, así como la edición de sus propiedades. Los botones con icono de flechas que hay junto a la lista de la izquierda sirven para desplazar arriba y abajo la página elegida, cambiando el orden en que aparecen en la interfaz.

En general resulta más cómodo usar este cuadro de diálogo, donde tenemos todas las operaciones del `TabControl` al alcance, que trabajar directamente sobre el diseñador de formularios, teniendo que abrir el menú contextual, seleccionar la página a modificar, etc.

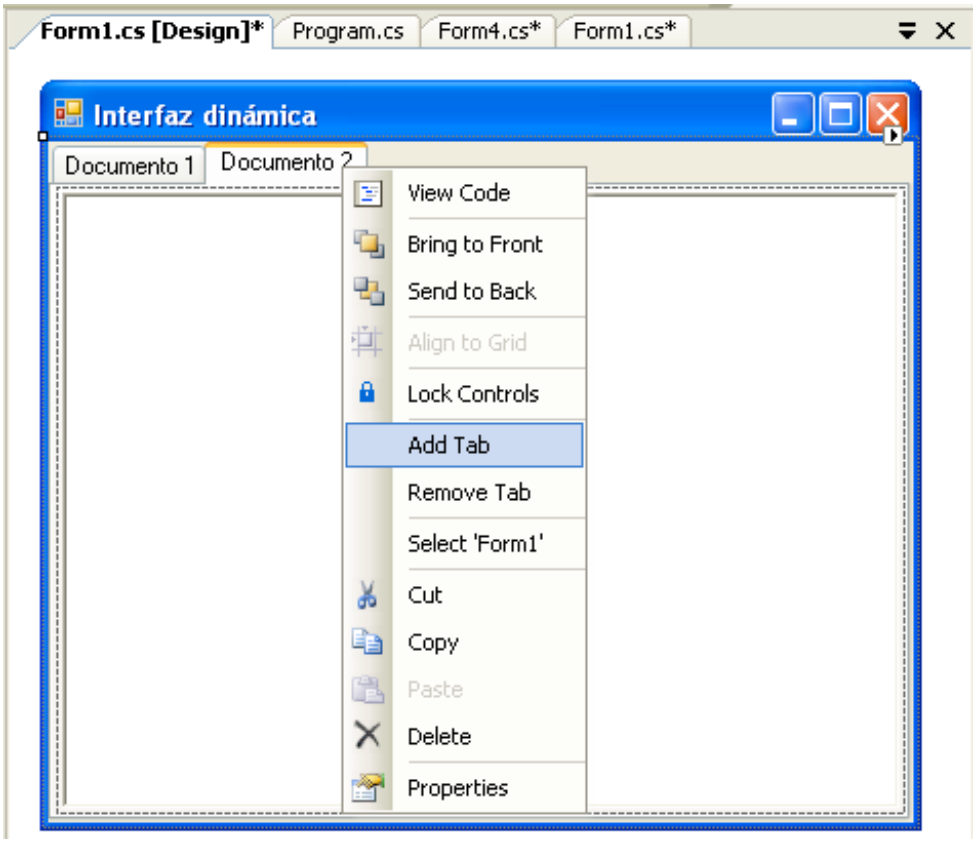

Figura 6.7. Añadimos páginas al `TabControl`.

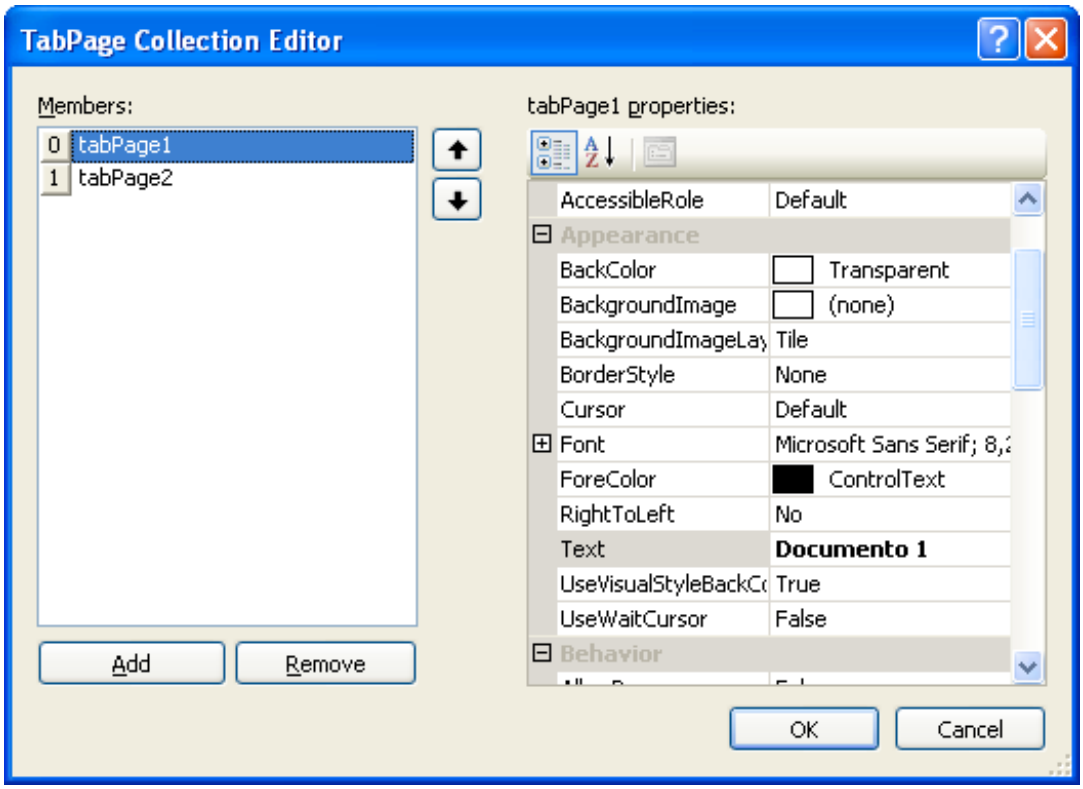

Figura 6.8. Editor específico para la propiedad `TabPages` del control `TabControl`.

6.2.4. Paneles ajustables por el usuario

Los tres contenedores descritos en los puntos previos pueden acoplarse al formulario, o el interior de otro contenedor, así como anclarse y ajustar su tamaño dinámicamente. Las propiedades `Dock`, `Anchor` y `AutoSize` son tan válidas para estos componentes como para el resto de los controles. Los usuarios, sin embargo, no obtienen control alguno sobre las dimensiones individuales de cada contenedor, aunque indirectamente, al cambiar el tamaño de la ventana, sí lo tengan.

Para dar a los usuarios la posibilidad de que ajusten el ancho o alto de las distintas secciones de la interfaz, llegando incluso a ocultarlo si es preciso, usaremos el control `SplitContainer`. Éste es un contenedor con dos secciones, en sentido horizontal o vertical dependiendo del valor asignado a la propiedad `Orientation`, con una línea de división intermedia de tipo interactivo. Esto significa que una sencilla operación de arrastrar y soltar es suficiente para realizar ajustes, aumentando el espacio de una sección al tiempo que se reduce el de la otra.

Al igual que `TableLayoutPanel` y `FlowLayoutPanel`, el control `SplitContainer` es una novedad de la plataforma .NET 2.0, no existente en versiones previas de Visual C#.

Las propiedades más interesantes de este control, aparte de la ya mencionada `Orientation` que establece la orientación de los paneles, son las siguientes:

- `IsSplitterFixed`: Esta propiedad es la que indica si el divisor existente entre los dos paneles puede desplazarse o, por el contrario, tiene una posición fija. Por defecto tiene el valor `false`, la posición no es fija, y realmente no tiene mucho sentido darle el valor `true` ya que en ese caso se tendrían dos paneles de dimensiones fijas.

- `FixedPanel`: En principio al actuar sobre la línea de división se cambia el tamaño, ancho o alto, de los dos paneles, a menos que usemos esta propiedad para indicar cuál de los dos tendrá un tamaño inamovible. El valor por defecto es `None`, pudiendo tomar además los valores `Panel1` y `Panel2`.

- `Panel1MinSize` y `Panel2MinSize`: Si optamos porque ambos paneles puedan cambiar de tamaño, mediante estas propiedades podemos establecer las dimensiones mínimas que pueden alcanzar. Es la forma de limitar la actuación del usuario hasta un cierto punto.

- `Panel1Collapsed` y `Panel2Collapsed`: Mediante estas dos propiedades es posible ocultar uno de los dos paneles, sencillamente dando el valor `true` a la que convenga. Si disponemos un botón o elemento similar en la interfaz, el usuario podrá mostrar y ocultar los paneles a demanda.

El aspecto quizá más interesante es que todos estos controles contenedores pueden anidarse unos dentro de otros, dando lugar a interfaces verdaderamente

elaboradas. Así, un `SplitContainer` puede contener a otros, así como a un `TableLayoutPanel` o `FlowLayoutPanel`, y éstos a un `TabControl`. Solamente tenemos que usar nuestra imaginación para ofrecer el mejor diseño y funcionalidad posibles.

6.2.5. Un diseño de ejemplo

Veamos en la práctica cómo usar algunos de los controles que se han descrito en estos últimos puntos, confeccionando una interfaz que, en el diseñador de formularios, tendrá el aspecto que puede verse en la figura 6.9. El formulario puede cambiarse de tamaño y el espacio se reparte de manera coherente, ocupando la barra superior el espacio que sea necesario para mostrar sus controles y quedando el resto para el área de edición de texto y el símil de ventana de propiedades a la derecha. El ancho de estos elementos será ajustable en ejecución.

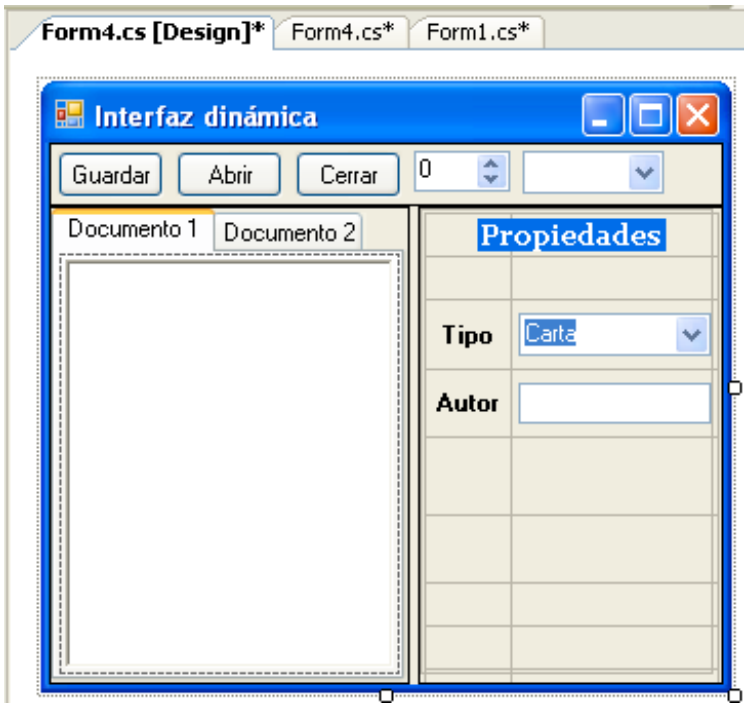

Figura 6.9. Interfaz a diseñar.

