APENDICE A

EL PROPÓSITO DE ESTE APÉNDICE ES explicar, o al menos mencionar, las novedades en el acceso a datos que incluye el nuevo Visual Studio 2005. Hay pocas novedades en ADO.NET de la versión 2.0 de la plataforma… siempre que nos limitemos a la biblioteca de clases. En lo que respecta a la técnica de enlace de datos (*data binding*) hay un poco más de animación: hay una nueva clase, *BindingSource*, y una nueva rejilla de datos, la *DataGridView*.

Los mayores cambios, sin embargo, afectan al entorno de desarrollo de Visual Studio 2005. Hay un nuevo tipo de entidades, llamadas fuentes de datos, o *data sources*, alrededor de las cuales gira toda la maquinaria de acceso a datos de la nueva versión. Podemos seguir usando la metodología de la versión anterior… pero Visual Studio ha escondido las herramientas necesarias, y es más complicado trabajar de esta manera.

Este apéndice al curso se centra, por lo tanto, en cómo podemos aprovechar este nuevo recurso para no perder productividad, y de paso, para disfrutar todas las ventajas que nos ofrece.

¿A qué clase pertenece una fuente de datos?

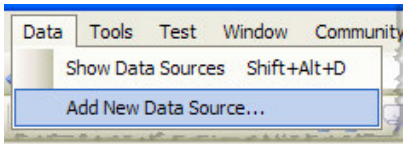
Se trata de una pregunta tramposa: las fuentes de datos sólo existen dentro del proyecto, en tiempo de diseño y compilación. Durante la ejecución, sin embargo, no existe ninguna clase que corresponda directamente a una fuente de datos. Es cierto que en ejecución permanecen las clases generadas a partir de estas entidades, pero la correspondencia funciona en un solo sentido.

¿Es ésta una buena idea? Tengo mis dudas. Una de las virtudes predicadas por la ingeniería de software es la uniformidad. Es deseable que la misma herramienta nos sirva en las distintas fases del desarrollo. No siempre es posible, por supuesto, pero es algo a lo que debemos intentar acercarnos. Pues bien: con los nuevos *datasources* tenemos entidades conceptuales que no son clases (¡aunque podrían serlo, y aunque de hecho se utilicen para generar clases!). ¿Qué hay de malo en ello? Suponga que debe importar una fuente de datos definida en un proyecto a un nuevo proyecto. ¿Cómo lo hace? Tropezaremos con esta dificultad más adelante, cuando intentemos repartir el nuevo código de acceso a datos entre los módulos de un sistema dividido en capas.

No obstante, tenemos que ser prácticos. Gracias a este sistema, ahora disponemos de una técnica más sencilla y a la vez más poderosa para crear interfaces visuales enlazadas a bases de datos.

Creación de fuentes de datos

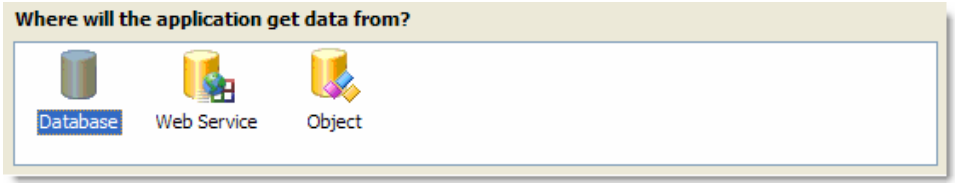
Abra Visual Studio 2005 y cree un proyecto vacío. Despliegue a continuación el submenú *Data* en el menú principal del entorno de desarrollo:



Aunque Visual Studio 2003 también tenía un submenú con el mismo nombre, los comandos en su interior eran muy distintos. Para empezar, ejecute el comando *Show Data*

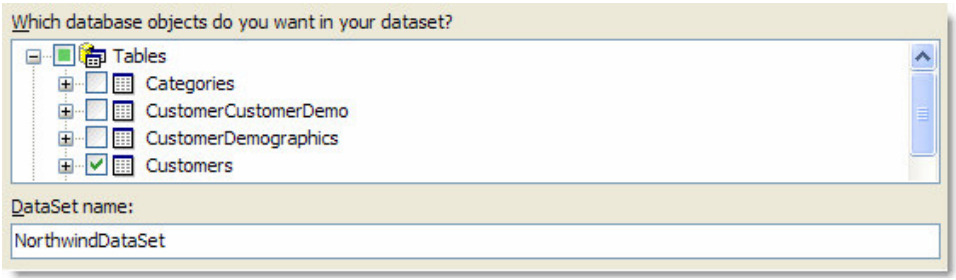
Sources, para activar la ventana que mostrará las fuentes de datos asociadas al proyecto. A continuación, vamos a ejecutar *Add New Data Source*, para que aparezca el asistente de creación de fuentes de datos.

En su primera página nos preguntarán de dónde procederán los "datos":

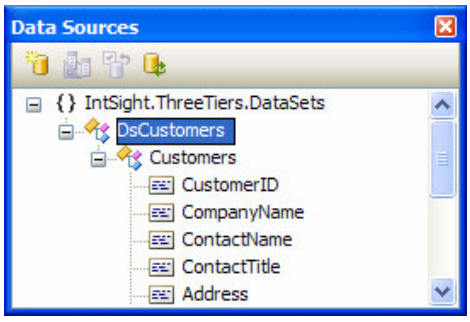


Ya estamos viendo uno de los objetivos más importantes de las fuentes de datos: permitir el tratamiento más o menos uniforme de los distintos orígenes de datos con los que puede trabajar .NET. Detrás de una fuente de datos se puede ocultar un servicio Web, una colección de objetos persistentes o un servidor de bases de datos viejo, bueno y predecible.

Si elegimos una base de datos, la segunda página nos pregunta por la cadena de conexión. Podemos elegir alguna de las configuradas en el *Explorador de Servidores*, o crear una nueva. También podemos almacenar la cadena en el fichero de configuración de la aplicación, para reutilizarla más adelante. Una vez indicado el servidor, debemos elegir los objetos de la base de datos que vamos a incluir en la fuente de datos. Para este ejemplo elegiré una sola tabla: la tabla *Customers* de la archifamosa *Northwind*, la base de datos de ejemplos de SQL Server.

