

Utilización del modo no conectado

En un modo no conectado, el enlace con el servidor de base de datos no es permanente. Hay que conservar de forma local los datos con los cuales se desea trabajar. La idea es volver a crear, con la ayuda de diferentes clases, una organización similar a la de una base de datos. Las principales clases vienen representadas en el siguiente esquema:

`DataSet`

Es el contenedor de mayor nivel, desempeña el mismo papel que la base de datos.

`DataTable`

Como su nombre indica, es el equivalente de una tabla de la base de datos.

`DataRow`

Esta clase desempeña el papel de un registro (fila).

`DataColumn`

Esta clase reemplaza un campo (columna) de una tabla.

`UniqueConstraint`

Es el equivalente de la clave primaria de una tabla.

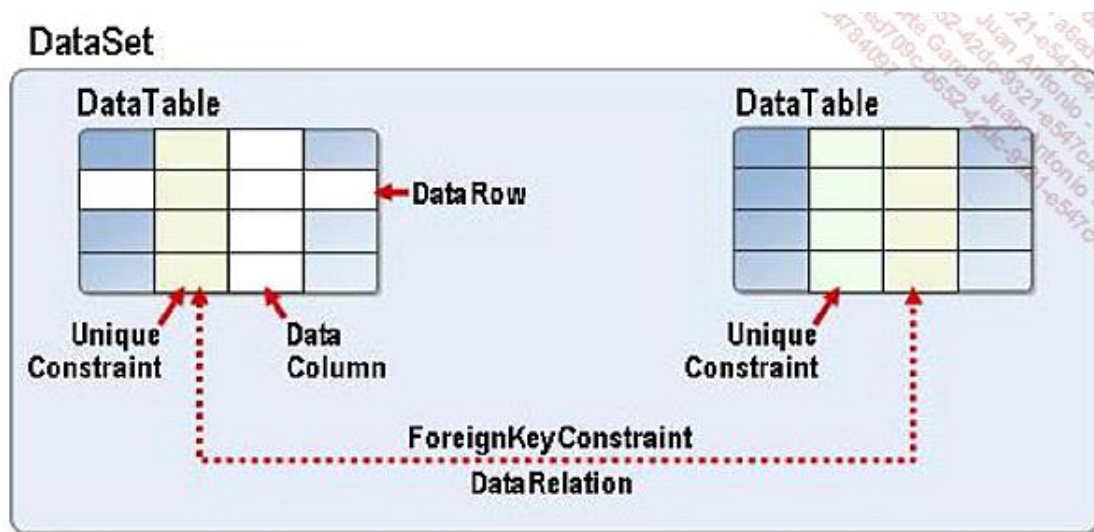
`ForeignKeyConstraint`

Es el equivalente de la clave foránea.

`DataRelation`

Representa un enlace padre/hijo entre dos `DataTable`.

El esquema siguiente representa esta organización.



Ahora vamos a ver cómo crear y manejar todas estas clases.

1. Rellenar un DataSet a partir de una base de datos

Para poder trabajar localmente con los datos, debemos volcarlos desde la base de datos en un `DataSet`. Cada proveedor de datos facilita una clase `DataAdapter`, que asegura el diálogo entre la base de datos y un `DataSet`. Todos los intercambios se hacen por el medio de esta clase, tanto desde la base hacia el `DataSet` como desde el `DataSet` hacia la base para la actualización de los datos. El `DataAdapter` utilizará una conexión para contactar con el servidor y uno o varios comandos para el tratamiento de los datos.

a. Utilización de un DataAdapter

La primera tarea que debe hacerse consiste en crear una instancia de la clase `SqlDataAdapter`. Luego debemos configurar el `DataAdapter` con el fin de indicarle qué datos deseamos volcar desde la base de datos. La propiedad `SelectCommand` debe referenciar un objeto `Command`, que contiene la instrucción SQL encargada de seleccionar los datos. El objeto `Command` utilizado también puede llamar un procedimiento almacenado. La única restricción es que la instrucción SQL ejecutada por el objeto `Command` sea una instrucción `SELECT`. La clase `DataAdapter` contiene también las propiedades `InsertCommand`, `DeleteCommand`, y `UpdateCommand`, que hacen referencia a los objetos `Command`, utilizados durante la actualización de la base de datos. Mientras no deseemos efectuar una actualización de la base, estas propiedades son opcionales. Se estudiarán más en detalle en la sección Utilización del modo no conectado - Actualizar la base de datos, en este capítulo.