Los pasos a seguir, partiendo del formulario inicialmente vacío, para llegar a este resultado son los siguientes:

- Insertamos un `FlowLayoutPanel` y damos el valor `Top` a la propiedad `Dock` para que se ajuste a la parte superior. El alto inicial será el necesario para mostrar los tres botones y demás controles que vamos a insertar en él. Daremos el valor `true` a la propiedad `AutoSize` para que en caso necesario ese alto aumente, a fin de mostrar todos los controles.

- Insertamos un `SplitContainer` en el formulario. Automáticamente ocupará todo el espacio disponible y mostrará un panel a la izquierda y otro a la derecha. Cambiamos el valor de la propiedad `BorderStyle` a fin de que el borde, referencia para ajustar el ancho en ejecución, sea visible.

- Insertamos en el panel izquierdo del `SplitContainer` un `TabControl`, al que añadiremos una segunda página. Establecemos los títulos de éstas y modificamos la propiedad `Dock` del `SplitContainer` para que ocupe toda el área disponible.

- Insertamos en cada página del `TabControl` un `RichTextBox`, usando una vez más la propiedad `Dock` para usar todo el espacio disponible.

- Insertamos en el panel derecho del `SplitContainer` un `TableLayoutPanel` con dos columnas, supuestamente para servir como página de las propiedades del documento que está editándose en la página activa del `TabControl`. Modificaremos la propiedad `Dock` del `TableLayoutPanel` para que use todo el panel.

- Finalmente hacemos doble clic sobre el botón **Abrir** que hemos insertado en el `FlowLayoutPanel` e introducimos el código mostrado a continuación, permitiendo así añadir en ejecución nuevas páginas con supuestos documentos.

```
// La variable N está definida como "int N = 3;" al inicio

// Agregamos una nueva página al TabControl estableciendo el título
tabControl1.TabPages.Add("Documento " + N);

// Creamos un nuevo control RichTextBox
RichTextBox RecuadroTexto = new RichTextBox();
// y modificamos su propiedad Dock
RecuadroTexto.Dock = DockStyle.Fill;

// Insertamos en la página recién añadida el RichTextBox
tabControl1.TabPages[tabControl1.TabPages.Count - 1].Controls.Add(
    RecuadroTexto);

N++; // incrementamos N para la página siguiente
```

En este momento ya podemos ejecutar el programa y comprobar el comportamiento de la interfaz. Si reducimos el ancho de la ventana lo suficiente, comprobaremos cómo el `FlowLayoutPanel` aumenta su altura para dar cabida a los controles que ya no quepan a la derecha. Los dos paneles del `SplitContainer` se ajustan a medida que las dimensiones de la ventana cambian, pudiendo también modificarse su anchura actuando directamente sobre la línea de división. Podemos cambiar de una página a otra del `TabControl`, así como agregar nuevas páginas haciendo clic sobre el botón **Abrir** y escribir texto.

Para completar este ejemplo, y ya como ejercicio de práctica, podría agregar un componente `OpenFileDialog` y un `SaveFileDialog` y escribir código asociado a los botones que hay en la parte superior para recuperar y almacenar el texto de un archivo, usando para ello los servicios que conocimos en el capítulo previo. Además del texto del `RichTextBox`, en el archivo podrían guardarse también las propiedades introducidas en los controles del panel derecho, propiedades que se recuperarían al leer el archivo y volverán a aparecer en dicho panel.

En la figura 6.10 puede verse un momento de la ejecución del programa, tras haber añadido una página más al `TabControl`, haber cambiado el tamaño de la ventana y estar arrastrando la línea de división entre los dos paneles del `SplitContainer`.

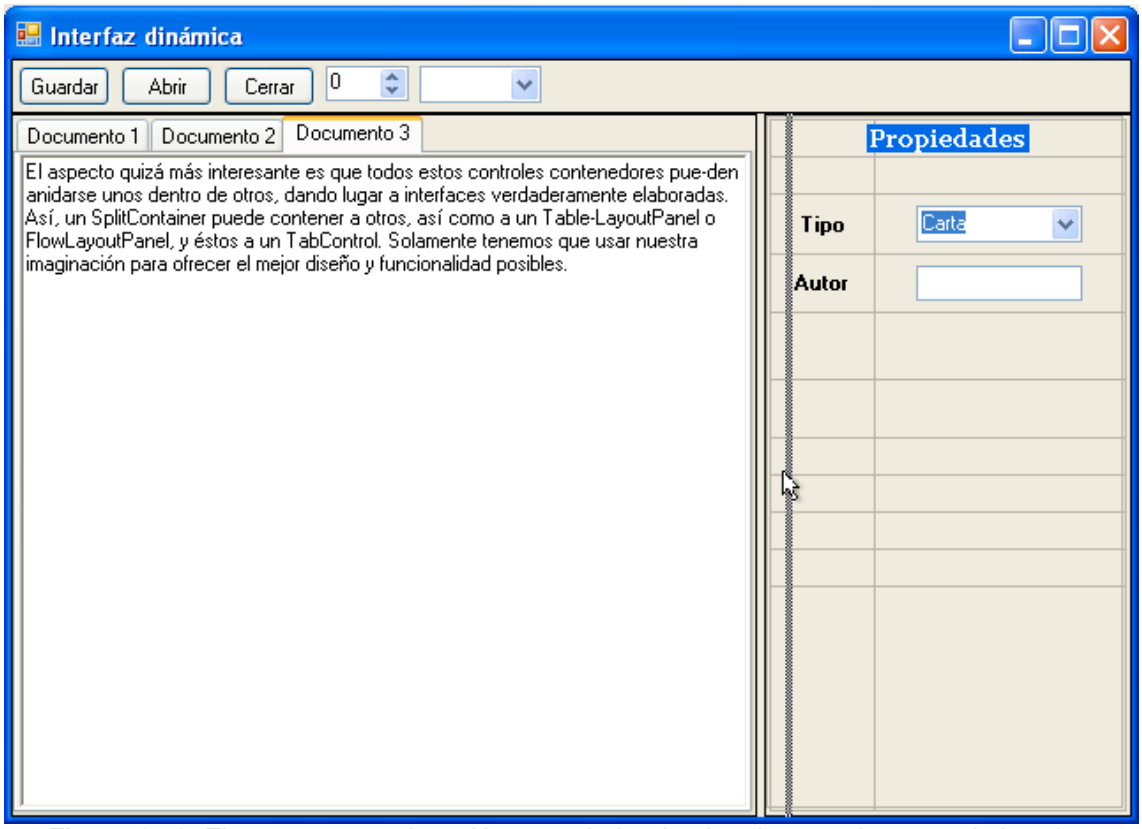

Figura 6.10. El programa en ejecución y tras haber hecho algunos ajustes en la interfaz.

6.3. Otros contenedores

Además de los que ya hemos conocido, en el grupo **Contenedores** del **Cuadro de herramientas** existen dos controles contenedores más: `GroupBox` y `Panel`. Éstos son mucho más sencillos que cualquiera de los anteriores, siendo su finalidad únicamente servir como contenedores de otros controles. Disponen de las mismas propiedades que la mayoría de los controles, incluidas las de diseño como `Anchor`, `Dock`, `Margin`, `Padding`, etc.

El control `GroupBox` muestra un título en su parte superior, mientras que `Panel` no dispone de dicho elemento. También se diferencian en el tipo de borde que usan como contorno.

Como contenedores simples pueden servirnos en distintas situaciones, por ejemplo para insertar más de un control en una celda de un `TableLayoutPanel` o sencillamente para diferenciar visualmente, a través de los títulos, bordes y colores de fondo, distintos apartados de una misma interfaz.

Resumen

Tras leer este capítulo, y poner en práctica los distintos controles y propiedades descritos, ya estamos en disposición de crear aplicaciones Windows con interfaces de usuario mucho más elaboradas y adaptables. Los contenedores que se han explicado en su mayor parte no tienen una funcionalidad final para el usuario: no pueden utilizarse para introducir un texto, seleccionar una opción o elegir un elemento de una lista. A cambio, sin embargo, se ocupan de distribuir los controles y ajustar sus posiciones y dimensiones de la manera más adecuada.

A los controles estándar, algunos de los cuales conocimos en el cuarto capítulo, y los contenedores, explicados en éste, hay que añadir otros controles Windows como las barras de estado y herramientas: `StatusStrip`, `ToolStrip` y `ToolStripContainer`. Con las nociones que ha adquirido no le resultará difícil identificar las propiedades clave y usarlos conjuntamente con los que ya conoce.

7. Cómo trabajar con bases de datos

Actualmente la mayoría de los proyectos de desarrollo de software que se ponen en marcha cuentan entre sus necesidades con el acceso a bases de datos, principalmente del tipo RDBMS, al convertirse éstas en los recipientes de información por excelencia en empresas y organizaciones de toda clase. Conectar con una base de datos, local (ejecutándose en el mismo equipo que el programa) o remota (ejecutándose en un servidor al que el cliente conecta mediante red), y ejecutar comandos de selección y modificación de datos, no hace mucho pasaba por ser una tarea prácticamente para expertos. Por suerte esto ha cambiado significativamente gracias a tecnologías como ADO.NET, los servicios de acceso a datos de la plataforma .NET, y entornos como el de *Visual C# 2005 Express Edition* que integran todo lo necesario.

ADO.NET es un avanzado modelo de acceso a bases de datos basado en el uso de una serie de proveedores, algunos de ellos específicos para un cierto RDBMS como SQL Server u Oracle y otros genéricos como ODBC, que actúan como intermediarios entre la base de datos y la aplicación. Aunque la plataforma .NET incorpora cuatro de estos proveedores: `SqlClient`, `OracleClient`, `OleDb` y `Odbc`, desde *Visual C# 2005 Express Edition* podremos usar únicamente una versión local del primero, adecuada para trabajar con *SQL Server 2005 Express Edition*, y el tercero para acceder a bases de datos Access.

El estudio de ADO.NET daría para un libro completo de extensión considerable, razón por la que en este capítulo nos limitaremos a analizar cómo podemos crear una base de datos desde el entorno de *Visual C# 2005 Express Edition*, editando su contenido si fuese necesario, y cómo acceder después a la información desde una aplicación Windows, usando para ello algunos de los componentes básicos de conexión a bases de datos.

7.1. El Explorador de bases de datos

A la hora de trabajar con bases de datos desde *Visual C# 2005 Express Edition*, una de las herramientas fundamentales es el **Explorador de bases de datos**. Esta ventana no está visible inicialmente, por lo que tendremos que recurrir al menú **Ver** para hacerla aparecer. En un principio solamente veremos en ella un nodo raíz, **Conexiones de datos**, y unos botones en la parte superior. Ese nodo tiene asociado un menú emergente con algunas opciones útiles.

La finalidad del **Explorador de bases de datos** es mantener las conexiones con bases de datos que hayamos definido, así como facilitar la definición de otras nuevas. Cada conexión se presenta como un nodo que es posible explorar, viendo las distintas categorías de objetos que contiene: tablas, vistas, procedimientos almacenados, etc.

También desde esta ventana es posible crear nuevas bases de datos, definir tablas, acceder a su contenido, etc.

Para poder escribir un programa que trabaja con una base de datos antes es necesario disponer de ese recurso. Podríamos usar Microsoft Access o las herramientas de administración de SQL Server para realizar ese trabajo, pero vamos a ver cómo es posible hacerlo todo desde el **Explorador de bases de datos**.