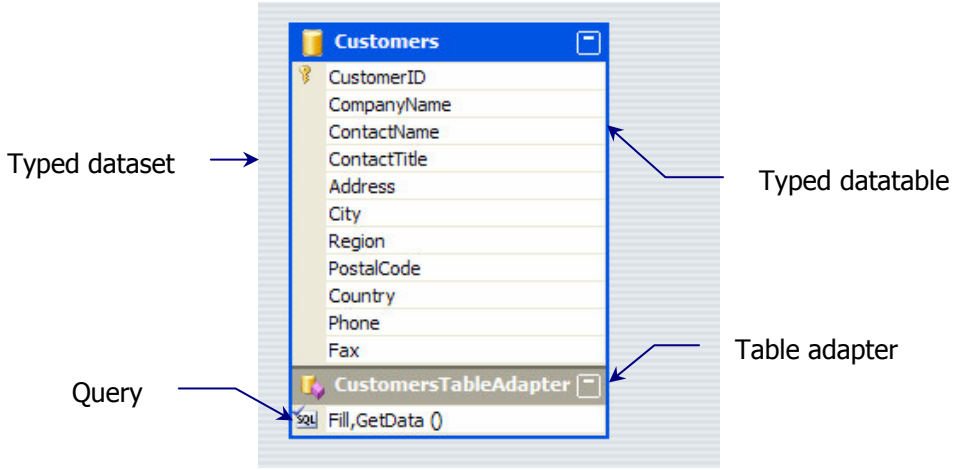


Como puede ver, estamos dando exactamente los mismos pasos que si fuésemos a definir un conjunto de datos con tipos o un adaptador de datos en la versión anterior de Visual Studio. ¡Incluso se nos pide descaradamente un nombre para cierto conjunto de datos! Cuando cerramos el asistente, no obstante, el primer indicio de que hemos cambiado algo en el proyecto es la aparición de un nuevo nodo en la ventana que muestra los conjuntos de datos del proyecto:

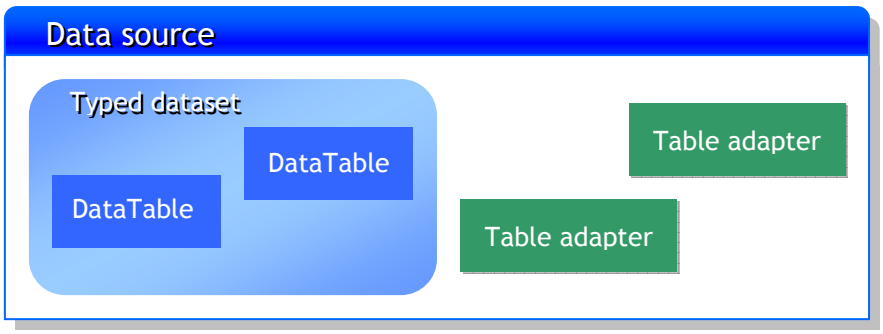


En cambio, si vamos al *Explorador de Soluciones*, veremos que se ha creado un fichero de extensión *xsd*, la misma que se utilizaba y se sigue utilizando para las definiciones de conjuntos de datos con tipos. Si hacemos doble clic sobre el fichero *xsd*, además, apare-

cerá el editor de conjuntos de datos con tipos de toda la vida… aunque veremos que hay cambios. Esta imagen muestra la entidad que corresponde a una definición de tablas:



Sólo la parte superior de esta caja representa una parte del conjunto de datos: una de sus tablas, en concreto. Por cada tabla incluida en el conjunto de datos se define en paralelo una segunda clase, categorizada como un *adaptador de tabla*, o *table adapter*.



Está claro el significado de las líneas dentro de la tabla de datos: se trata de las columnas seleccionadas de la tabla SQL. Sin embargo, el adaptador de tabla también se crea, al menos, con una fila y, como veremos luego, se pueden añadir más "filas" al adaptador. En este caso se trata de consultas o *queries*, y corresponden a sentencias SQL que se pueden activar con la ayuda del adaptador. La primera consulta, la que se crea por omisión, encapsula la sentencia **select** utilizada para definir el conjunto de datos:

```
select * from Customers
```

En este ejemplo estamos tratando con una fuente de datos de estructura muy simple, pues tenemos una sola tabla. Por supuesto, podemos incluir más de una tabla y establecer relaciones entre ellas, al igual que hacíamos con los conjuntos de datos con tipos de Visual Studio 2002/2003.

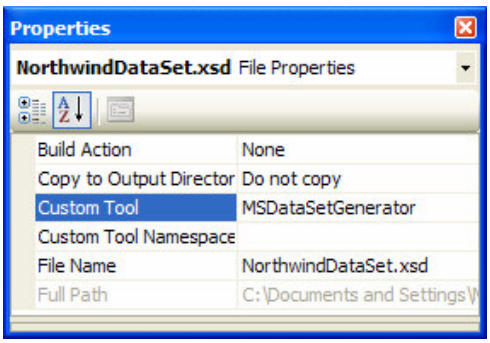
NOTA

Al parecer, Microsoft ha hecho algunos cambios en el formato XML de los ficheros XSD utilizados para la definición de conjuntos de datos con tipos, con el fin de soportar el nuevo concepto de fuente de datos. Al menos en la beta, ha desaparecido el botón que nos permitía editar manualmente la definición en XML del conjunto o fuente de datos. Esto significa, entre otras cosas, que ha desaparecido de momento (o al menos, no sé dónde han metido) la posibilidad de añadir anotaciones al esquema XML para controlar el nombre de las clases generadas.

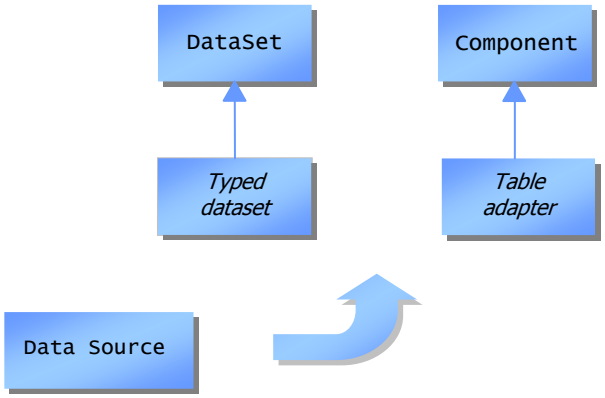
Clases generadas a partir de una fuente de datos

Si ya ha trabajado con los conjuntos de datos con tipos de .NET v1.1, recordará que el resultado de este editor gráfico se almacena como un fichero XSD (*XML Schema Defi-*

nition). Cuando el proyecto se compila, el fichero XSD se envía a una herramienta identificada como *MSDataSetGenerator* en las propiedades del proyecto, que corresponde a la aplicación *xsd.exe*, incluida en el SDK de la plataforma.



Como resultado de este proceso, se generan y compilan varias clases. Entre ellas tenemos las ya familiares relacionadas con el conjunto de datos en sí, aunque ahora se generan clases genéricamente denominadas *adaptadores de tablas*:



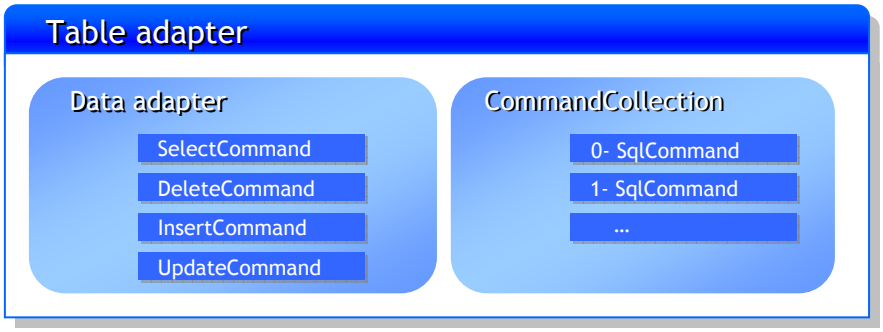
El diagrama sólo muestra las dos clases principales generadas, pero dentro de la clase generada para el conjunto de datos con tipos se declaran varias clases adicionales, para representar tablas, filas y eventos personalizados. Por ejemplo, si creamos un conjunto de datos con tipos que sólo contiene una tabla llamada *Customers*, se crean las siguientes clases anidadas dentro del conjunto de datos con tipos:

Clase	Ancestro
CustomersDataTable	*System.Data.DataTable*
CustomersRow	*System.Data.DataRow*
CustomersRowChangeEvent	*System.EventArgs*
CustomersRowChangeEventHandler	*System.MulticastDelegate*

Adaptadores de tablas

Además, para cada tabla incluida en el conjunto de datos se crea una clase adicional: el adaptador de tablas correspondiente. Las clases de estos adaptadores de tablas, a diferencia de lo que ocurre con los conjuntos de datos con tipos, no tienen ningún tipo de ancestro común. Todas estas clases, por omisión, descienden directamente de la clase *Component*, aunque podemos indicar otra clase base con la ayuda del editor gráfico. Tampoco hay una interfaz común que se deba implementar, y esto complica bastante la programación de código genérico. Más adelante veremos cómo se puede superar este obstáculo.

Dentro de un adaptador de tablas típico encontramos la siguiente estructura:



Por una parte, hay un adaptador de datos interno, con nivel de visibilidad privado. En realidad, este adaptador constituye el núcleo del adaptador de tablas, por lo que podemos considerar que un adaptador de tablas es una especie de "adaptador de datos con tipos".

El adaptador contiene un comando con la instrucción **select** que define el contenido del conjunto de datos asociado. Pero existe también un vector independiente de comandos, que contiene como mínimo un comando con una sentencia **select**. Este vector existe porque podemos asociar más de una consulta al adaptador de tablas. La situación más frecuente que justifica su necesidad es la existencia de distintos criterios de búsqueda sobre la misma tabla. Por ejemplo, podemos definir un adaptador de tablas mediante la siguiente instrucción elemental:

```
select * from Customers
```

Ahora bien, es muy probable que no nos interese traer todos los registros de clientes a la capa de presentación, especialmente si la tabla es grande. Por lo tanto, utilizaremos estas otras dos consultas para dicho propósito:

```
select * from Customers
where  CompanyName like @name

select * from Customers
where  CompanyId = @companyId
```

La primera, o una versión más sofisticada, serviría para la búsqueda de registros. La segunda consulta nos permitiría recuperar los datos de un registro de manera independiente. En el adaptador de tablas correspondiente, tendríamos una propiedad *CommandCollection* con dos entradas… o tres, en el caso en que hubiésemos configurado el adaptador con la consulta sin restricciones inicial.

Veamos los niveles de visibilidad de los componentes de un adaptador de tablas:

Propiedad	Visibilidad
Adapter	**private**
ClearBeforeFill	**public**
Connection	**internal**
CommandCollection	**protected**

¡Parece más un catálogo de niveles de acceso que cualquier otra cosa! Entiendo, por ejemplo, que la propiedad *Connection* se declare como interna: si es necesario, podremos hacer que dos adaptadores de tablas utilicen exactamente la misma instancia de conexión, y a la vez, no será posible enredar con esta propiedad fuera del ensamblado donde hemos creado el adaptador de tablas. Entiendo también, aunque no comparto, la decisión de hacer que *Adapter* sea una propiedad privada. Supongo que el programador pensó que *Adapter* apuntaba a un objeto demasiado delicado para usted y yo lo manejásemos impunemente. Por otra parte, *ClearBeforeFill* es una propiedad muy útil cuyo

valor podemos cambiar de acuerdo a las circunstancias. Cuando está activa, el adaptador elimina cualquier registro del conjunto de datos antes de leer los nuevos registros.

Lo que no alcanzo a comprender es el motivo por el que *CommandCollection* es una propiedad protegida… teniendo en cuenta que es la única con este nivel de acceso. ¿Pensó el programador que podíamos necesitarla en una clase derivada? En tal caso, ¿por qué entonces no nos permite acceder al resto de las propiedades desde la hipotética clase derivada?

NOTA

En realidad, el nivel de accesibilidad de la propiedad *Connection* se puede modificar en tiempo de diseño. Basta con seleccionar el adaptador en el editor gráfico de la fuente de datos y ajustar la propiedad especial *ConnectionModifier*. Es "especial" porque no existe tal propiedad en tiempo de ejecución, sino que se emplea solamente para generar el código del adaptador. Este es un truco de Visual Studio que veremos repetirse muchas veces.

Veamos los métodos que se generan para un adaptador de tablas recién creado:

Método	Propósito
Fill	Lee registros dentro de un conjunto de datos ya existente
GetData	Crea un conjunto de datos, lo llena y lo devuelve
Insert	Inserta un registro dados los valores de sus columnas
Update	Método sobrecargado (ver a continuación)
Delete	Elimina un registro dados los valores de sus columnas

Como verá, hay un método para cada una de las cuatro operaciones clásicas del lenguaje de manipulación de datos de SQL. En realidad, la instrucción **select** se las ha arreglado para conseguir dos métodos:

```
public virtual int Fill(DsCustomers.CustomersTable dataTable);
public virtual DsCustomers.CustomersTable GetData();
```

El método *Fill* exige que le pasemos un conjunto de datos para llenarlo de registros, mientras que *GetData* se encarga de crear el conjunto de datos por su cuenta y riesgo. Por cierto, ¿se ha dado cuenta de que son métodos virtuales?

Las tres operaciones de actualización tienen sus correspondientes métodos:

```
public virtual int Insert(
    string CustomerID,
    // …
    string Fax);

public virtual int Update(
    string CustomerID,
    // …
    string Fax,
    string Original_CustomerID,
    // …
    string Original_Fax);

public virtual int Delete(
    string Original_CustomerID,
    // …
    string Original_Fax);
```

No es el tipo de métodos que se utiliza más en las aplicaciones de interfaz gráfica de usuarios, porque estas operaciones casi siempre se llevarán a cabo mediante controles de datos. Sí serían útiles dentro de una aplicación Web.

Finalmente, tenemos cuatro variantes del método que debemos usar para enviar nuestros cambios a la base de datos. Por desgracia, el método se llama también *Update*:

```
public virtual int Update(DsCustomers dataSet);
public virtual int Update(DsCustomers.CustomersTable dataTable);
```

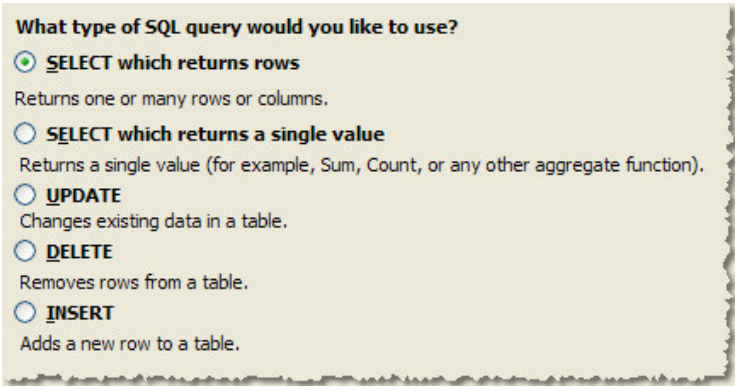
```
public virtual int Update(System.Data.DataRow[] dataRows);
public virtual int Update(System.Data.DataRow dataRow);
```

Sería más apropiado haberlo llamado *Save*, pero es cierto que *Update* es el nombre del método equivalente en la interfaz *IDataAdapter* de .NET v1.1. Hubiera agradecido una variante adicional de *Update* que indicase el tipo de filas a grabar: las nuevas o modificadas, o las eliminadas. De este modo, nos ahorraríamos una llamada al método *Select* de la tabla cuando grabásemos actualizaciones sobre una jerarquía maestro/detalles.