El método `Fill` de la clase `DataAdapter` se utiliza para rellenar el `DataSet` con el resultado de la ejecución del comando `SelectCommand`. Este método espera como parámetro el `DataSet` que debe rellenar y un objeto `DataTable` o una cadena de caracteres que se usa para nombrar el `DataTable` en el `DataSet`. El `DataAdapter` utiliza, internamente, un objeto `DataReader` para obtener el nombre de los campos y el tipo de los campos con objeto de crear el `DataTable` en el `DataSet` y luego rellenarlo con los datos. El `DataTable` y los `DataColumn` se crean sólo si no existen anteriormente. En caso de que sí, el método `Fill` utiliza dicha estructura existente. Si se crea un `DataTable`, se añade a la colección **Tablas** del `DataSet`. El tipo de datos de los `DataColumn` se define en función de los mapeos previstos por el proveedor de datos, entre los tipos de la base de datos y los tipos .NET. El siguiente ejemplo rellena un `DataSet` con el código, el apellido, la dirección y la ciudad de los clientes.

```
public static void TestDataSet1()
{
    SqlCommand cmd;
    SqlConnection ctn;
    DataSet ds;
    SqlDataAdapter da;

    ctn = new SqlConnection();
    ctn.ConnectionString = "Data Source=localhost;Initial
                          Catalog=Northwind;Integrated Security=true";

    cmd = new SqlCommand();
    cmd.Connection = ctn;
    cmd.CommandText = " SELECT CustomerId,ContactName,Address,city from
                      Customers";

    ds = new DataSet();
    da = new SqlDataAdapter();
    da.SelectCommand = cmd;
    da.Fill(ds, "Customers");
}
```

En este código, la conexión no ha sido ni abierta ni cerrada explícitamente. En efecto, el método `Fill` abre la conexión si ya no está abierta y, en este caso, la vuelve a cerrar también al final de su ejecución. Sin embargo, si necesita utilizar varias veces el método `Fill`, es más eficaz que gestione por sí mismo la apertura y el cierre de conexión. En todos los casos, el método `Fill` deja la conexión en el estado en el que la ha encontrado.

Por supuesto, un `DataSet` puede contener varios `DataTable` creados a partir de `DataAdapter` diferentes. Los datos pueden provenir de bases de datos diferentes, incluso de tipos de servidores diferentes.

Cuando el `DataAdapter` construye el `DataTable`, los nombres de los campos de la base se utilizan para nombrar los `DataColumn` . Es posible personalizar estos nombres creando objetos `DataTableMapping` y añadiéndolos a la colección **TableMappings** del `DataAdapter`. Estos objetos `DataTableMapping` contienen ellos mismos objetos `DataColumnMapping` utilizados por el método `Fill` , como traductores entre los nombres de los campos en la base y los nombres de los `DataColumn` en el `DataSet` . En este caso, durante la llamada del método `Fill` , debemos indicarle el nombre del `DataTableMapping` que se ha de utilizar. Si para uno o varios campos no hay mapeo disponible, entonces el nombre del campo en la base se utiliza como nombre para el `DataColumn` correspondiente. Por ejemplo, podemos utilizar esta técnica para traducir los campos de la base **Northwind**.

El siguiente código efectúa esta traducción y visualiza el nombre de los `DataColumn` del `DataTable` creado:

```
public static void TestTableMapping()
{
    SqlCommand cmd;
    SqlConnection ctn;
    DataSet ds;
    SqlDataAdapter da;
    DataTableMapping mapeo;
    ctn = new SqlConnection();
    ctn.ConnectionString = "Data Source=localhost;Initial
                          Catalog=Northwind;Integrated Security=true";
    cmd = new SqlCommand();
    cmd.Connection = ctn;
    cmd.CommandText = " SELECT CustomerId,ContactName,Address,city from
                      Customers";

    ds = new DataSet();
    da = new SqlDataAdapter();
    da.SelectCommand = cmd;
    mapeo = new DataTableMapping("Customers", "Clientes");
    mapeo.ColumnMappings.Add("CustomerId", "CodigoCliente");
    mapeo.ColumnMappings.Add("ContactName", "Apellido");
    mapeo.ColumnMappings.Add("Address", "Dirección");
    mapeo.ColumnMappings.Add("city", "Ciudad");
    da.TableMappings.Add(mapeo);
    da.Fill(ds, "Customers");
    foreach ( DataColumn dc in ds.Tables["Clientes"].Columns)
    {
        Console.Write(dc.ColumnName + "\t");
    }
}
```

Esto es lo que visualizamos:

CodigoCliente	Apellido	Dirección	Ciudad
---------------	----------	-----------	--------

b. Añadir restricciones a un DataSet

El método `Fill` sólo transfiere hacia el `DataSet` los datos que provienen de la base. Cuando la tabla que trae el `DataAdapter` de la base de datos presenta restricciones, éstas no se vuelcan en el `DataSet` . Para poder recuperar estas restricciones en el `DataSet` , hay dos soluciones posibles:

- Modificar la propiedad `MissingSchemaAction` del `DataAdapter` con el valor `MissingSchemaAction.AddWithKey` .

```
da.MissingSchemaAction = MissingSchemaAction.AddWithKey;
```

- Proceder en dos etapas llamando primero el método `FillSchema` del `DataAdapter` para crear la estructura completa del `DataTable`, y luego llamar el método `Fill` para rellenar el `DataTable` con los datos.

```
da.FillSchema(ds, SchemaType.Mapped, "Customers");  
da.Fill(ds, "Customers");
```

El segundo parámetro del método `FillSchema` indica si se debe tener en cuenta el mapeo o si se utiliza la información proveniente de la base de datos.

Es importante añadir las restricciones de claves primarias, ya que el método `Fill` se va comportar de manera diferente según existan o no.

Si existen a nivel del `DataSet`, cuando el método `Fill` importa un registro desde la base, verifica si ya no existe una fila con el valor de clave primaria en el `DataTable`. Si es el caso, sólo actualiza los campos de la fila existente. Si, por el contrario, no hay una fila con un valor de clave primaria idéntica, entonces se crea la fila en el `DataTable`.

Si no hay restricción de clave primaria en el `DataTable`, el método `Fill` añade todos los registros procedentes de la base de datos. En este caso, puede que haya duplicados en el `DataTable`. Eso es particularmente importante cuando se debe llamar el método `Fill` varias veces para, por ejemplo, obtener los datos modificados por otra conexión a la base de datos.

2. Configurar un `DataSet` sin base de datos

No es necesario disponer de una base de datos para poder utilizar un `DataSet`; puede servir de alternativa a la utilización de tablas para la gestión interna de los datos de una aplicación. En este caso, todas las operaciones efectuadas automáticamente por el `DataAdapter` deberán realizarse manualmente mediante el código. Esto incluye en particular la creación de los `DataTable` con sus `DataColumn`. La primera operación que se ha de realizar consiste en crear una instancia de la clase `DataTable`. El constructor espera como parámetro el nombre de la `DataTable`. Luego se utiliza este nombre para identificar el `DataTable` en la colección `Tables` del `DataSet`. Después de su creación, el `DataTable` no contiene estructura alguna. Por lo tanto, debemos crear uno o varios `DataColumn` y añadirlos a la colección **Columns** del `DataTable`.

Se pueden crear los `DataColumn` haciendo uso de los constructores de la clase o automáticamente durante la adición a la colección **Columns**. La primera solución aporta más flexibilidad, ya que permite la configuración de numerosas propiedades del `DataColumn` en el momento de su creación. Debe, como mínimo, indicar un nombre y un tipo de datos para el `DataColumn`.

```
col = new DataColumn("Bruto", Type.GetType("decimal"));  
table.Columns.Add(col);  
table.Columns.Add("Iva", Type.GetType("decimal"));
```

Un `DataColumn` también se puede construir como una expresión basada en uno o varios otros `DataColumn`. En este caso, debe indicar durante la creación del `DataColumn` la expresión que sirve para calcular su valor. Por supuesto el tipo de datos generado por la expresión debe ser compatible con el tipo de datos del `DataColumn`. También debe tener cuidado con el diseño de la expresión, respetar las mayúsculas o minúsculas y vigilar con no crear referencias circulares entre los `DataColumn`.

```
table.Columns.Add("Neto", Type.GetType("decimal"), "Bruto * (1 + Iva / 100)");
```

Para asegurar la unicidad de los valores de un `DataColumn`, es posible utilizar un tipo de `DataColumn` autoincrementado. La propiedad `AutoIncrement` de este `DataColumn` se debe colocar en **true**. También puede modificar el paso de incremento con la propiedad `AutoIncrementStep` y el valor de salida con la propiedad `AutoIncrementSeed`. El valor que contiene este `DataColumn` se calcula automáticamente durante la inserción de una fila en el `DataTable`, en función de estas propiedades y filas ya existentes en la `DataTable`.