A fin de poder seguir los pasos que se indican en los puntos siguientes, creando una base de datos SQL Server y usándola después desde un programa, es imprescindible que antes haya instalado SQL Server 2005 Express Edition*. Es un producto que también puede obtenerse gratuitamente de Microsoft y le ofrece prácticamente las mismas características del producto completo, de tal forma que puede escribir aplicaciones que, con posterioridad, podrían ponerse en explotación frente a un servidor real.*

7.1.1. Creación de una base de datos

Comencemos por el principio, creando una base de datos con la que podamos trabajar. Para ello haremos clic en el último botón del **Explorador de bases de datos** a fin de definir una nueva conexión. Inmediatamente aparecerá el cuadro de diálogo que puede verse en la figura 7.1.

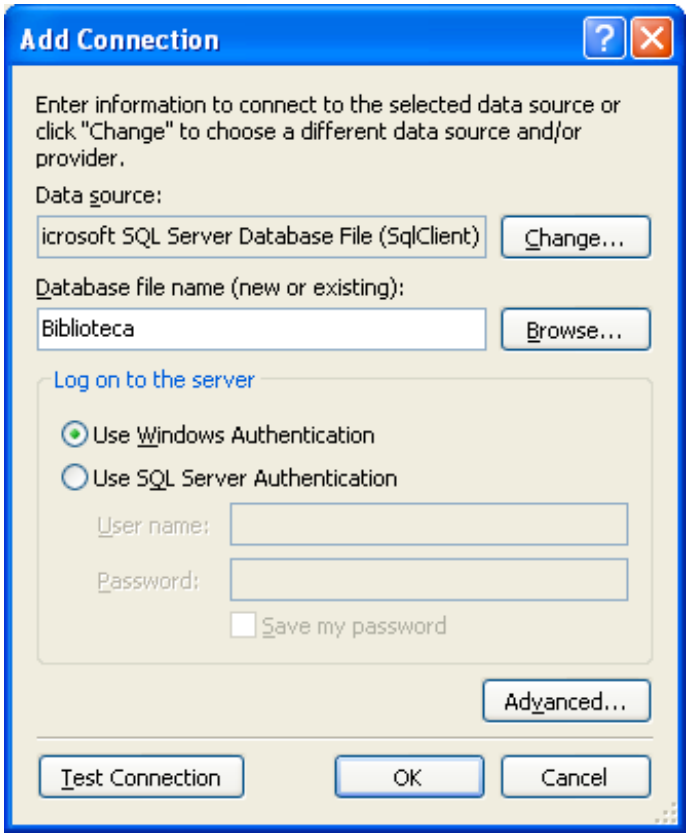

Figura 7.1. Definimos una nueva conexión a base de datos.

Por defecto se selecciona el proveedor ADO.NET para trabajar con SQL Server, pero pulsando el botón **Cambiar** se abrirá otro cuadro de diálogo en el que podremos elegir entre usar bases de datos Access o SQL Server. Éstas son las dos posibilidades en la edición *Express* de *Visual C# 2005*, pero en ediciones superiores puede trabajarse también, según se ha apuntado antes, con bases de datos Oracle y cualquier origen ODBC.

Nos quedamos con el valor por defecto e introducimos en el apartado **Nombre de archivo de base de datos** el nombre que deseamos dar a la base de datos. En este caso pretendemos crearla y, por tanto, podemos darle el nombre que prefiramos. Para agregar una conexión a una base de datos existente no tendríamos más que hacer clic en el botón **Examinar** y elegirla.

Para terminar, la pulsación del botón **Aceptar** procederá a la creación de la base de datos. El asistente nos indicará que la base de datos no existe y nos preguntará si deseamos crearla, a lo que responderemos afirmativamente y, si todo va bien, ésta aparecerá en el **Explorador de bases de datos**.

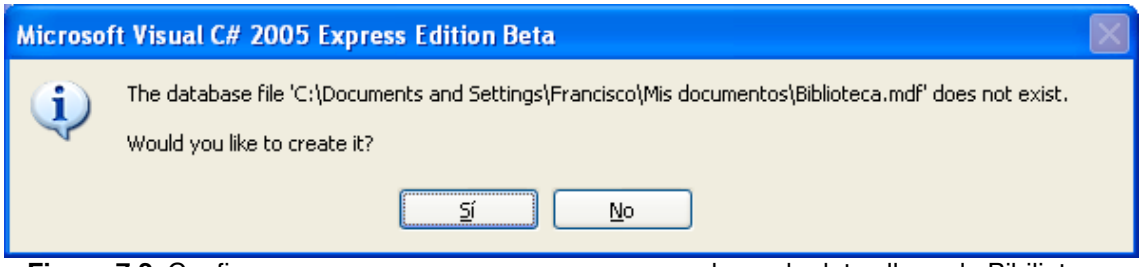

Figura 7.2. Confirmamos que queremos crear una nueva base de datos llamada Bibilioteca.

Otra forma de crear la base de datos, que en principio estará vacía, consiste en abrir el menú contextual del proyecto actual, en el **Explorador de soluciones**, elegir la opción **Agregar>Nuevo elemento** y finalmente seleccionar el elemento **Base de datos SQL** del cuadro de diálogo, introduciendo el nombre de la base en el recuadro que hay en la parte inferior. La nueva base de datos se añadirá como un elemento más del proyecto, y se distribuirá junto con él. Haciendo doble clic sobre ese elemento conseguiremos abrir la conexión en el **Explorador de bases de datos**, como si hubiésemos definido una nueva conexión.

7.1.2. Definición de tablas

Definida una conexión con una base de datos, ya sea una existente o que acabemos de crear, desde el **Explorador de bases de datos** podemos examinar las distintas categorías de objetos que contenga. En la carpeta **Tablas** (véase la figura 7.3 en la página siguiente) se encontrarán las tablas de la base de datos, recipientes últimos de la información útil para nosotros de la base de datos. Dicha carpeta, como la mayoría de las demás existentes, estarán vacías en un principio puesto que trabajamos con una base de datos recién creada.

Sirviéndonos de las opciones del menú emergente asociado a cada carpeta: **Tablas**, **Vistas**, **Procedimientos almacenados**, **Funciones**, etc., podremos realizar distintas operaciones, entre ellas la creación de nuevos objetos de la categoría elegida. Nosotros vamos a centrarnos en el diseño de una nueva tabla que nos permita almacenar información relativa a libros, con una estructura muy sencilla.

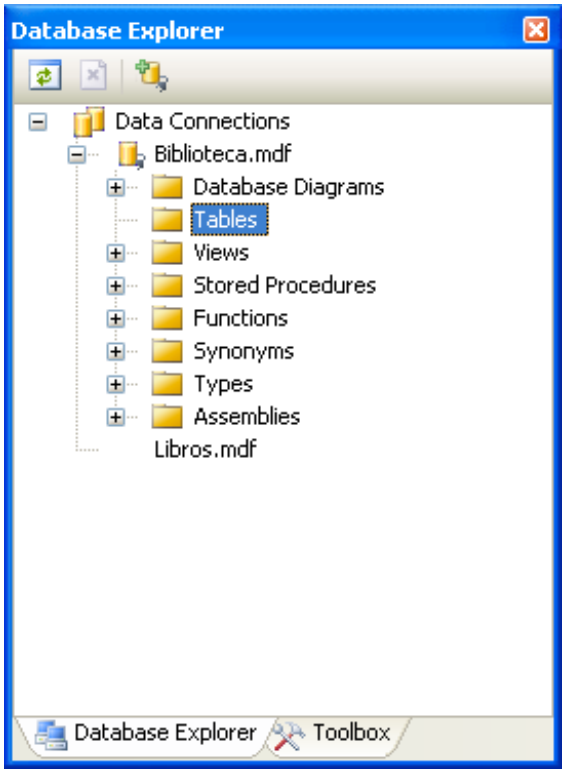

Figura 7.3. Los objetos de la base de datos se organizan en distintas categorías y carpetas.

Al elegir la opción **Agregar nueva tabla** del menú emergente de la carpeta **Tablas** se abrirá un diseñador específico en el zona central del entorno, mostrando dos áreas bien diferenciadas: en la parte superior la lista de columnas con que contará la tabla y en la inferior las propiedades de la columna elegida en cada momento en dicha lista.

El proceso de creación básicamente se reduce a ir introduciendo el nombre que se desea dar a cada columna, elegir el tipo de información que podrá contener y, finalmente, configurar cualquier otra propiedad que fuese precisa. Como se aprecia en la figura 7.4 (en la página siguiente), el diseñador dispone de un menú contextual que facilita el establecimiento de la clave primaria de la tabla, la inserción y borrado de columnas, definición de relaciones e índices, restricciones, etc.

Nuestra tabla, como puede verse en la citada figura 7.4, se compondrá de tres columnas: `Codigo`, `ISBN` y `Titulo`. La primera será de tipo `int` y actuará como columna identidad (cambiaremos de **No** a **Sí** el apartado **Especificación de identidad** en las propiedades de columna), lo cual significa que irá tomando automáticamente valores secuenciales y únicos. Esta columna será la clave principal de la tabla. Las otras dos contendrán secuencias de caracteres con una longitud máxima concreta: 13 en el caso de `ISBN` y 50 en el de `Titulo`.

Finalizada la definición de la tabla, al cerrar el diseñador aparecerá un cuadro de diálogo preguntándonos si deseamos guardar esta nueva tabla. Responderemos afirmativamente y facilitaremos un nombre para la nueva tabla, por ejemplo `Libros`. En este momento el **Explorador de bases de datos** debe mostrar en la carpeta **Tablas** la tabla que acabamos de crear. Si la desplegamos veremos aparecer las columnas definidas en ella.