Consultas

Los cinco métodos antes presentados son generados automáticamente cuando creamos un adaptador de tablas. No obstante, una vez que éste existe, podemos añadir más métodos a la clase generada utilizando el editor gráfico. Estos métodos adicionales nos permitirán recuperar filas de la tabla utilizando algún filtro especial, o recuperar valores escalares como el total de registro o el valor medio de alguna columna de tipo numérico, o incluso nos pueden servir para modificar el contenido de la tabla mediante instrucciones SQL o procedimientos almacenados.

Para probar esta posibilidad, debemos pulsar el botón derecho del ratón sobre la barra de título de un adaptador de tablas, y ejecutar el comando de menú *Add Query*. Visual Studio reaccionará mostrándonos un asistente en el que debemos seleccionar si queremos usar una instrucción SQL, un procedimiento almacenado ya existente o uno definido sobre la marcha. Seleccionaremos el uso de instrucciones, y a continuación veremos la página siguiente del asistente:



Aunque el comando se titula *Add Query*, el término "consulta" se interpreta aquí en su sentido más general. Puede tratarse de una modificación, una inserción, un borrado… o incluso una llamada a un procedimiento almacenado, como vimos en la página anterior del asistente. Si usamos una instrucción **select**, ésta puede devolver un conjunto de filas o un valor escalar. Para simplificar, supongamos que usamos una instrucción **select** que devuelve un conjunto de datos:

```
select *
from   dbo.Customers
where  City = @City
```

Observe que hemos usado un parámetro para restringir las filas devueltas por la columna. He usado una condición muy sencilla, pero está claro que podríamos incluir condiciones mucho más complejas. En el siguiente paso del asistente, nos piden el nombre que queremos darle a los dos métodos que se van a generar para la recuperación de datos. En este caso, los nombres más apropiados, junto con los prototipos automáticamente generados, serían los siguientes:

```
public virtual int FillByCity(
    CustomersDataTable dataTable, string city);
public virtual CustomersDataTable GetDataByCity(string city);
```

La novedad consiste en que el parámetro usado en la consulta SQL se ha transmutado en un parámetro del método.

Tipos anulables

Hagamos un experimento: vamos a añadir una consulta a un adaptador de tablas. Para ponernos de acuerdo, supondremos que se trata de la tabla de clientes. Elija una consulta del tipo *SELECT which returns a single value*. Cuando se le pida la instrucción SQL teclee lo siguiente:

```
select * from dbo.Customers
```

Llame *CustomersCount* a la consulta, guarde los cambios y compile el proyecto. Si busca el método generado en la clase del adaptador, se encontrará con el siguiente prototipo:

```
public virtual System.Nullable<int> CustomersCount();
```

Para más misterio, cuando intente llamar el método, deje que se active Intellisense, y verá que el método aparece con este otro prototipo:

```
public virtual int? CustomersCount();
```

Estamos viendo un nuevo recurso de todos los lenguajes en la nueva versión de la plataforma; vamos, no obstante, a explicarlo utilizando la sintaxis de C#. Para que C# tenga tipos anulables, han tenido que ocurrir dos cosas:

- La adición de los tipos genéricos al lenguaje: en realidad, un tipo anulable es una "instancia" del tipo genérico *Nullable<T>*.
- Un par de adiciones sintácticas para simplificar el uso de este tipo especial. En vez de tener que escribir *Nullable<int>*, se acepta *int?* como equivalente. Hay también un nuevo operador, representado por dos interrogaciones sucesivas *??* que funciona de manera parecida a la función **isnull** de SQL Server.

Hay una condición adicional que se exige a los tipos anulables:

- Sólo se permiten tipos anulables construidos sobre tipos de valor.

Por ejemplo, podemos usar *int?*, porque el tipo entero es un tipo primitivo que .NET trata como si se tratase de una estructura. Podemos usar *DateTime?* por las mismas razones. Sin embargo, no se puede usar un tipo anulable basado en el tipo **string**, porque *System.String* es en realidad una clase.

¿Para qué se utilizan estos nuevos tipos? Muy simple: si la estructura base *T* de un tipo anulable *T?* admite *n* valores diferentes, el tipo *T?* admitirá entonces *n+1* valores diferentes. Esto es, se admiten todos los valores originales… más el valor especial **null**. Le podemos dar la interpretación que queramos a este hecho, pero ADO.NET lo utiliza para representar el valor **null** de SQL en determinados casos:

- Para representar valores nulos en las filas de los conjuntos de datos con tipos se sigue utilizando el convenio de la versión 1.1. Este convenio dicta que, cuando una columna puede albergar valores nulos, hay que consultar la función *IsXxxNull* antes de preguntar por el valor de la misma. Si no lo hacemos, se produce una excepción. Por último, para asignar un valor nulo en una columna, se utiliza un método llamado *SetXxxNull*. Nada ha cambiado en los conjuntos de datos con tipos.

- Donde sí se utilizan tipos anulables es en los métodos asociados a los adaptadores de tablas, como veremos enseguida.

Ya hemos visto que el valor devuelto por la consulta escalar se considera como un tipo anulable. Esto se hace así por simple precaución: en realidad, una consulta basada en la función **count***(*)* no puede devolver un valor nulo… pero a Visual Studio le costaría mucho descartar estos casos particulares. Para recibir y utilizar el valor de retorno de un método con este prototipo, utilizaríamos código parecido al siguiente:

```
int? cantidad = customersTableAdapter.CustomersCount();
if (cantidad.HasValue)
    MessageBox.Show(cantidad.Value.ToString());
    // O: cantidad.ToString(), para abreviar
else
    MessageBox.Show("¡Valor nulo!");
```

Para saber si el valor almacenado en *cantidad* es un valor "verdadero", en vez de un valor nulo, se utiliza la propiedad lógica *HasValue*. Una alternativa sería comparar directamente la variable con la constante **null**:

```
if (cantidad != null)
    MessageBox.Show(cantidad.Value.ToString());
```

En caso afirmativo, el valor interno se extrae mediante la propiedad *Value*. En el caso de una expresión de tipo *int?*, *Value* será de tipo entero, y así sucesivamente. También podemos obtener ese valor mediante una conversión forzada de tipos:

```
int? cantidad = customersTableAdapter.CustomersCount();
int c = (int)cantidad;
```