Este tipo de `DataColumn` se suele utilizar como clave primaria de una `DataTable`. Usted tiene la posibilidad de definir la clave primaria de un `DataTable` facilitando la propiedad `PrimaryKey` de una tabla que contenga los diferentes `DataColumn` que deben componer la clave primaria. Los `DataColumn` implicados verán que algunas de sus propiedades se modifican automáticamente. La propiedad `Unique` se colocará en **true**, y la propiedad `AllowDBNull`, en **false**. Si la clave primaria está constituida de varias `DataColumn`, sólo se modificará la propiedad `AllowDBNull` en estos `DataColumn`.

```
col = new DataColumn("Numero", Type.GetType("int"));
col.AutoIncrement = true;
col.AutoIncrementSeed = 1000;
col.AutoIncrementStep = 1;
table.Columns.Add(col);
table.PrimaryKey=new DataColumn[] {col};
```

3. Manejar los datos en un DataSet

Sea cual sea el método utilizado para rellenar un `DataSet`, el objetivo de cualquier aplicación consiste en manejar los datos presentes en el `DataSet`. La clase `DataTable` contiene muchas propiedades y métodos que facilitan el manejo de los datos.

a. Lectura de los datos

La lectura de los datos es la operación más frecuente realizada en un `DataSet`. Primero hay que obtener una referencia sobre la `DataTable` que contiene los datos: luego podemos recorrer la colección **Rows** del `DataTable`. Esta colección es una instancia de la clase `DataRowCollection`. Por defecto, dispone de la propiedad `Item`, que permite el acceso a una fila particular por un índice. La propiedad `count` permite conocer el número de filas disponibles. En un `DataTable`, no hay noción de puntero de registro, de registro corriente, de métodos de desplazamiento en el juego de resultados. Si quiere gestionar todas estas nociones, debe administrarlas explícitamente en su código. El método `GetEnumerator` pone a nuestra disposición una instancia de clase que implementa la interfaz `IEnumerator`. Gracias a esta instancia de clase, tenemos acceso a los métodos `MoveNext` y `Reset`, así como a la propiedad `Current`. Estos tres elementos permiten recorrer fácilmente todas las filas del `DataTable`. Cada fila corresponde a una instancia de la clase `DataRow`. Esta clase posee también una propiedad `Item` por defecto que aporta un acceso a los diferentes campos del `DataRow`. Se puede obtener cada campo gracias a su nombre o su índice.

El siguiente código ilustra estas nociones mostrando la lista de los clientes:

```
public static void TestLecturaDataTable()
{
    SqlCommand cmd;
    SqlConnection ctn;
    DataSet ds;
    SqlDataAdapter da;
    IEnumerator en;
    ctn = new SqlConnection();
    ctn.ConnectionString = "Data Source=localhost;Initial
                          Catalog=Northwind;Integrated Security=true";

    cmd = new SqlCommand();
    cmd.Connection = ctn;
    cmd.CommandText = " SELECT ContactTitle,ContactName from Customers";
    ds = new DataSet();
    da = new SqlDataAdapter();
    da.SelectCommand = cmd;
    da.Fill(ds, "Customers");
    // se recupera el enumerador en las filas+ de la DataTable
    en = ds.Tables["Customers"].Rows.GetEnumerator();
```

```
// nos volvemos a situar al principio de la tabla (por seguridad)
en.Reset();
// bucleamos hasta que el método MoveNext nos indica que queda filas
while (en.MoveNext())
{
    // accedemos a los campos por el nombre
    Console.Write(((DataRow)en.Current)["ContactName"] + "\t");
    // o por el numero
    Console.WriteLine(((DataRow)en.Current)[0]);
}
Console.ReadLine();
}
```

b. Creación de restricciones sobre una DataTable

Puede utilizar restricciones para activar limitaciones sobre los datos presentes en un `DataTable`. Las restricciones constituyen reglas que se aplican a un `DataColumn` o a sus `DataColumn` relacionadas. Determinan las acciones efectuadas cuando se modifica el valor contenido en una fila. Sólo se tienen en cuenta para un `DataSet` si su propiedad `EnforceConstraints` se coloca en **true**.

Se pueden utilizar dos tipos de restricciones:

UniqueConstraint

Este tipo de restricción va a garantizar que el valor o los valores presentes en un `DataColumn` o un grupo de `DataColumn` sean únicos. La instalación de una restricción única se efectúa al crear una instancia de la clase `UniqueConstraint` con la lista de los `DataColumn` afectados por la restricción. Luego, esta `UniqueConstraint` se debe añadir a la colección **Constraints** del `DataTable` .

```
table.Constraints.Add(new UniqueConstraint(new DataColumn[] { col }));
```

Si la restricción sólo se refiere a un `DataColumn` , también es posible modificar simplemente la propiedad `Unique` de este `DataColumn` a **true**, para crear una restricción única. Hay que observar también que la creación de una clave primaria genera automáticamente una restricción única; sin embargo, lo contrario no es cierto. La violación de la restricción tras una modificación de una fila desencadena una excepción.