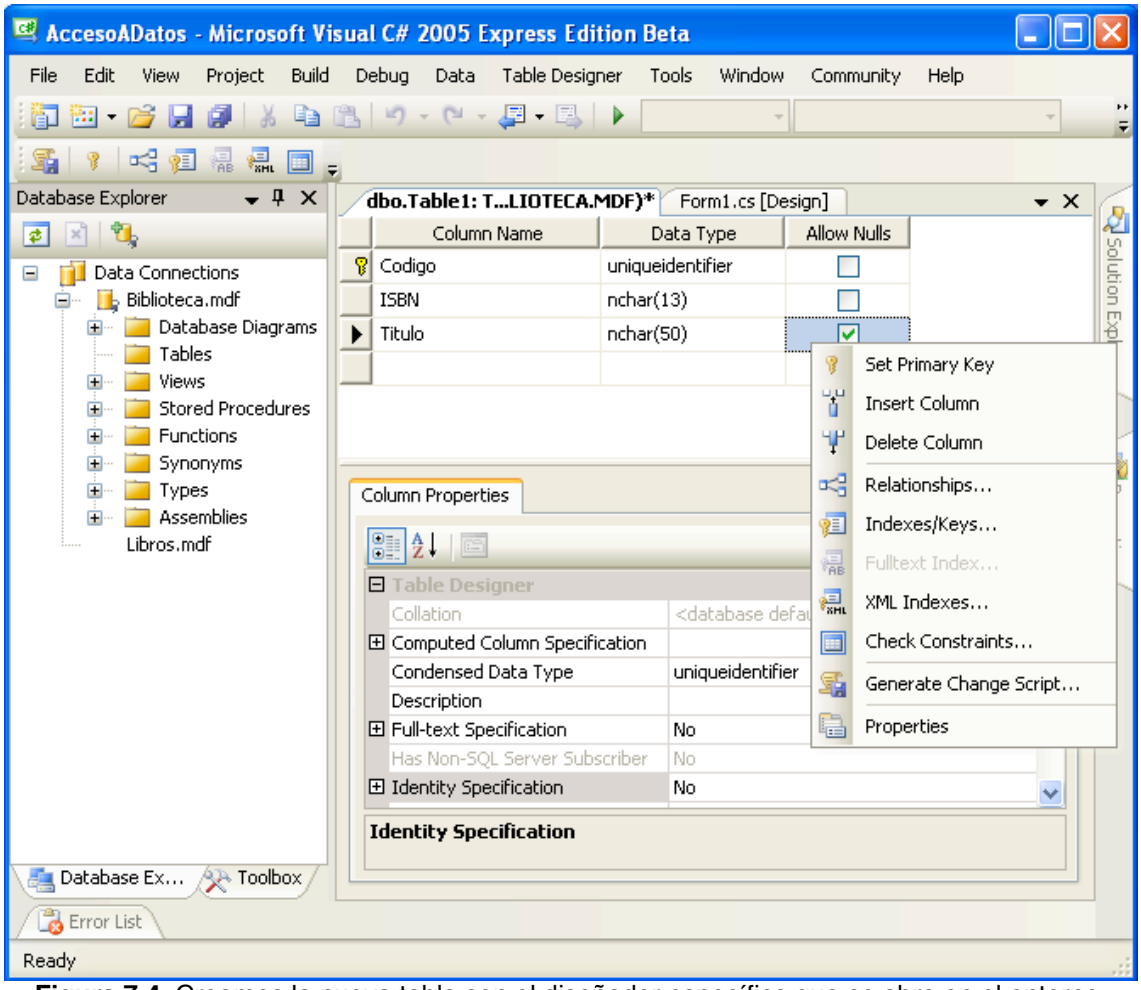

Figura 7.4. Creamos la nueva tabla con el diseñador específico que se abre en el entorno.

7.1.3. Edición del contenido de una tabla

El **Explorador de bases de datos** nos será útil no solamente para examinar la estructura de los objetos contenidos en la base de datos, o agregar otros nuevos, sino también para acceder a la información propiamente dicha: los datos alojados en las filas y columnas de las tablas. Esto nos permitirá efectuar prácticamente cualquier operación de edición sobre las tablas: inserción de nuevas filas, modificación de las existentes, borrado de una o más filas, agrupación, filtrados, etc.

Para abrir una tabla y acceder a su contenido usaremos la opción **Mostrar datos de tabla** del menú emergente asociado a la tabla. Esto abrirá en el área central del entorno un editor, en forma de cuadrícula, mostrando una barra de navegación en la zona inferior y una paleta de herramientas específica en la parte superior.

En principio, puesto que tenemos una base de datos nueva y con una tabla recién creada, poco podemos hacer más que añadir nuevas filas, introduciendo el ISBN y título de uno o más libros. Podemos desplazarnos libremente por la cuadrícula para elegir cualquiera de las filas y actuar sobre ella. Con los primeros botones de la barra de herramientas, contando de izquierda a derecha, es posible acceder a paneles

adicionales para configurar la selección de datos o editar directamente las sentencias SQL que se desean ejecutar.

Mientras manipulamos el contenido de la tabla desde el entorno de Visual C# 2005 Express Edition *estaremos realmente trabajando con el servidor de datos, SQL Server 2005, aplicación que se encargará de almacenar y recuperar la información, asegurar que se cumplen las restricciones que pudiera haber establecidas, etc.*

La figura 7.5 muestra un momento de la edición de la tabla que habíamos creado anteriormente, tras haber añadido algunas filas de datos. Al cerrar la página de edición todos los cambios permanecerán en la base de datos por lo que, en la práctica, es como si estuviésemos usando una interfaz genérica para operar sobre bases de datos, al estilo de Microsoft Access.

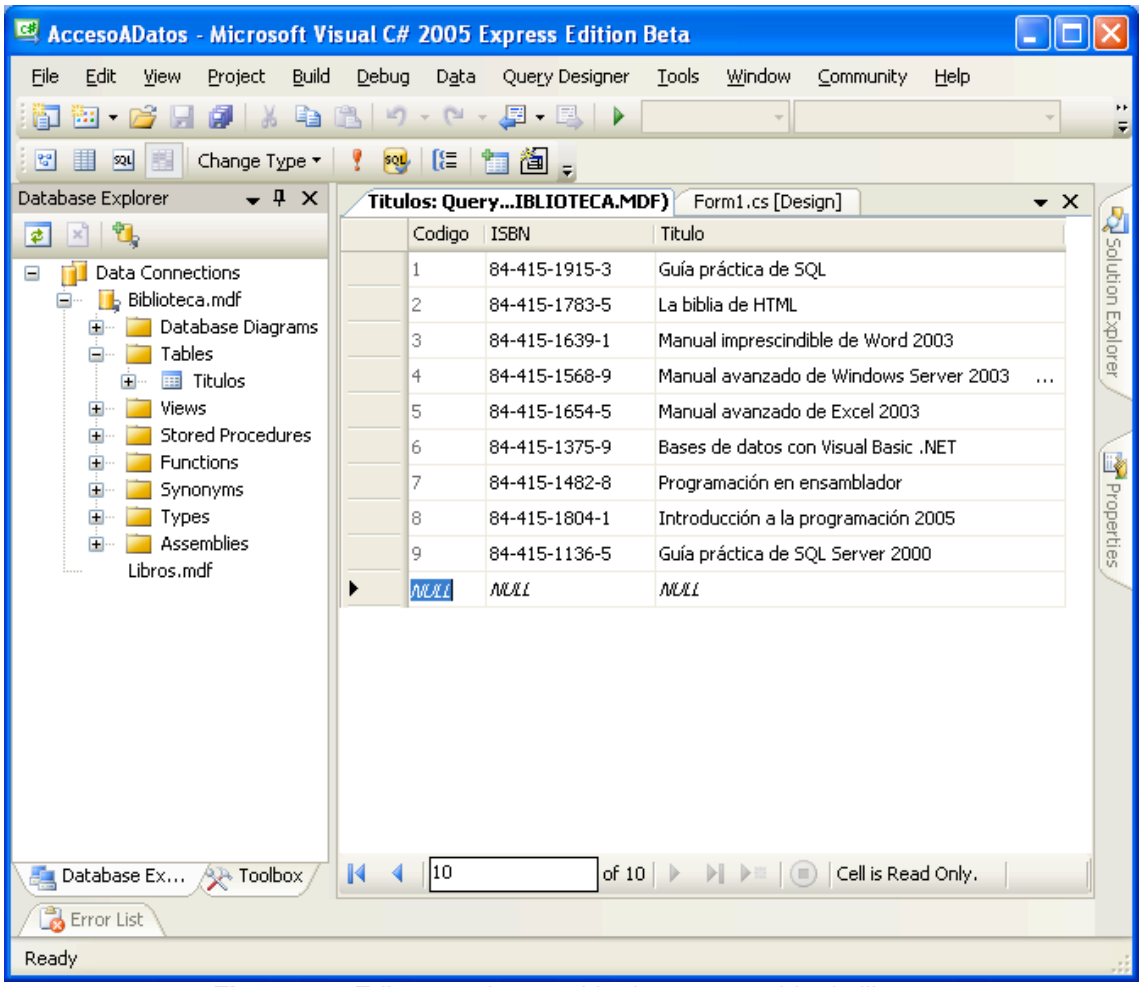

Figura 7.5. Editamos el contenido de nuestra tabla de libros.

Siguiendo el mismo procedimiento que se ha descrito podríamos definir la estructura de nuevas tablas, establecer relaciones entre ellas, diseñar diagramas, consultas, procedimientos almacenados, etc. Solamente tenemos que dedicar un rato a recorrer las opciones presentes en cada una de las carpetas y objetos, comprobando la funcionalidad de cada editor y diseñador.

7.2. La ventana de orígenes de datos

El **Explorador de bases de datos** es una herramienta que, como acabamos de ver, facilita la conexión a una base de datos desde el propio entorno de desarrollo, permitiendo operar sobre ella sin necesidad de recurrir a las utilidades específicas que ofrece el RDBMS. De esta manera podemos seguir trabajando en el entorno al que estamos acostumbrados incluso a la hora de realizar tareas de DBA, como son la definición de una estructura de base de datos.

De cara al desarrollo de aplicaciones, el aspecto más destacable del **Explorador de bases de datos** es el hecho de que se encarga de definir las cadenas de conexión a las bases de datos, identificando para ello el tipo de RDBMS, el equipo en que se ejecuta, el tipo de autenticación que se usará y el nombre de la base de datos. Estas cadenas de conexión las aprovecharemos posteriormente para definir uno o más orígenes de datos.

Un *origen de datos* es un objeto de la aplicación que representa un cierto conjunto de datos que residen en algún lugar, generalmente en un RDBMS, facilitando el acceso a ellos con fines de visualización y edición local en el programa. Por defecto la ventana **Orígenes de datos**, que será en la que vayamos alojando los orígenes de datos precisos para cada aplicación, no está visible. Para mostrarla no tenemos más que usar la opción **Datos>Mostrar orígenes de datos**. Nos encontraremos con una ventana inicialmente vacía y con un vínculo **Agregar nuevo origen de datos** que nos invita a definir un nuevo origen de datos.

7.2.1. Definición de un nuevo origen de datos

Asumiendo que queremos crear una aplicación Windows que conecte con la base de datos que hemos creado en los puntos anteriores, mostrando el contenido de su única tabla y permitiendo las operaciones más habituales, lo primero que tendremos que hacer será definir un origen de datos. Podemos hacerlo usando el enlace que aparece en la parte central de la ventana **Orígenes de datos** o bien pulsando el primero de los botones que hay en la parte superior.

En cualquier caso se pondrá en marcha el **Asistente para configuración de orígenes de datos**, un asistente en varios pasos que se encargará de crear los objetos que sean necesarios y agregarlos a nuestro proyecto. Además el nuevo origen de datos, con sus tablas, columnas, etc., aparecerá en la propia ventana **Orígenes de datos**, desde donde podremos arrastrarlos hasta el interior de un formulario.

La primera página del asistente nos permite elegir la procedencia de la información que necesitamos en nuestro programa: una base de datos, un servicio Web u otro objeto. Dejamos seleccionada la primera opción, **Base de datos**, y pulsamos el botón **Siguiente** para avanzar al paso siguiente.

A continuación elegiremos la conexión a usar para acceder a la base de datos. En la ventana aparece una lista (véase la figura 7.6, en la página siguiente) con las conexiones predefinidas en el **Explorador de bases de datos**, de tal forma que podemos elegir cualquiera de ellas para usarla. También podemos recurrir al botón **Nueva conexión** para definir una conexión en este mismo momento. Asumiendo que

hemos seguido los pasos de puntos previos, y que por tanto en el **Explorador de bases de datos** tenemos la base de datos Biblioteca creada con anterioridad, elegiremos esa conexión de la lista y pulsaremos de nuevo en **Siguiente**.