Si intentamos la conversión, o si llamamos a *Value* cuando la variable contiene un nulo, se produce una excepción. Antes mencioné la existencia de un nuevo operador simbólico en C# que emula el comportamiento de la conocida función **isnull** de SQL Server. El operador se representa mediante dos signos de interrogación, y devuelve el primer operando si éste no es nulo, o el segundo, en caso contrario:

```
int? x = null;
// Muestra -1 en la consola
System.Console.WriteLine((x ?? -1).ToString());
```

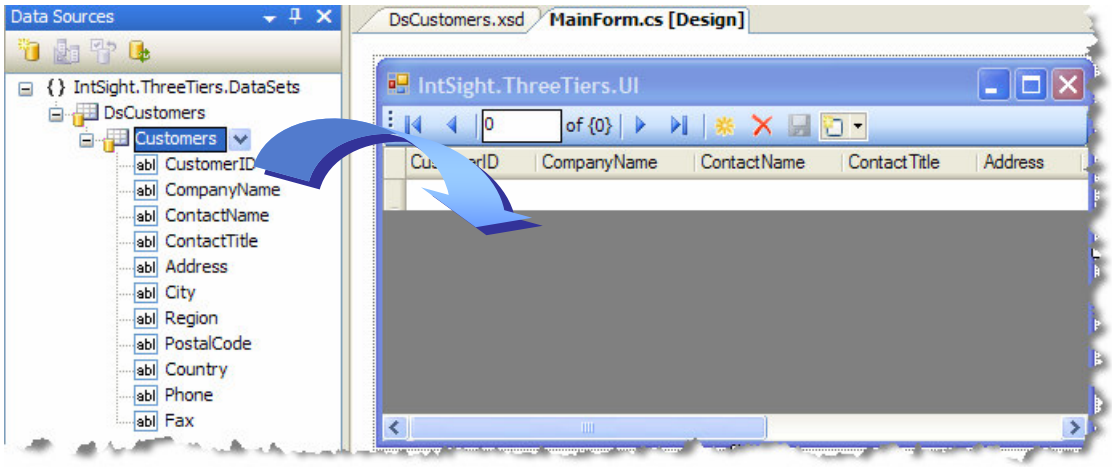
El segundo escenario en el que se utilizan tipos anulables con adaptadores de tablas es en la generación de métodos para la actualización directa de registros. Este es, por ejemplo, el prototipo del método de inserción directa de registros de productos en la base de datos:

```
public virtual int Insert(
    string ProductName, int? SupplierID, int? CategoryID,
    string QuantityPerUnit, decimal? UnitPrice,
    short? UnitsInStock, short? UnitsOnOrder, short? ReorderLevel,
    bool Discontinued);
```

Observe que se han utilizado tipos anulables para los valores de casi todas las columnas… con la excepción de las que son de tipo cadena. Como las cadenas son tipos de referencia, si quisiéramos pasar un nulo para la columna *QuantityPerUnit*, sólo tendríamos que pasar la constante **null** en dicha posición. Lo mismo podríamos hacer con el nombre del producto, pero como la tabla no admite nulos en esta columna, si pasáramos dicho valor se produciría una excepción. En las restantes columnas anulables, se considera también que el puntero nulo, **null**, es equivalente al valor nulo de SQL. Observe que la última columna no es anulable: evidentemente, esto ocurre porque la columna correspondiente en la base de datos no admite nulos.

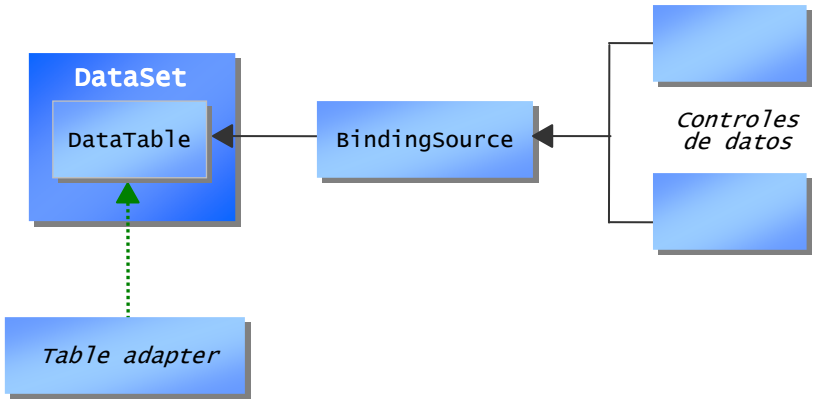
Creación de controles

Una novedad muy llamativa es la posibilidad de arrastrar partes de una fuente de datos sobre un formulario o control de usuario, para crear controles enlazados a datos, como se muestra en la siguiente imagen.



En este caso, hemos arrastrado la tabla completa *Customers* sobre el formulario principal, y Visual Studio ha creado y configurado adecuadamente una rejilla de datos (¡la nueva rejilla, de tipo *DataGridView*!) y una barra de navegación, de tipo *BindingNavigator*. Para comprender todo lo que ha ocurrido al soltar la tabla, veamos qué componentes se han añadido en la zona inferior del área de diseño:

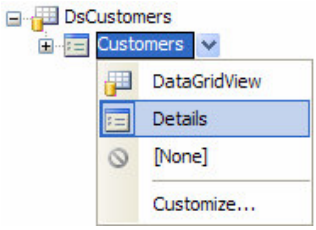
Por una parte, se ha creado una instancia en tiempo de diseño para el conjunto de datos con tipos definido por el origen de datos. Para facilitar el enlace a datos de los controles que necesitemos usar, se ha añadido un nuevo componente de .NET 2.0: *BindingSource*. Este nuevo componente permite aislar a los controles visuales de la representación concreta del origen de datos con el que deben trabajar. Tanto la rejilla como la barra de navegación harán referencia a este componentes, en vez de usar directamente el conjunto de datos, que es lo que ocurría en las versiones anteriores. Observe, dicho sea de paso, que en la bandeja de componentes hay un icono para la barra de navegación: aunque se trata de un componente visual, este icono simplifica la configuración del control asociado. Por último, hay también una instancia del adaptador de tablas correspondiente a la tabla de clientes:



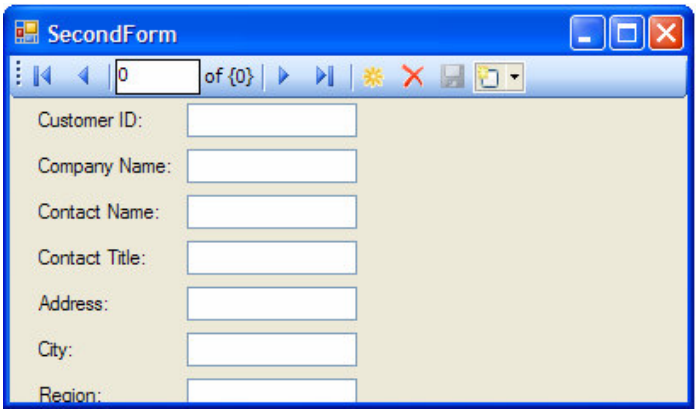
He dibujado una línea discontinua para conectar el adaptador de tablas con la tabla, para indicar que esta conexión no es de tipo permanente, sino que se realiza por medio de código en tiempo de ejecución, mediante la ejecución de un método del adaptador.