ForeignKeyConstraint

Las `ForeignKeyConstraint` controlan cómo van a comportarse los `DataTable` relacionados durante la modificación o la supresión de un valor en el `DataTable` principal. Se puede considerar una acción diferente para una supresión y para una modificación. La clase `ForeignKeyConstraint` dispone de las propiedades `DeleteRule` y `UpdateRule` , que indican el comportamiento durante la supresión o la modificación. Son posibles los valores siguientes:

Cascade

La supresión o modificación se propaga a la fila o las filas relacionadas.

SetNull

El valor se modifica a `DBNull` en las filas relacionadas.

SetDefault

El valor por defecto se toma en las filas relacionadas.

None

No se lleva a cabo ninguna acción sobre las filas relacionadas.

La adición de una `ForeignKeyConstraint` se hace por la creación de una instancia indicando los `DataColumn` del `DataTable` padre y los `DataColumn` de la tabla hijo. Si varios `DataColumn` forman parte de la restricción, se facilitan en forma de tabla.

El siguiente código añade una restricción entre el `DataTable` **Facturas** y el `DataTable` **LineasFactura**, para que la supresión de una factura conlleve la supresión de todas sus filas.

```
var fkFact_LineasFact = new ForeignKeyConstraint("FK_FACT_LIGNESFACT",
        ds.Tables["Facturas"].Columns["Numero"],
        ds.Tables["LineasFactura"].Columns["NumFact"]);
fkFact_LineasFact.AcceptRejectRule = AcceptRejectRule.Cascade;
fkFact_LineasFact.DeleteRule = Rule.Cascade;
ds.EnforceConstraints = true;
```

c. Creación de relaciones entre las DataTables

En un `DataSet` que contiene varios `DataTable`, puede añadir relaciones entre los `DataTable`. Estas relaciones permiten la navegación entre las filas de los diferentes `DataTable`. Debe crear una instancia de la clase `DataRelation` y añadirla a la colección `Relations` del `DataSet`. La creación se puede hacer directamente con el método `Add` de la colección `Relations`. La información que hay que facilitar es:

- El nombre de la relación que permite encontrar a continuación la `DataRelation` en la colección.
- El `DataColumn` o los `DataColumn` padres bajo la forma de una tabla de `DataColumn` si hay varias.
- El `DataColumn` o los `DataColumn` hijas bajo la forma de una tabla si hay varias.

El código siguiente añade una relación entre la tabla **Customers** y la tabla **Orders**:

```
ds.Relations.Add("Cliente_Pedidos",
        ds.Tables["Customers"].Columns["CustomerId"],
        ds.Tables["Orders"].Columns["CustomersId"]);
```

Hay que observar que las `DataRelation` funcionan en paralelo con las `ForeignKeyConstraint` y las `UniqueConstraint`. Por defecto, la creación de la relación va a colocar una `UniqueConstraint` en la tabla padre y una `ForeignKeyConstraint` en la tabla hijo. Si no desea que estas restricciones se agreguen automáticamente en caso de que no existan, debe añadir un booleano **false** como cuarto parámetro durante la adición de la `DataRelation`.

d. Recorrer las relaciones

El objetivo principal de las relaciones consiste en permitir la navegación de un `DataTable` hacia otro en el interior de un `DataSet`. Así podemos obtener todos los objetos `DataRow` de un `DataTable` vinculados con un `DataRow` de otro `DataTable`. Por ejemplo, después de haber cargado las tablas **Customers** y **Orders** en el `DataSet` y establecido una relación entre estas dos tablas, podemos, desde una fila del `DataTable` **Customers**, obtener del `DataTable` **Orders** todos los pedidos de este cliente. El método `GetChildRows` devuelve en forma de una tabla de `DataRow` todas las filas que contienen los pedidos de este cliente.

Este método toma como parámetro el nombre de la `DataRelation` utilizada para seguir el enlace. El siguiente ejemplo de código aplica esto: muestra, para cada cliente, el número y la fecha de sus pedidos:

```
public static void TestRelations()
{
```

```

SqlCommand cmdCustomers;
SqlCommand cmdOrders;
SqlConnection ctn;
DataSet ds;
SqlDataAdapter daCustomers;
SqlDataAdapter daOrders;

ds = new DataSet();
ctn = new SqlConnection();
ctn.ConnectionString = "Data Source=localhost;Initial
                        Catalog=Northwind;Integrated Security=true";

cmdCustomers = new SqlCommand();
cmdCustomers.Connection = ctn;
cmdCustomers.CommandText = " SELECT * from Customers";
daCustomers = new SqlDataAdapter();
daCustomers.SelectCommand = cmdCustomers;
daCustomers.Fill(ds, "Customers");

cmdOrders = new SqlCommand();
cmdOrders.Connection = ctn;
cmdOrders.CommandText = " SELECT * from Orders";
daOrders = new SqlDataAdapter();
daOrders.SelectCommand = cmdOrders;
daOrders.Fill(ds, "Orders");

ds.Relations.Add("Cliente_Pedidos",
                 ds.Tables["Customers"].Columns["CustomerId"],
                 ds.Tables["Orders"].Columns["CustomerId"]);
foreach ( DataRow lineaCliente in ds.Tables["Customers"].Rows)
{
    Console.WriteLine(lineaCliente["ContactName"]);
    foreach ( DataRow lineaPedidos in
        lineaCliente.GetChildRows("Cliente_Pedidos"))
    {
        Console.WriteLine("\t" + "pedido N {0} de {1}",
                          lineaPedidos["OrderId"],
                          lineaPedidos["OrderDate"]);
    }
}
}