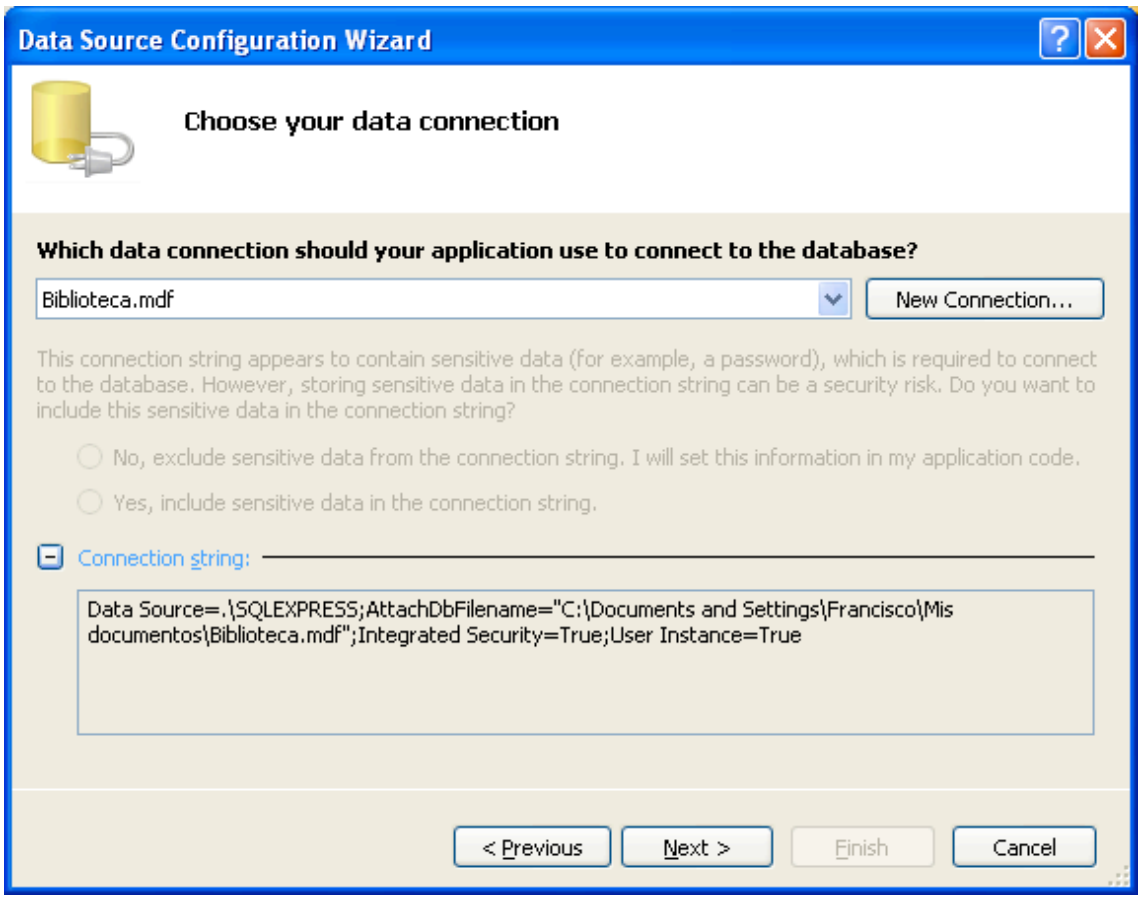

Figura 7.6. Seleccionamos la conexión que deseamos utilizar para crear el origen de datos.

A partir de la conexión elegida el asistente generará una cadena de conexión, puede verla en la parte inferior de la misma figura 7.6. En ella se indica que el servidor de datos es `SQLEXPRESS` (*SQL Server 2005 Express Edition*), el archivo de datos Biblioteca.mdf y que se utilizará la seguridad integrada basada en autenticación Windows. Esta cadena de conexión puede guardarse en la aplicación, dándole un nombre, facilitando así su reutilización para, por ejemplo, acceder a otras tablas, vistas o procedimientos de la misma base de datos. Por eso en el paso siguiente el asistente nos preguntará si deseamos guardar la cadena de conexión, proponiendo un nombre por defecto para la misma. Aceptamos los parámetros por defecto y pulsamos una vez más el botón **Siguiente** para continuar.

En este momento el asistente ya sabe la base de datos que pretendemos utilizar desde nuestra aplicación, pero ésta puede contener multitud de objetos y, seguramente, no los necesitaremos todos. Por ello en el paso siguiente nos propone que elijamos los objetos de la base de datos que queremos tener en la aplicación, a fin de generar un conjunto de datos a partir de ellos. Este conjunto de datos será una clase derivada de `System.Data.DataSet`, el conjunto de datos genérico de ADO.NET, en la que se definirán propiedades a medida para facilitar el acceso a cada tabla, vista, columna, etc.

Dado que nosotros queremos usar la única tabla que existe en la base de datos, nos limitamos a desplegar el nodo **Tablas** y marcar la tabla, como se ha hecho en la figura 7.7. Opcionalmente podemos, en la parte inferior de la ventana, modificar el nombre que se dará a la clase derivada de `DataSet`.

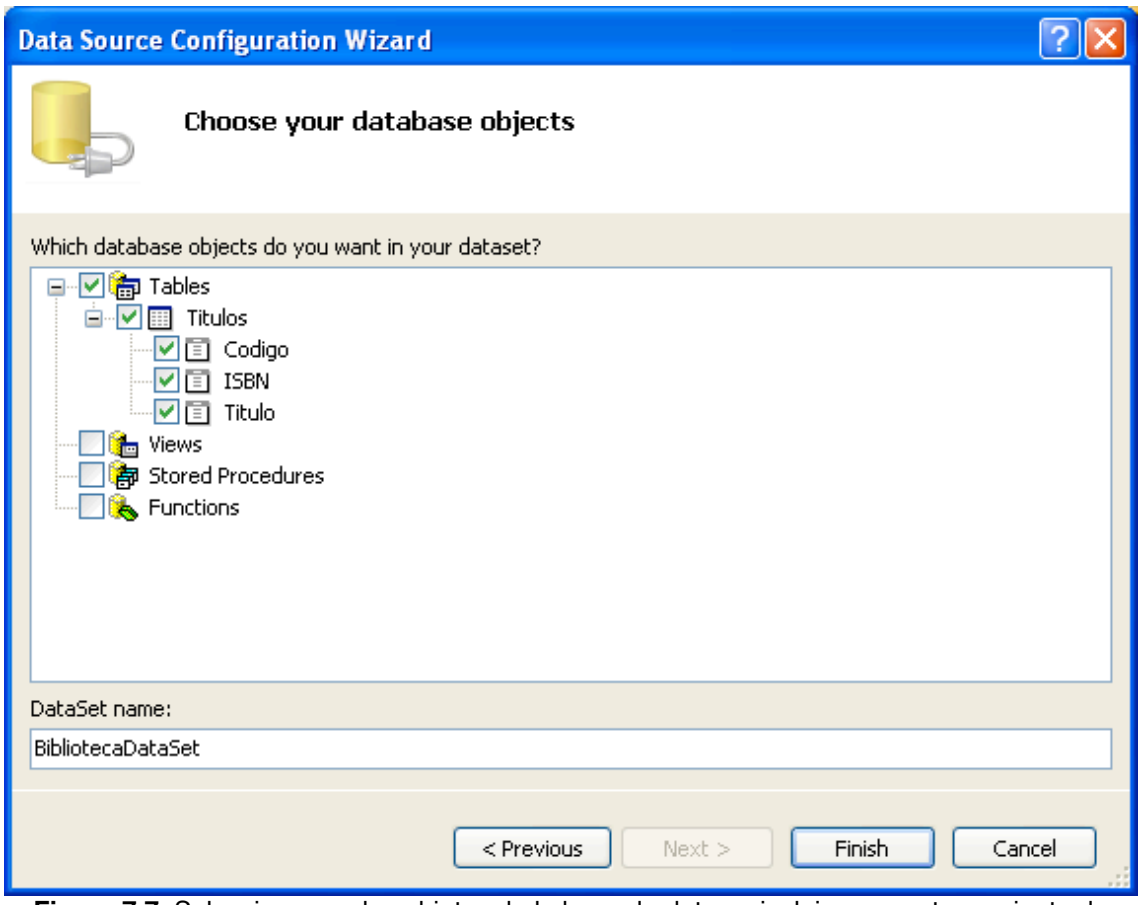

Figura 7.7. Seleccionamos los objetos de la base de datos a incluir en nuestro conjunto de datos.

Éste es el último paso del asistente. Hacemos clic en el botón **Finalizar** y se procede a crear el conjunto de datos, agregando al proyecto un esquema XSD con su estructura y un módulo de código asociado con la clase derivada de `DataSet`. Ésta aparece también en la ventana **Orígenes de datos** (véase la figura 7.8).

7.2.2. El diseñador de conjuntos de datos

Los conjuntos de datos generados por el asistente que acabamos de utilizar no son elementos estáticos, sino que podemos adaptarlos durante el desarrollo de la aplicación si fuese preciso. Con este fin recurriremos básicamente a dos herramientas: el propio asistente, que podemos volver a abrir mediante el botón **Configurar DataSet con el asistente** de la ventana **Orígenes de datos** (el tercero contando de izquierda a derecha), o bien usando el diseñador de conjuntos de datos. Éste se abre con el botón **Editar DataSet con el Diseñador** (segundo de izquierda a derecha) de la misma ventana.

*El diseñador de conjuntos de datos puede abrirse también haciendo doble clic sobre el esquema XSD que aparece en el **Explorador de soluciones**, sin necesidad de recurrir a la ventana **Orígenes de datos**.*

En nuestro caso el diseñador mostrará un recuadro conteniendo la tabla `Titulos`, un objeto de tipo `System.Data.DataTable`, y un adaptador de datos asociado, un objeto `TableAdapter`. El adaptador de datos es un intermediario que se ocupa de recuperar los datos desde el RDBMS e introducirlos en el conjunto de datos del programa, así como de devolver los cambios que éste efectúe a la información de vuelta al servidor de datos.

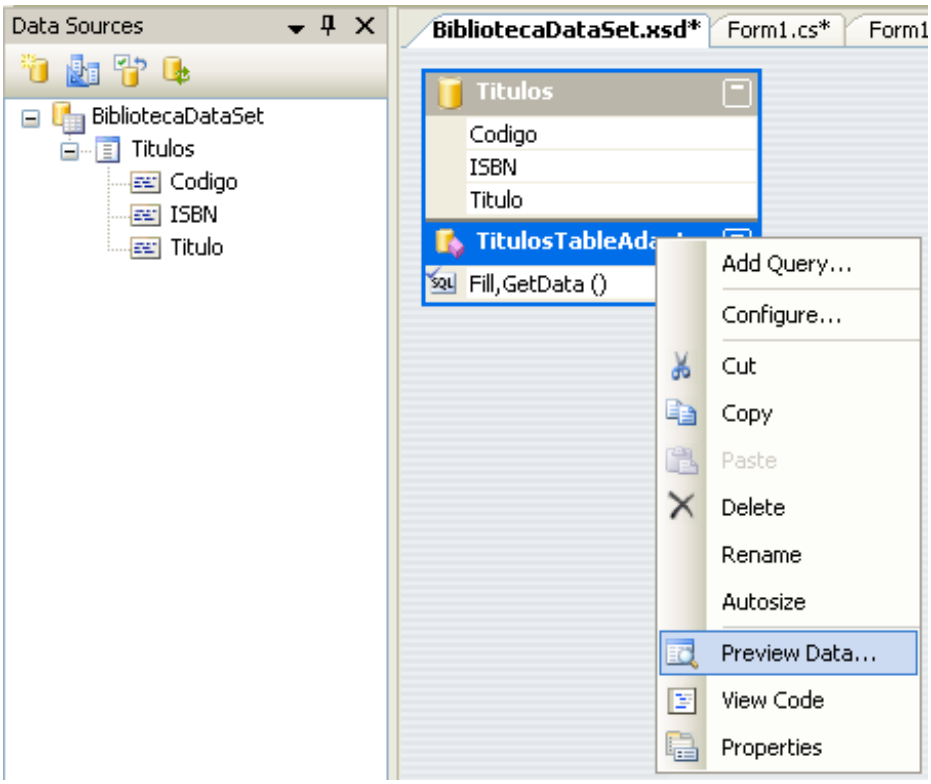

Figura 7.8. El diseñador de conjuntos de datos con nuestra tabla.