La vista de detalles

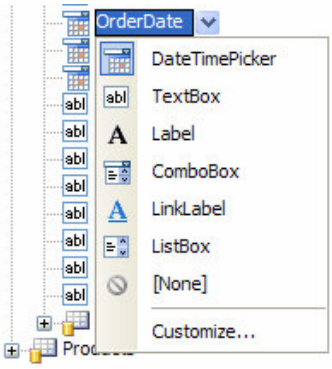
Si desplegamos la lista adyacente a la tabla de clientes, en el árbol de la fuente de datos, podremos controlar cuál control debe crearse cuando se deje caer la tabla sobre un formulario:



Inicialmente, como hemos visto, Visual Studio crea rejillas cuando se arrastra una tabla, pero podemos elegir *Details* si queremos generar una vista para la edición o navegación registro a registro. En realidad, no hay una clase que se llame *Details*, sino que es un truco de diseño para crear el diseño típico en estos casos:



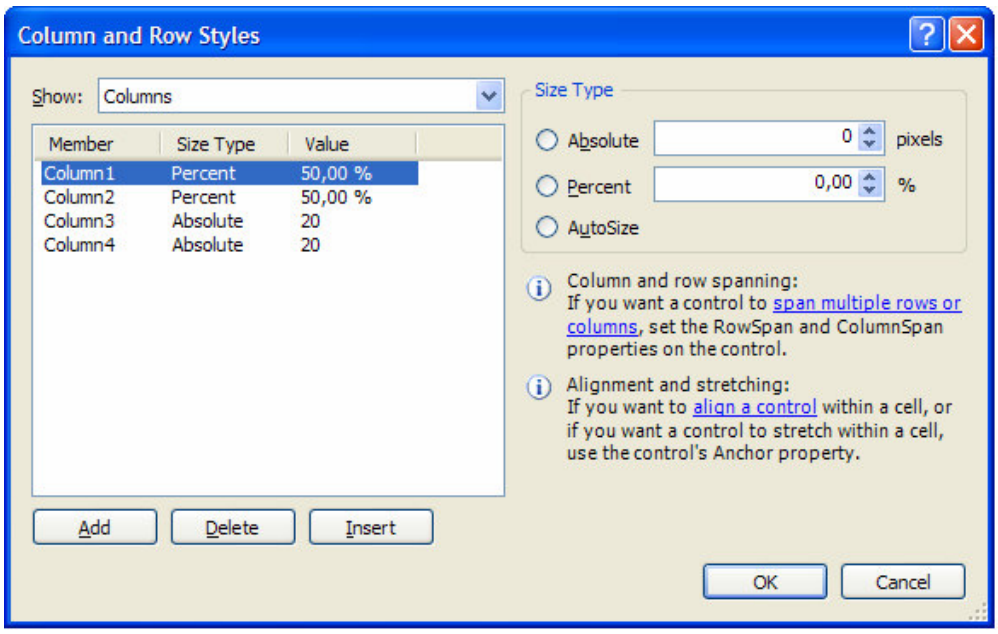
¿Y qué ocurre cuando hay tantas columnas en una tabla que no caben en el formulario? Está claro que podemos arrastrar directamente las columnas, una por una, para lograr cualquier diseño que nos dicte nuestra fantasía. Para cada columna, además, podemos elegir qué tipo de control queremos:



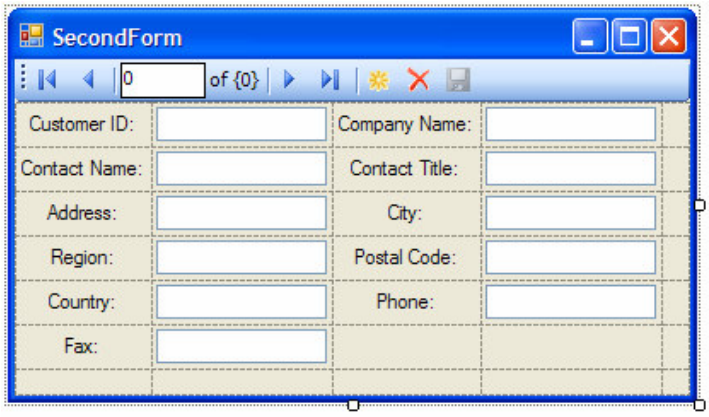
De todos modos, Visual Studio 2005 guarda una bala para casos de apuro: los nuevos controles que aparecen agrupados bajo el título *Containers*. En concreto, nos interesa el control llamado *TableLayoutPanel*. Deje caer uno de estos sobre un formulario vacío, haga que ocupe toda el área cliente del mismo, y modifique su propiedad *Columns* para

que haya cuatro columnas, en vez de las dos que se crean automáticamente. Deje caer nuevamente la fuente de datos, esta vez sobre el control de diseño tabular, y contemple el resultado.

En verdad, el aspecto inicial del formulario no será muy esperanzador, pero todo se arregla si ejecuta el comando *Edit Rows and Columns*, de la lista de tareas asociadas al control. Este es el cuadro de diálogo que debe aparecer:



Si me permite una sugerencia, puede seleccionar la opción *AutoSize* para las cuatro columnas. Luego puede mostrar las filas, en el mismo diálogo, y activar *AutoSize* para todas ellas. Con esto, el aspecto mejorará enormemente:



En realidad, he creado una fila y una columna adicional, para que sean estas las que crezcan cuando se redimensione el formulario.

Para rematar, Visual Studio nos permite registrar nuevas clases de controles para que aparezcan como opciones al configurar el aspecto visual de las columnas de un origen de datos.

Transacciones

Todas estas adiciones en tiempo de diseño son llamativas… pero quien se haya acostumbrado a trabajar directamente con las clases que ofrece ADO.NET tanto en ésta versión como en las anteriores, puede sentirse un poco perdido al ver que esta nueva meto-

dología esconde de su vista las clases ya familiares. El problema más importante se presenta cuando hay que utilizar transacciones. La metodología para el uso de transacciones en .NET v1.1 era bastante complicada, para ser sinceros, y a falta de alguna sorpresa en la versión definitiva de Visual Studio 2005, no hay signos de que se vaya a simplificar ahora.

Por suerte, existen remedios sencillos para atajar cualquier complicación en este sentido. Veamos, en primer lugar, en qué consiste el problema:

- Primero, aclaremos que vamos a trabajar con transacciones creadas y manejadas explícitamente. Una solución alternativa sería utilizar servicios corporativos (léase COM+) y transacciones declarativas, con *"enlistamiento automático"*. Sí, es una alternativa aceptable, pero este tipo de transacciones es menos eficiente que las transacciones explícitas "de toda la vida", y mientras no lo requiera la complejidad del modelo de datos y de aplicación, es mejor no recurrir a estos extremos.
- Una transacción "normal" se crea a partir de la conexión. Esto no es mayor problema: aunque inicialmente la propiedad *Connection* de los distintos adaptadores de tablas se declara como **internal**, esto significa que podemos utilizarla sin más dentro del mismo ensamblado donde se definen los adaptadores. Y en último caso, el diseñador visual de los adaptadores de tablas permite que elijamos el nivel de protección de esta propiedad.
- Pero el primer problema que encontramos es que, inicialmente, cada adaptador utiliza su propio componente de conexión, sin compartirlo con otros adaptadores. Para sincronizar varios adaptadores dentro del paraguas de una misma transacción tendremos que asignar la misma conexión a todos ellos.
- Más complicado aún: los adaptadores de datos, es decir, los componentes pertenecientes a la clase *SqlDataAdapter*, están enterrados profundamente dentro de los adaptadores de tablas. Hay una propiedad *Adapter* que nos permitiría llegar a estos objetos… pero por desgracia, el generador de código la declara como **private**, y esta vez el diseñador visual no nos permite modificar el nivel de acceso.
- Incluso si tuviésemos acceso a los *SqlDataAdapter*, nos quedaría pendiente escribir un patrón de código bastante engorroso: para que cada uno de los tres comandos que potencialmente contiene un adaptador pueda ejecutarse en el contexto de una transacción, hay que asignar una referencia a ésta en cada una de las propiedades *Transaction* de cada comando.