```

La navegación de una fila hijo hacia una fila padre también es posible usando el método `GetParentRow`, que también espera como parámetro el nombre de la relación utilizada como enlace.

La parte del código siguiente muestra, para cada pedido, el nombre del cliente que lo ha hecho:

```

foreach (DataRow l in ds.Tables["Orders"].Rows)
{
    Console.WriteLine("el pedido {0} ha sido pasado por {1}",
        l["OrderId"], l.GetParentRow("Cliente_Pedidos")["ContactName"]); }

```

e. Estado y versiones de una DataRow

La clase `DataRow` es capaz de monitorizar las diferentes modificaciones aplicadas a los datos que contiene. La propiedad `RowState` permite controlar las modificaciones aportadas a la fila.

Son posibles cinco valores definidos en una enumeración para esta propiedad:

Unchanged

La fila no ha cambiado desde que se llenó el `DataSet` con el método `Fill` o desde que se aceptaron las modificaciones con el método `AcceptChanges`.

Added

La fila se ha añadido, pero las modificaciones aún no han sido aceptadas por el método `AcceptChanges`.

Modified

Uno o varios campos de la fila se han modificado.

Deleted

La fila se ha borrado, pero las modificaciones aún no han sido aceptadas por el método `AcceptChanges`.

Detached

La fila se ha creado pero aún no forma parte de la colección **Rows** de un `DataTable`.

También se dispone de las diferentes versiones de una fila. Cuando acceda a los valores contenidos en una fila, puede especificar la versión que le interesa.

Para ello, la enumeración `DataRowVersion` propone cuatro valores:

Current

Versión actual de la fila. Esta versión no existe para una fila cuyo estado es `Deleted`.

Default

Versión por defecto de la fila. Para una fila cuyo estado es `Added`, `Modified`, `Unchanged`, esta versión es equivalente a la versión `Current`. Para una fila cuyo estado es `Deleted`, esta versión es equivalente a la versión `Original`. Para una fila cuyo estado es `Detached`, esta versión es igual a la versión `Proposed`.

Original

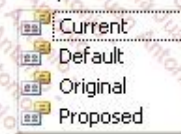
Versión de origen de la fila. Para una fila cuyo estado es `Added`, esta versión no existe.

Proposed

Versión transitoria disponible durante una operación de modificación de la fila o para una fila que no forma parte de la colección **Rows** de un `DataTable`.

Se debe especificar la versión deseada durante el acceso a un campo particular de un `DataRow`. Para ello, hay que utilizar una de las constantes anteriores a continuación del nombre o del índice del campo durante la utilización de la propiedad `Item`, por defecto, del `DataRow`.

```
ds.Tables["Customers"].Rows[1]["ContactName", DataRowVersion.]
```



Se utilizarán estas diferentes versiones durante la actualización de la base de datos para gestionar los accesos concurrentes por ejemplo.

f. Adición de datos

La adición de una fila a un `DataTable` se efectúa simplemente al añadir un `DataRow` a la colección **Rows** de un `DataTable`. Previamente hace falta crear una instancia de la clase `DataRow`. Es a este nivel cuando encontramos un problema.

```
DataRow NuevaFila;  
NuevaFila = new DataRow();  
'System.Data.DataRow.DataRow(System.Data.DataRowBuilder)' no es accesible debido a su nivel de protección
```

No hay constructor disponible para la clase `DataRow`. Tranquilícese, no es un error sino algo realmente voluntario que no haya constructor para esta clase. En efecto cuando necesitamos una nueva instancia de un `DataRow`, no queremos un `DataRow` cualquiera sino un `DataRow` específico al esquema de nuestro `DataTable`. Por esta razón se le confía a él la tarea de crear la instancia que necesitamos por medio del método `NewRow`.

```
DataRow nuevaLinea;  
nuevaLinea = ds.Tables["Customers"].NewRow();
```