Desde este diseñador podemos modificar la tabla, agregando y eliminando columnas, así como configurar el adaptador de datos y obtener una vista previa de la información que recuperará del RDBMS. Seleccionando las columnas veremos aparecer en la lista **Propiedades** los atributos de cada una, pudiendo ajustar el tipo, nombre que tendrá en el programa (independiente del nombre original en la base de datos), valor por defecto, etc.

En el **Cuadro de herramientas** aparecerán los objetos que podemos añadir para crear nuevas tablas de datos, relaciones, consultas y adaptadores. Cada uno de estos objetos cuenta con opciones o asistentes que facilitan el proceso de configuración por lo que, en la práctica, nuestro trabajo se reducirá en la mayoría de los casos a elegir entre varias posibilidades de una determinada lista para adecuar la información obtenida.

7.2.3. Asociar elementos de interfaz a tablas y columnas

La última tarea para la que usaremos la ventana **Orígenes de datos**, en el punto siguiente, será la confección de interfaces de usuario con controles conectados a las columnas procedentes de la base de datos. Antes, sin embargo, podemos modificar la configuración por defecto que establece la asociación entre cada tipo de columna y el control que se empleará en la interfaz.

Al elegir una tabla o una columna en **Orígenes de datos** comprobaremos que existe un menú desplegable asociado. El de las tablas nos permite elegir entre dos configuraciones predeterminadas: **DataGridView** y **Detalles**, ofreciendo una opción **Personalizar** que permitirá crear una configuración a medida. En el caso de las columnas, según se aprecia en la figura 7.9, en la lista aparecen los controles que podemos asociar para la presentación de su contenido. Tendremos más o menos controles dependiendo del tipo de dato que tenga la columna en origen.

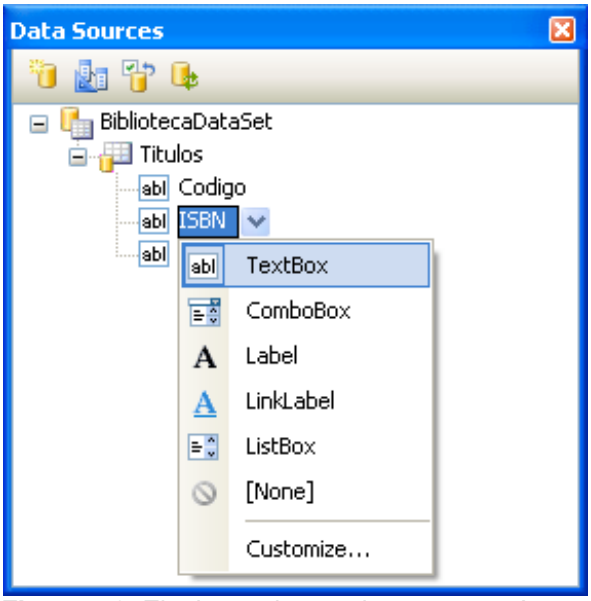

Figura 7.9. Elegimos el control que se usará para mostrar los datos de cada una de las columnas al insertarlas en un formulario.

Usando estos menús desplegables podemos establecer, por ejemplo, que la columna `Codigo` se muestre en un control `Label` en lugar de un `TextBox`, ya que se trata de una columna que no puede ser modificada y, además, contiene un valor que va generándose automáticamente.

Mediante la opción **Personalizar** se accede a una ventana de opciones donde podemos elegir, por cada tipo de columna, los controles que es posible utilizar para su presentación/edición. Esos controles aparecerían, a partir de ese momento, como una opción más en la lista desplegable de las columnas que fuesen de dicho tipo. No se trata, por tanto, de una configuración específica para cada columna de cada tabla, sino que se establece por tipos de datos.

7.3. Una interfaz de usuario conectada a datos

Partiendo del origen de datos que tenemos preparado y configurado en la ventana **Orígenes de datos**, diseñar una interfaz de usuario que nos permita acceder a los datos, para examinarlos y editarlos, es una tarea realmente sencilla.

Las interfaces con controles conectados a datos suelen ser de uno de dos tipos posibles: basados en una cuadrícula, con filas y columnas, o bien ofreciendo lo que se conoce como vista de detalle. En el primer caso se recurre a un control `DataGridView`, mientras que en el segundo se emplean controles que ya conocemos como `Label`, `TextBox`, `ListBox`, etc.

El tipo usado por defecto es el primero. Por ello, si arrastramos la tabla desde **Orígenes de datos** hasta el formulario (como se ha hecho en la figura 7.10) se inserta un `DataGridView` conectado al `DataSet` y un control de navegación en la parte superior.

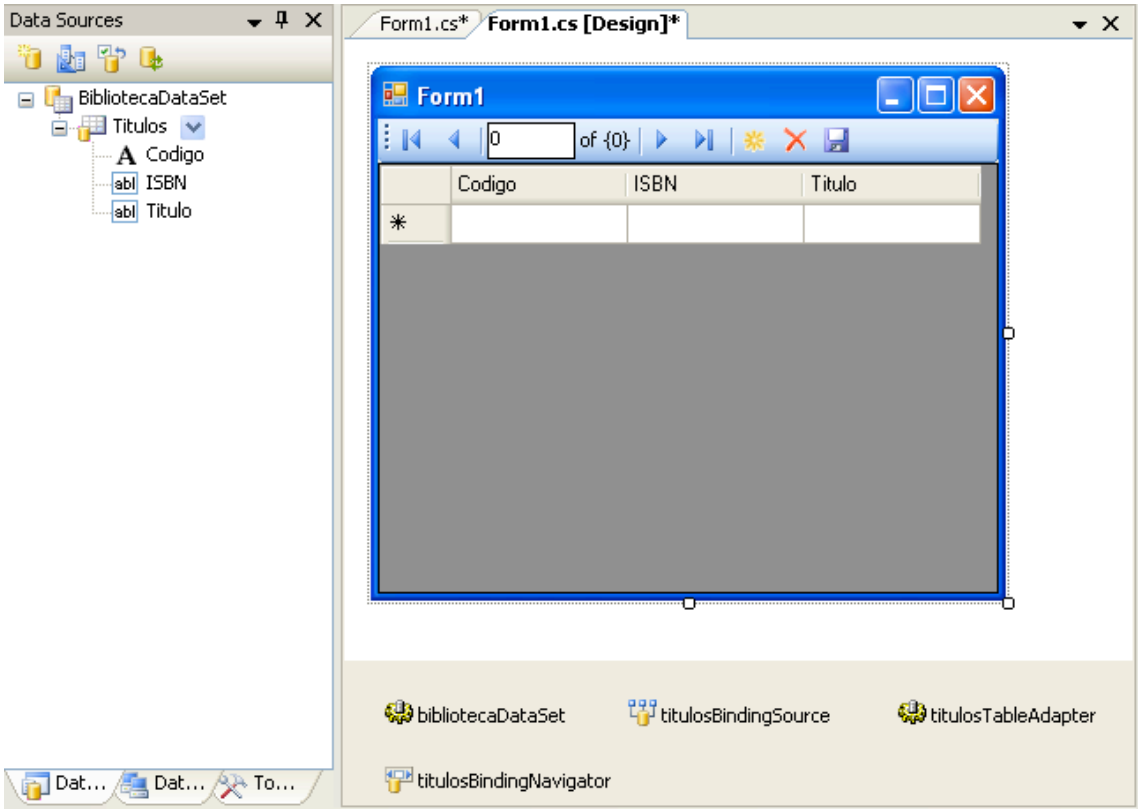

Figura 7.10. Una interfaz mostrando la tabla en forma de cuadrícula.

Si antes de arrastrar y soltar la tabla hasta el formulario desplegamos el menú asociado en **Orígenes de datos** y elegimos **Detalles**, en lugar de un `DataGridView` obtendremos un barra de navegación, la misma que aparece en la figura 7.10, y una serie de controles `Label` y `TextBox` representando a cada una de las columnas. En este caso la ventana mostraría una fila de datos solamente, mientras que el `DataGridView` visualiza de forma simultánea todas las filas y columnas.

Otra opción a nuestro alcance es colocar en el formulario de forma individual cada una de las columnas, arrastrándolas y soltándolas desde **Orígenes de datos** de una en una en lugar de como conjunto. El beneficio que obtenemos, frente a la inserción directa de los controles desde el **Cuadro de herramientas**, es que el diseñador se ocupa de establecer la propiedad `DataBindings` de cada control para vincularlo a su respectiva columna de la tabla de datos.

En cualquier caso, sigamos el procedimiento que sigamos, al final nos encontraremos con una aplicación completamente funcional que nos permite ver el contenido de la tabla en la base de datos, añadir nuevas filas, eliminar y cambiar las existentes y, finalmente, enviar los cambios al RDBMS. En la figura 7.11 puede verse en ejecución la interfaz basada en el control `DataGridView`.

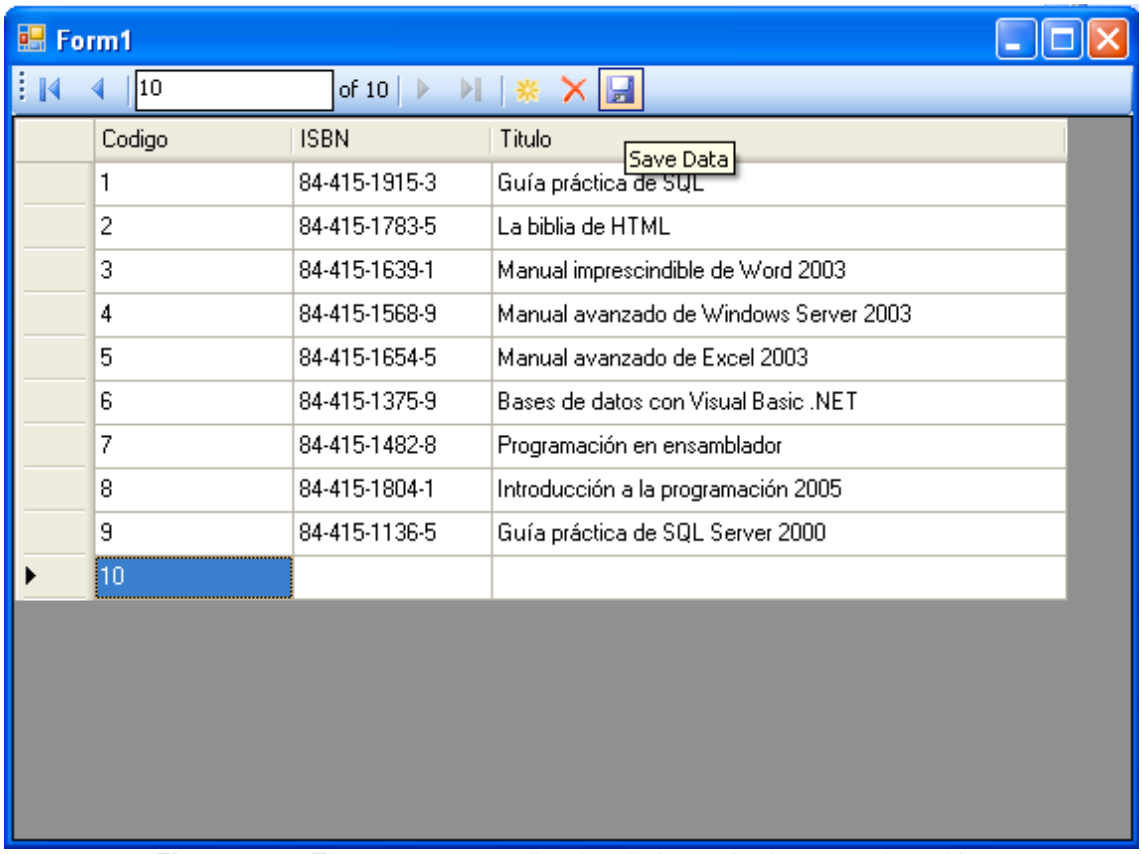

Figura 7.11. El programa accediendo a la base de datos en ejecución.