Como ya he dicho, hay soluciones sencillas, aunque parezca complicado. Eso sí: para simplificar la explicación voy a utilizar un ejemplo de lectura de datos, en vez de lo normal, que sería mostrar una transacción que sincronice escrituras. No es muy frecuente encontrar transacciones asociadas a la lectura de registros… aunque muchas veces la coherencia de la información recuperada así lo exigiría. La situación real más frecuente en que se necesita una transacción para lecturas es la presencia de información redundante procedente de más de un registro. Por ejemplo:

- Departamentos y empleados, si el departamento mantiene una columna con el número de empleados.
- Cuentas bancarias y movimientos, si la cuenta mantiene una columna con el saldo actual.

Esta redundancia no tiene por qué corresponder a un mal diseño. Muchas veces, la columna redundante se utiliza para acelerar la verificación y cumplimiento de las reglas de negocio.

Clases parciales

Nuestra salvación lleva el nombre de una adición a los lenguajes .NET en esta versión: las *clases parciales*. Sabemos, y si no lo sabía se lo cuento ahora, que cuando se compila un proyecto en C#, el compilador mezcla de manera virtual todos los ficheros de textos para generar el ensamblado de salida a partir de esa mezcla. No hay nada en C# que permita compilar un fichero por separado para generar posteriormente ensamblados o módulos mezclando el resultado de estas compilaciones "parciales". Es decir: no existe en .NET un formato de ficheros compilados similar al formato OBJ de los viejos tiempos de DOS y Windows. Como corolario, no existe "enlazador" o *linker*, hablando con propiedad, en .NET.

Le advierto que esta es una filosofía estupenda para el diseño de lenguajes. Se acabaron los problemas con las referencias recursivas entre módulos, por ejemplo. Y como atestigua la experiencia, esto no implica una carga desmedida para el compilador. Es cierto que los viejos compiladores clásicos que hacían toda su labor en una o dos pasadas eran monstruos de eficiencia, pero con la disponibilidad actual de memoria RAM, no es problema alguno representar todo el código de un proyecto en un árbol de sintaxis en memoria.

No obstante, en las versiones anteriores de C#, cada clase debía definirse completamente dentro de un mismo fichero fuente. Ya no es necesario hacer tal cosa, siempre que marquemos la clase con el nuevo modificador **partial**. Cuando se compile el proyecto, el compilador reunirá todos los fragmentos parciales y recompondrá el código fuente completo. Y ya está. Tenga presente que estos fragmentos sólo pueden estar dentro de un mismo proyecto. No existe la posibilidad de ampliar una clase "parcial" definida en un ensamblado ya compilado.

Esta nueva característica ha sido ideada para simplificar el uso de código generado mecánicamente. Por ejemplo, en las versiones anteriores, la configuración de un formulario en tiempo de diseño se traducía en un método situado dentro del fichero de código asociado al formulario. En ese fichero también debíamos ubicar nuestro propio código. Era fácil corromper el código generado por el entorno visual, y el propio entorno se las tenía que pasar canutas para localizar la zona en la que podía escribir sin preocupaciones.

En Visual Studio 2005, por el contrario, cuando creamos un formulario, el entorno crea dos ficheros de código relacionados. Si el formulario se llama form1, el primero de ellos se llamará *form1.cs*, y será donde escribiremos nuestro código, igual que antes. Ahora bien, el código generado por el diseñador visual irá a parar a *form1.designer.cs*, un segundo fichero de código, que es el que debemos respetar. En ambos ficheros se definirá un fragmento de clase parecido a éste:

```
public partial class Form1 : Form …
```

En realidad, en el fichero administrado por el diseñador visual, la clase se declarará así:

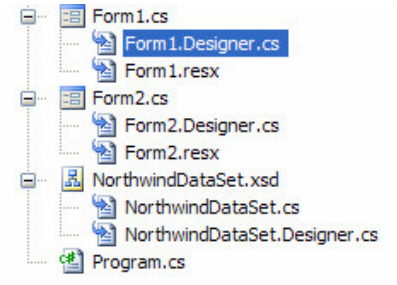
```
partial class Form1 …
```

Esta vez, no se declara la clase base ni el nivel de acceso a la nueva clase. De este modo, estos detalles pueden especificarse en el fichero que manejamos directamente. Por supuesto, la segunda especificación parcial podría haber aportado la clase base y el

nivel de acceso, y en ese caso se comprobaría, al compilar la clase, que ambas especificaciones coincidiesen.

Una propiedad para las transacciones

Como puede imaginar, esto mismo se repite con las clases generadas para una fuente u origen de datos: hay un fichero con el código generado, que no debemos tocar, y un fichero en el que podemos teclear lo que nos dé la gana.



Hay varias técnicas para activar este fichero personalizado:

- En el Explorador de Soluciones, pulse el botón derecho del ratón sobre el fichero *xsd* y ejecute el comando de menú *View Code*.
- Del mismo modo, puede activar el editor visual del conjunto de datos con un doble clic sobre el fichero *xsd*, y usar el menú de contexto del editor visual para ejecutar nuevamente *View Code*.

Inicialmente, veremos la siguiente declaración dentro del fichero que se nos presenta:

```
public partial class NorthwindDataSet
{
}
```

¿Y el resto de las clases? Hay que tener mucho cuidado al llegar aquí: las clases generadas para los tipos especializados de tablas y filas se definen como clases anidadas dentro de la clase antes mencionada. Si quisiéramos añadir miembros, por ejemplo, a la clase *CustomersDataTable*, nos bastaría con añadir lo siguiente:

```
public partial class NorthwindDataSet
{
    public partial class CustomersDataTable
    {
    }
}
```

Ahora bien, las clases que realmente nos interesan son las clases generadas como adaptadores de tablas. Y por algún motivo, Microsoft decidió declarar estas clases como clases independientes, ¡pero ubicadas en un espacio de nombres anidado! Si quisiéramos ampliar tanto *CustomersDataTable* como el adaptador de tablas asociados, tendríamos que modificar de esta manera el fichero de código:

```
namespace MiProyecto
{
    public partial class NorthwindDataSet
    {
        public partial class CustomersDataTable
        {
        }
    }
```

```
        namespace NorthwindDataSetTableAdapters
        {
            partial class CustomersTableAdapter {
            }
        }
    }
```

No se trata de física nuclear; sólo tenemos que conocer el convenio de nombres utilizado por Visual Studio.