El estado de esta fila es de momento `Detached`. Luego podemos añadir datos en esta nueva fila.

```
nuevaLinea["ContactName"] = "García";
```

Después de ello, nos queda añadir la fila de la colección **Rows** del `DataTable`.

```
ds.Tables["Customers"].Rows.Add(nuevaLinea);
```

El estado de esta nueva fila es ahora `Added`.

g. Modificación de datos

Se realiza la modificación de los datos contenidos en una fila simplemente asignando a los campos correspondientes los valores deseados. Estos valores están almacenados en la versión `Current` de la fila. El estado de la fila es entonces `Modified`. Esta solución presenta una pequeña desventaja. Si se modifican simultáneamente varios campos de una fila, puede haber estados transitorios que violen restricciones colocadas en el `DataTable`. Por ejemplo, es el caso si existe en el `DataTable`, una restricción de clave primaria colocada en dos `DataColumn`. Esto tiene como efecto activar una excepción. Para paliar este problema, podemos pedir que se ignore temporalmente la verificación de las restricciones para esta fila. El método `BeginEdit` pasa la fila en modo edición y suspende entonces la verificación de las restricciones para esta fila. Los valores asignados a los campos no están almacenados en la versión `Current` de la fila, sino en la versión `Proposed`. Cuando haya finalizado con las modificaciones de la fila, las puede validar o cancelar llamando respectivamente al método `EndEdit` o el método `CancelEdit`. También, puede verificar los valores gestionando el evento `ColumnChanged` del `DataTable`.

En el gestor de eventos, recibe un argumento de tipo `DataColumnChangeEventArg` que permite saber qué `DataColumn` ha sido modificado (`args.Column.ColumnName`), el valor propuesto para este `DataColumn` (`args.ProposedValue`) y que permite cancelar las modificaciones (`args.row.CancelEdit`). En caso de validación con el método `EndEdit`, la versión `Proposed` de la fila se copia en la versión `Current` y el estado de la fila se convierte en `Modified`. Si, por el contrario, cancela las modificaciones con el método `CancelEdit`, la versión `Current` no se modifica y el estado de la fila es `unchanged`. En todos los casos, después de la llamada de uno de estos dos métodos, se reactiva la verificación de las restricciones.

El siguiente ejemplo permite la modificación del código postal de un cliente verificando que éste es efectivamente numérico:

```
public static void TestModificationLigne ()  
{  
    SqlCommand cmd;  
    SqlConnection ctn;  
    string codigoCliente;
```

```

        string codigoPostal;
        SqlParameter paramCodigoCliente;
        DataSet ds;
        SqlDataAdapter da;
        DataTable table;
        ctn = new SqlConnection();
        ctn.ConnectionString = "Data Source=localhost;Initial
                                Catalog=Northwind;Integrated Security=true";

        ctn.Open();
        cmd = new SqlCommand();
        cmd.Connection = ctn;
        Console.WriteLine("introducir el código del cliente a modificar:");
        codigoCliente = Console.ReadLine();
        cmd.CommandText = " SELECT * from Customers WHERE CustomerID = @Code";
        paramCodigoCliente = new SqlParameter("@Code", codigoCliente);
        paramCodigoCliente.Direction = ParameterDirection.Input;
        cmd.Parameters.Add(paramCodigoCliente);
        ds = new DataSet();
        da = new SqlDataAdapter(cmd);
        da.Fill(ds, "Clientes");
        table = ds.Tables["Clientes"];
        table.ColumnChanged += table_ColumnChanged;
        table.Rows[0].BeginEdit();
        Console.WriteLine("introducir el nuevo código postal del cliente:");
        codigoPostal = Console.ReadLine();
        table.Rows[0]["PostalCode"] = codigoPostal;
        table.Rows[0].EndEdit();
        Console.WriteLine("el nuevo código postal es: {0}",
                        table.Rows[0]["PostalCode"]);

        Console.ReadLine();
    }
    public static void table_ColumnChanged(object sender,
                                           System.Data.DataColumnChangeEventArgs e)
    {
        int cp;
        if (e.Column.ColumnName == "PostalCode")
        {
            if (int.TryParse(((string)e.ProposedValue), out cp))
            {
                e.Row.CancelEdit();
            }
        }
    }
}

```

h. Supresión de datos

Dispone de dos soluciones diferentes. Puede borrar una fila o suprimir una fila. El matiz entre estas dos soluciones es sutil:

La supresión de una fila se hace con el método `Remove`, que retira definitivamente la `DataRow` de la colección **Rows** del `DataTable`. Esta supresión es definitiva.