Resumen

Como hemos podido ver en este capítulo, *Visual C# 2005 Express Edition* incorpora todas las herramientas necesarias para conectar con bases de datos, definir conjuntos de datos y crear interfaces que facilitan el acceso a la información. Al final hemos obtenido una aplicación Windows que permite trabajar sobre una tabla de SQL Server sin haber escrito una sola línea de código, empleando únicamente elementos como el **Explorador de bases de datos** o la ventana **Orígenes de datos**, conjuntamente con los diseñadores que ya conocíamos.

Si examinamos el proyecto, sin embargo, encontraremos una importante cantidad de código tanto en el formulario, usando el adaptador de datos para recuperar información y enviar los cambios, como en los módulos asociados a cada `DataSet`. Ese código, generado automáticamente por los diseñadores y asistentes, es el que se encarga de todo el trabajo, de tal forma que no necesitamos conocer prácticamente nada sobre ADO.NET para construir programas que acceden a bases de datos.

En la práctica, sin embargo, siempre nos será de utilidad conocer los fundamentos del funcionamiento de clases como `DataSet`, `TableAdapter` o `DataTable`. Toda esta información podemos encontrarla en la documentación de referencia sobre ADO.NET, en la ayuda en línea del producto.

8. Uso de los *Starter Kits*

Una de las formas más rápidas de aprender a programar con un cierto lenguaje y usando unos servicios determinados consiste en examinar código escrito por otros, observando las técnicas empleadas, objetos a los que se recurre para cada tarea, etc. Con *Visual C# 2005 Express Edition* esta técnica la tenemos al alcance de la mano, ya que sus diseñadores y asistentes generan una importante cantidad de código que podemos estudiar.

Pero si bien las plantillas que hemos usado hasta ahora para crear aplicaciones, Windows o de consola, producen un bloque de código más o menos elaborado, lo cierto es que no dejan de ser unos sencillos esqueletos a los que tenemos que añadir el músculo y el nervio, en forma de controles y código. Todo lo contrario que las plantillas *Starter Kit* que aparecen en la ventana **Nuevo proyecto**, capaces de generar una aplicación completa y terminada.

Como se aprecia en la figura 8.1, son dos las plantillas de este tipo: **Starter Kit del protector de pantalla** y **Starter Kit de mi colección de películas**. El primero de ellos crea un proyecto que genera un ejecutable, operando como un protector de pantalla una vez instalado. El segundo produce una completa aplicación para mantener información sobre una colección de películas.

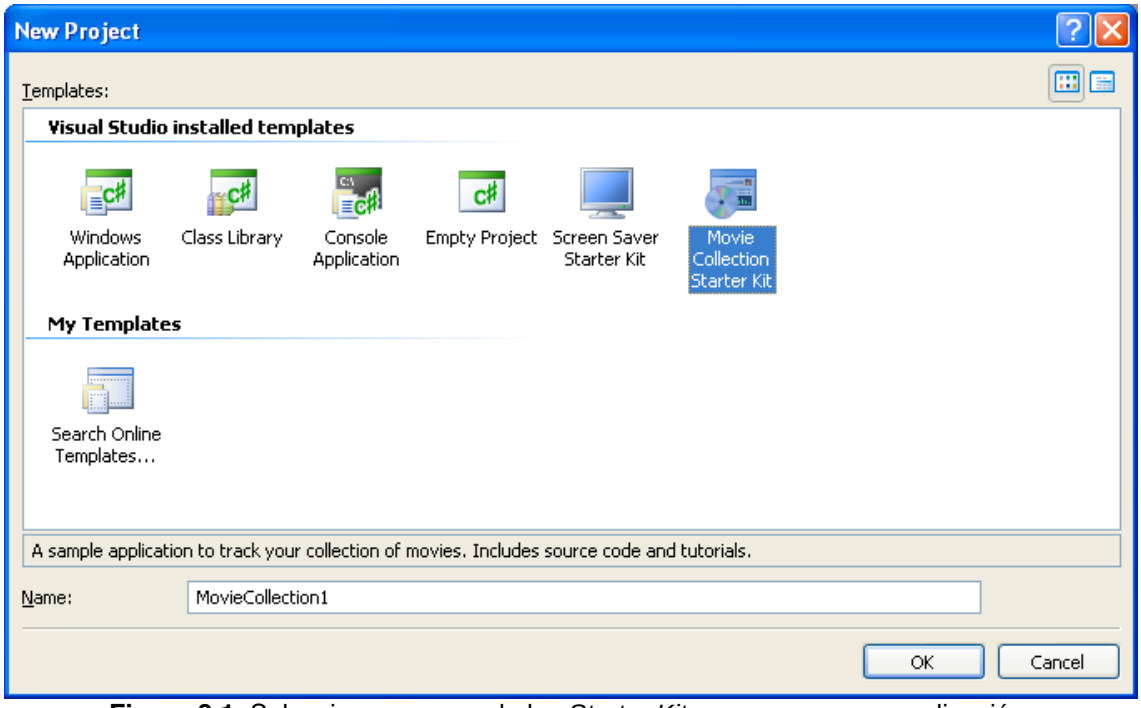

Figura 8.1. Seleccionamos uno de los *Starter Kit* para generar una aplicación.

8.1. Creación de un protector de pantalla

Un protector de pantalla es una aplicación que el sistema ejecuta tras un cierto tiempo de inactividad, no un programa de usuario final. De hecho, en cuanto el protector detecte la pulsación de una tecla, un botón del ratón o un movimiento de este dispositivo debe terminar de inmediato. Opcionalmente puede aportar un cuadro de diálogo de configuración que debería mostrar cuando se facilite el indicador `/c` en la línea de argumentos.

Por lo demás, no hay unas directrices estrictas de qué debe hacer y cómo ha de funcionar un protector de pantalla. La mayoría de ellos eliminan el contenido actual de la pantalla, ocultándolo con una ventana propia, y muestran un texto, una animación, imágenes, etc. No hay nada que impida, sin embargo, que el protector use el contenido actual de la pantalla (capturándolo) para generar algún efecto.

El *Starter Kit* que incorpora *Visual C# 2005 Express Edition* crea un salvapantallas que, aparte de mostrarnos la estructura general de una aplicación de este tipo, cuenta con dos características distintivas: recupera titulares RSS de un sitio remoto y los muestra sobreimpresos a unas imágenes que cambian aleatoriamente. Se hace uso, por tanto, de un amplio conjunto de servicios de la plataforma .NET, entre ellos los de comunicación a través de redes y GDI+.

8.1.1. Documentación del proyecto

Al usar el asistente para creación de salvapantallas lo primero que veremos aparecer será la página de documentación del proyecto. En ésta se explica qué hace el protector de pantalla, cómo debe compilarse el proyecto y efectuar la configuración para que Windows lo use como un salvapantallas más. También se proponen algunos cambios que puede efectuar al código para personalizar el resultado.

La segunda parte de la documentación describe cómo funciona el proyecto, indicando los módulos que lo forman y la tarea que acomete cada uno de ellos. Esto nos servirá para localizar rápidamente la parte que nos interese modificar.

Finalmente, en cuanto a la documentación se refiere, se describe la forma en que Windows se comunica con los salvapantallas, cómo se almacenan las opciones de comunicación, qué son XML y RSS, etc. En conjunto tenemos no solamente un salvapantallas, sino también una extensa documentación que nos permitirá profundizar en su funcionamiento, entenderlo y modificarlo según nos convenga.

*Podemos comprobar de inmediato el funcionamiento del savapantallas, sin llegar siquiera a instalarlo como tal siguiendo los pasos indicados en la documentación, sencillamente pulsando **F5** para ejecutarlo desde el propio entorno de desarrollo. Veremos cómo se oscurece la pantalla y en ella aparecen unas imágenes de fondo y una lista de noticias obtenidas de Microsoft. En cuando movamos el ratón, o pulsemos alguna tecla o botón, el salvapantallas se cerrará.*

8.1.2. Análisis de la línea de comandos

Los salvapantallas modifican su comportamiento según las opciones que se entreguen en la línea de comandos, al ser ejecutados. De esta manera Windows puede solicitar la visualización del cuadro de diálogo que configura el salvapantallas, la ejecución en una ventana en miniatura o la ejecución normal.

Al ejecutar un programa es posible facilitar una lista de parámetros en la propia línea de comandos, parámetros que en el caso de los programas C# son recibidos como una matriz de cadenas por el método `Main`, una técnica que no le resultará extraña si alguna vez ha programado en lenguaje C. En el módulo Program.cs del proyecto encontraremos dicho método, con el código mostrado a continuación:

```
static void Main(string[] args)
{
    if (args.Length > 0)
    {
        // Get the 2 character command line argument
        string arg = args[0].ToLower(
           CultureInfo.InvariantCulture).Trim().Substring(0, 2);
        switch (arg)
        {
            case "/c":
                // Show the options dialog
                ShowOptions();
                break;
            case "/p":
                // Don't do anything for preview
                break;
            case "/s":
                // Show screensaver form
                ShowScreenSaver();
                break;
            default:
                MessageBox.Show("Invalid command line argument :" +
                arg, "Invalid Command Line Argument",
                MessageBoxButtons.OK, MessageBoxIcon.Error);
                break;
        }
    }
    else
    {
        // If no arguments were passed in, show the screensaver
        ShowScreenSaver();
    }
}
```

El condicional `args.Length > 0` comprueba si se ha facilitado algún parámetro en la línea de comandos, ya que de no ser así se invoca directamente al método `ShowScreenSaver` para poner en marcha el salvapantallas. Si hay argumentos se procede a convertir a minúsculas y recuperar los dos primeros caracteres únicamente, tras lo cual se analiza si el parámetro es `/c`, `/p`, `/s` o ninguno de ellos para proceder en consecuencia.

Si quisiéramos que el programa aceptase opciones adicionales, éste sería el punto adecuado en el que tendríamos que insertar nuestro código. Bastaría con añadir más apartados al bloque `switch`.

8.1.3. El formulario del salvapantallas

En el **Explorador de soluciones** podemos ver que el proyecto cuenta con dos formularios: OptionsForm.cs y ScreenSaverForm.cs. Este último es el formulario en el que se ejecuta el salvapantallas al ponerse en marcha, donde está el código de mayor interés del programa.

Al abrir el formulario comprobaremos que está vacío, solamente un texto en su parte central indicando que la acción tiene lugar en los métodos `OnPaint` y `OnPaintBackground`. Debajo del formulario encontraremos un componente `Timer`, cuya finalidad es generar un evento periódico conforme al lapso indicado en su propiedad `Interval`. En este proyecto se usa para cambiar cada 10 segundos la imagen de fondo y noticia destacada.