En este paso, retocaremos los adaptadores de tablas que queramos involucrar en nuestra transacción. No podemos generar el objeto de transacción dentro de la clase del adaptador, usando directamente la conexión que éste debe contener. ¿El motivo? Que la transacción será común a varios de estos adaptadores, por lo que es mejor generar el objeto en territorio neutro. Pero lo que sí haremos es declarar una propiedad para asignar la transacción a cada comando de un mismo adaptador. Quitando el envoltorio de las declaraciones de espacios de nombres, esto es lo que necesitaríamos para cada adaptador de tablas que vayamos a usar:

```
public partial class OrdersTableAdapter
{
    private SqlTransaction transaction;

    public SqlTransaction Transaction
    {
        get { return transaction; }
        set
        {
            transaction = value;
            SqlDataAdapter ad = this.Adapter;
            if (ad.DeleteCommand != null)
                ad.DeleteCommand.Transaction = value;
            if (ad.InsertCommand != null)
                ad.InsertCommand.Transaction = value;
            if (ad.UpdateCommand != null)
                ad.UpdateCommand.Transaction = value;
            foreach (SqlCommand cmd in this.CommandCollection)
                cmd.Transaction = value;
        }
    }
}
```

Observe cómo accedemos impunemente a la propiedad privada *Adapter*: ¡estamos en la misma clase en que la que se define dicha propiedad! Aunque, eso sí, en un fichero diferente… Note también que hemos asignado la transacción no solamente a los tres adaptadores configurados para las actualizaciones, sino también a cada comando presente en la propiedad *CommandCollection*. Recuerde que ahí es donde se almacenan los comandos utilizados para las consultas a la medida, como *CustomersCount*.

Un coordinador de transacciones

Acabo de decir que la coordinación de la transacción debe efectuarse desde territorio neutro, al involucrar potencialmente a más de un adaptador de tablas. En principio, podríamos añadir el método necesario al conjunto de datos… pero es muy mala idea. El conjunto de datos pertenece tanto a la capa intermedia como a la capa de presentación. Sin embargo, las lecturas y escrituras sobre la base de datos pertenecen solamente a la capa intermedia. Lo que podemos hacer es crear una clase aparte, que podemos incluso declarar en el mismo fichero *NorthwindDataSet.cs*, si nos apetece. Vamos a declarar la

nueva clase con el modificador **static**, otra característica añadida a C# 2.0. Las clases estáticas sólo pueden declarar miembros estáticos, y el efecto es similar al de las antiguas "bibliotecas de funciones" anteriores a la programación orientada a objetos:

```
public static class OrderDetailsCoordinator
{
    // …
}
```

El método que definiremos asumirá las siguientes responsabilidades:

- Elegirá un objeto de conexión para asociarlo a todos los adaptadores de tablas involucrados. Puede ser un objeto creado a propósito, o puede reutilizar la conexión ya existente de un adaptador de tablas elegido al azar.
- Abrirá la conexión, y garantizará que la cierra antes de terminar.
- Creará una transacción sobre la conexión elegida, y la asignará a cada uno de los adaptadores de tablas involucrados. Es recomendable que antes de terminar, asigne un puntero nulo en las propiedades *Transaction* de estos adaptadores de tablas, para permitir que el objeto de transacción ya usado pueda ser eliminado por el colector de basura.
- Ejecutará los comandos deseados en los adaptadores de tablas. En este caso, se trata de la lectura de registros a partir de los adaptadores.
- Si todo ha ido bien, se confirmará el éxito de la transacción. Si se produce una excepción, antes de propagarla al código que llama a nuestro método, debemos anular la transacción.

Y este es el código fuente que necesitamos:

```
public static void Fill(
    TabAdapt.NorthwindDataSet dataset,
    OrdersTableAdapter ordersAdapter,
    Order_DetailsTableAdapter detailsAdapter)
{
    SqlConnection conn =
        detailsAdapter.Connection = ordersAdapter.Connection;
    conn.Open();
    try
    {
        SqlTransaction trans =
            conn.BeginTransaction(IsolationLevel.Serializable);
        detailsAdapter.Transaction = trans;
        ordersAdapter.Transaction = trans;
        try
        {
            ordersAdapter.Fill(dataset.Orders);
            detailsAdapter.Fill(dataset.Order_Details);
            trans.Commit();
        }
        catch
        {
            trans.Rollback();
            throw;
        }
        detailsAdapter.Transaction = null;
        ordersAdapter.Transaction = null;
    }
```

```
        finally
        {
            conn.Close();
        }
    }
```

Es cierto que hay unas cuantas líneas de código en nuestro método, pero es posible reducir el tamaño del método si utilizamos algunos trucos. Observe que no he "protegido" la desconexión de la transacción de los adaptadores usados con una instrucción **try/finally**. El motivo ha sido no complicar el fragmento anterior, pero está claro que debíamos haber usado el bloque de protección.

Hagamos balance

El nuevo sistema de clases, ¿es bueno o es malo? Creo que hay más cosas positivas que negativas. En mi opinión, lo más interesante es la posibilidad de crear controles enlazados a datos mediante arrastrar y soltar. La encapsulación de los antiguos adaptadores de datos es otra cosa: es discutible si se trata de la mejor técnica posible o no, pero en cualquier caso, me temo que no será la "técnica definitiva".

Hay varias promesas por implementar en la beta 2, y habrá que esperar a la versión final para pronunciarse: me estoy refiriendo a la posibilidad de asociar manejadores de eventos a los eventos de modificación de tablas y columnas en tiempo de diseño. En teoría, un doble clic sobre una columna en el editor visual del conjunto de datos con tipos bastaría para definir un método en el fichero de código que podemos modificar y enlazarlo desde el fichero mantenido por el diseñador. El problema con esta técnica, que falla todavía en la beta 2, es que sin ella se hace muy complicado añadir manejadores de eventos desde un fragmento de una clase parcial. En realidad, hay ironía en este asunto: tengo la impresión de que la persona que se inventó este tinglado sólo programa en Visual Basic.NET… un lenguaje que tiene una cláusula *handles* para asociar manejadores de eventos de manera supuestamente declarativa.

Hay que tener en cuenta también si vamos a utilizar este sistema en una aplicación monolítica o en un sistema dividido en capas físicas. En este último caso, hay varios problemas a superar para poder adaptar esta estructura de clases. El problema está en que los adaptadores de tablas mezclan elementos de la capa de presentación con las clases que acceden a la base de datos física. No hay nada que no tenga solución, de momento, pero prefiero esperar a la versión definitiva antes de lanzarme a recomendar paliativos.

En definitiva, el anuncio de LINQ para la tercera generación de lenguajes de la plataforma presagia la llegada de cambios masivos en algunas de las interfaces de programación para bases de datos. Estaremos atentos…

NOTA Este documento es una versión preliminar y deliberadamente abreviada del documento que se incluirá temporalmente como parte del curso de **Programación con ADO.NET en C#** a la espera de su actualización a .NET v2.0.