Si revisamos las propiedades del formulario, en la ventana **Propiedades**, nos daremos cuenta de que `WindowState` contiene el valor `Maximized`. Esto implica que cuando el formulario se muestre ocupará automáticamente todo el espacio disponible en la pantalla. Asimismo también se ha cambiado la propiedad `ShowInTaskBar`, dándole el valor `False` para que el programa no muestre un botón en la barra de tareas como si fuese una aplicación corriente.

Al examinar el código de este formulario apreciaremos que cuenta con un proceso de inicialización relativamente complejo. Se recuperan una serie de imágenes de un directorio y se almacenan en una lista, dejándolas preparadas para usarlas como fondo. También se usa una clase definida en un módulo aparte para leer el canal RSS sobre C# de MSDN, obteniendo la lista de titulares.

Cada vez que se produce el evento que comunica que hay que dibujar el fondo de la ventana, que en este caso cubre toda la pantalla, se ejecuta el método `OnPaint-Background`. En éste se usa el método `DrawImage`, uno de los muchos de GDI+, para dibujar una de las imágenes recuperadas con anterioridad. De manera similar, en el método `OnPaint` se dibuja la lista de noticias obtenidas antes.

El evento generado por el control `Timer` se limita a invocar al método `Refresh` del formulario, solicitando una actualización de su contenido. En respuesta, el formulario ejecutará los citados métodos `OnPaintBackground` y `OnPaint`.

Finalmente podemos observar cómo los eventos de pulsación de teclas, botones del ratón y desplazamiento del ratón provocan el cierre del formulario. Al ser éste la ventana principal, su cierre pone fin a la aplicación.

8.1.4. El formulario de configuración

El segundo formulario con que cuenta el proyecto es mucho más estándar, siendo su finalidad permitir cambiar el URL del RSS desde el que se obtienen las noticias y el

directorio donde se buscan las imágenes. Estos parámetros se almacenan en la configuración de la aplicación y son utilizados posteriormente, desde el formulario del salvapantallas, para adecuar su funcionamiento.

En el diseño del formulario (véase la figura 8.2) se han usado dos controles `TableLayoutPanel` anidados, uno con tres filas y otro en la última fila del anterior con cuatro columnas, tres de las cuales están ocupadas por los botones. Es una muestra de cómo puede distribuirse una interfaz de usuario sin emplear posiciones ni dimensiones fijas, según se explicó en un capítulo previo.

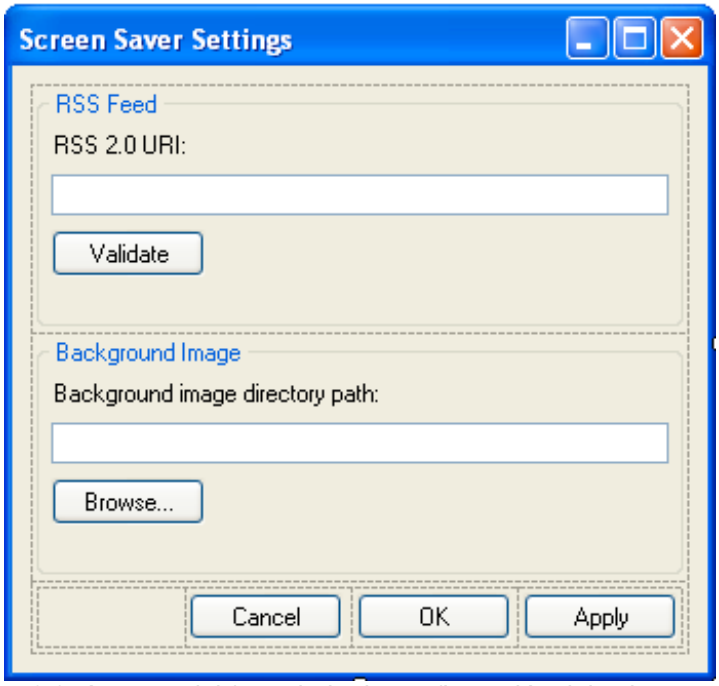

Figura 8.2. Aspecto del formulario de configuración del salvapantallas.

Al examinar el código lo único de interés especial lo encontramos en la recuperación de los valores actuales de los parámetros de configuración, en el constructor de la clase `OptionsForm`, y su posterior salvaguarda, en el método `ApplyChanges`. El objeto `Properties.Settings` representa las opciones de configuración disponibles en la aplicación, opciones que se definieron en su momento en la ventana de propiedades del proyecto.

Si quisiéramos agregar más opciones de configuración tendríamos que dar tres pasos:

- Definir el nombre y tipo de los parámetros en la página correspondiente de la ventana de propiedades del proyecto, accesible mediante la opción **Propiedades** del menú emergente que aparece en el **Explorador de soluciones**.

- Adecuar el formulario para dar cabida a esos nuevos parámetros, por ejemplo añadiendo otros recuadro de texto y botones.

- Modificar el código añadiendo las sentencias necesarias para recuperar el valor actual de los nuevos parámetros y guardarlo cuando proceda.

8.1.5. Otros elementos del proyecto

Además de los formularios y el código de puesta en marcha, en este proyecto encontramos también una serie de clases definidas en las carpetas **Rss** y **UI**. La finalidad de las primeras es recuperar el canal RSS desde el URI indicado, abriendo para ello una conexión con el servidor remoto y empleando un objeto `XmlTextReader` para ir interpretando el documento XML. En el método `FromUri` de la clase `RssFeed` puede verse la sencillez con la que se efectúa este proceso, sin necesidad de recurrir a servicios de bajo nivel de comunicación TCP/IP ni nada parecido.

En cuanto a los elementos de la carpeta **UI**, definen una clase llamada `ItemListView` capaz de representar una lista de elementos de cualquier tipo siempre que implementen una serie de miembros, establecidos en la interfaz `IItem`. En dicha clase se usa GDI+ para dibujar la lista (véase la figura 8.3) y generar efectos de fundido. Es una buena referencia si queremos aprender a usar GDI+.

Figura 8.3. El salvapantallas generado por el *Starter Kit* en funcionamiento.

8.2. Mantener datos sobre una colección de películas

Si el *Starter Kit* anterior puede servirnos para aprender a crear un salvapantallas y, de paso, conocer algunas de las clases de GDI+ necesarias para dibujar en pantalla o las que permiten recuperar un documento alojado en un servidor remoto, el segundo produce una aplicación en la que lo más destacable es el uso de una base de datos SQL Server 2005 para guardar información sobre una colección de películas, así como el diseño de unos controles a medida para componer la interfaz de usuario.

Al usar este *Starter Kit* nos encontraremos inicialmente con la página de documentación del proyecto, similar a la descrita en el caso previo. En esta página podemos ver que existe una versión mejorada de esta aplicación que podemos descargar de Internet, siendo la diferencia que dicha versión es capaz de conectarse a la red para recuperar información sobre la películas que solicitemos.

En el **Explorador de soluciones** veremos que el proyecto incorpora una base de datos: DVDCollectionDatabase, así como un conjunto de datos: `DVDCollectionDataSet`. Éste puede ser abierto en el correspondiente diseñador para examinar las columnas existentes y, si quisiéramos, efectuar los cambios pertinentes.

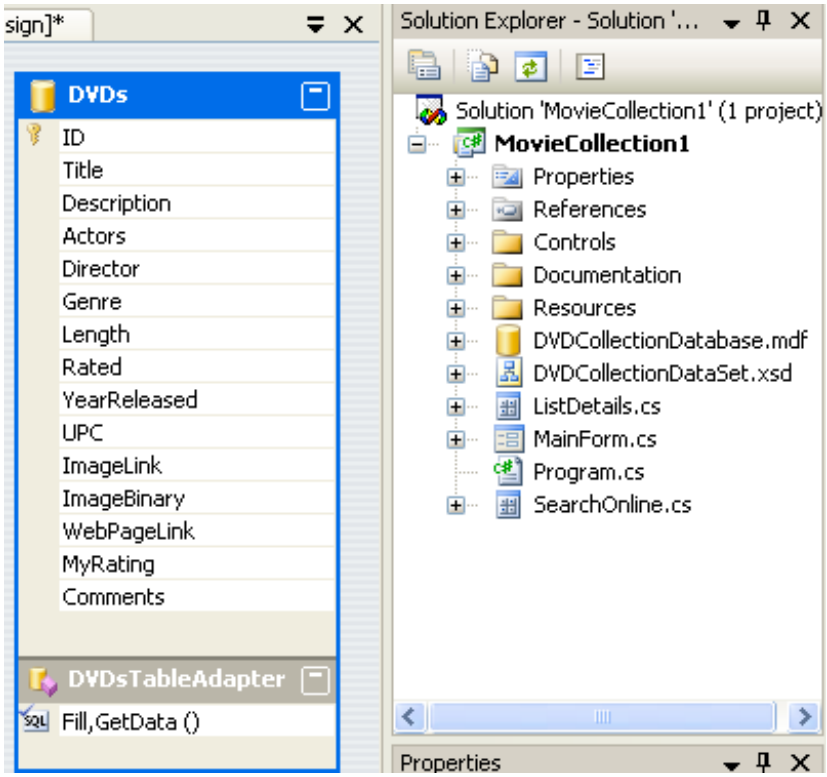

Figura 8.4. Elementos del proyecto y columnas de la base de datos.

A pesar de que la aplicación cuenta con vistas diferentes, concretamente una de detalle y otra de búsqueda, en el proyecto existe un único formulario: `MainForm`. Lo que hace éste es mostrar en su interior uno de los dos controles diseñados a medida para el programa, teniendo además la particularidad de que se minimiza quedando en la barra de taras como un icono, para lo cual se recurre al componente `NotifyIcon`.

Mediante las propiedades del componente `NotifyIcon` podemos mostrar en la barra de tareas cualquier icono, así como un texto explicativo y un menú emergente. Tan solo hay que asignar esos elementos a las propiedades correspondientes. Además el control genera eventos cuando detecta que el puntero del ratón pasa sobre el icono o se hace clic sobre él, lo cual permite implementar el comportamiento que nos interese.

Los dos controles de usuario, a la postre, son los que contienen todo el código de acceso a la base de datos, recuperación y almacenamiento de información. Como ya sabemos, ese código se genera automáticamente mediante operaciones de arrastrar y soltar. Las interfaces de usuario de los controles son bastante llamativas, pero lo único especial que tienen es que la propiedad `BackgroundImage` de distintos paneles y botones hace referencia a una serie de imágenes incorporadas al proyecto.

Resumen

Mediante los *Starter Kit* que incorpora *Visual C# 2005 Express Edition* obtenemos aplicaciones casi completas, preparadas para ser usadas, que podemos utilizar como punto de partida para, modificándolas o tomándolas como plantillas, crear las nuestras propias. Lo más importante, quizá, sea el hecho de que podemos aprender a usar los distintos servicios de la plataforma .NET y determinadas técnicas examinando los proyectos que generan estas opciones.

Además de los integrados en el entorno, a través del menú **Comunidad** podemos efectuar una búsqueda para encontrar otros *Starter Kits* que pudiese haber disponibles en Internet, ya sean de Microsoft o creados por terceros. En ese mismo menú encontraremos opciones que pueden sernos muy útiles durante nuestro aprendizaje, ya que nos permiten buscar soluciones de otros programadores a problemas que ahora se nos presentan a nosotros.

Índice alfabético