El método `Deleted` sólo marca la fila para suprimirla posteriormente. El estado de la fila pasa a `Deleted` y sólo en el momento de validar las modificaciones se suprime realmente la fila de la colección **Rows** del `DataTable`. Si se cancelan las modificaciones, la fila se queda en la colección **Rows**.

El método `Remove` es un método de la colección **Rows** (actúa directamente sobre su contenido), el método `Delete` es un método de la clase `DataRow` (sólo hace cambiar una propiedad de la fila).

```
// borra la línea
ds.Tables["Customers"].Rows[1].Delete();
// suprime la línea
ds.Tables["Customers"].Rows.Remove(ds.Tables["Customers"].Rows[1]);
```

i. Validar o cancelar las modificaciones

Hasta ahora, las modificaciones efectuadas en una fila son temporales, todavía es posible volver a la versión anterior, o por el contrario, validar de manera definitiva las modificaciones en las filas (pero todavía aún no en la base). Los métodos `AcceptChanges` o `RejectChanges` permiten respectivamente la validación o la anulación de las modificaciones. Se pueden aplicar sobre un `DataRow` individual, un `DataTable` o un `DataSet` entero. Cuando se invoca al método `AcceptChanges`, se desencadenan las siguientes acciones:

- El método `EndEdit` se llama implícitamente para la fila.
- Si el estado de la fila era `Added` o `Modified`, se convierte en `Unchanged` y la versión `Current` se considera la versión `Original`.
- Si el estado de la fila era `Deleted`, entonces se suprime la fila.

El método `RejectChanges` ejecuta las siguientes acciones:

- El método `CancelEdit` se llama implícitamente para la fila.
- Si el estado de la fila era `Deleted` o `Modified`, se convierte en `Unchanged` y la versión `Original` es considerada la versión `Current`.
- Si el estado de la fila era `Added`, entonces se suprime la fila.

Si existen restricciones de clave foránea, la acción del método `AcceptChanges` o `RejectChanges` se propaga a las filas hijos en función de la propiedad `AcceptRejectRule` de la restricción.

j. Filtrar y ordenar datos

Es frecuente necesitar limitar la cantidad de datos visibles en un `DataTable` o aun modificar el orden de las filas. La primera solución que viene a la mente consiste en recrear una consulta SQL con una restricción o una cláusula `ORDER BY`. Pero eso implica olvidar que estamos en un modo de funcionamiento desconectado y que es deseable limitar los accesos a la base o, incluso peor, que la base no esté disponible. Por lo tanto, sólo debemos utilizar los datos disponibles teniendo cuidado de no perderlos. La clase `DataView` nos va a ser muy útil para solucionar nuestros problemas. Esta clase nos va a servir para modificar la visión de los datos en el `DataTable` sin riesgo para los propios datos. Puede haber varios `DataView` para un mismo `DataTable`; corresponden a puntos de vista diferentes del `DataTable`. Prácticamente todas las operaciones realizables en un `DataTable` también lo son mediante un `DataView`.

Hay dos soluciones disponibles para obtener un `DataView`:

- Crear una instancia gracias a uno de los constructores.
- Utilizar la instancia facilitada por defecto por la propiedad `DefaultView`.

El primer constructor utilizable espera simplemente como parámetro el `DataTable` a partir del cual se genera el `DataView`. En este caso, no hay ningún filtro ni tampoco ordenación efectuada sobre los datos visibles por el `DataView`.

Se obtiene un resultado equivalente utilizando la propiedad `DefaultView` de un `DataTable`.

EL segundo constructor permite especificar un filtro, un criterio de ordenación y la versión de las filas implicadas. Para ser visibles en el `DataView`, las filas deberán corresponder a todos estos criterios. También se pueden modificar los diferentes criterios con tres propiedades.

RowFilter

Esta propiedad acepta una cadena de caracteres que representa la condición que se debe completar para que una fila sea visible. Esta condición tiene una sintaxis totalmente similar a las condiciones de una cláusula `WHERE`. Se pueden utilizar los operadores `And` y `Or` para asociar varias condiciones.

El siguiente ejemplo muestra el nombre de los clientes comerciales o directores de venta en España:

```
public static void TestDataView ()
{
    SqlCommand cmd;
    SqlConnection ctn;
    DataSet ds;
    SqlDataAdapter da;
    DataTable table;
    ctn = new SqlConnection(); ctn.ConnectionString = "Data Source=localhost;Initial
                                                    Catalog=Northwind;Integrated Security=true";

    ctn.Open();
    cmd = new SqlCommand();
    cmd.Connection = ctn;
    cmd.CommandText = " SELECT * from Customers";
    ds = new DataSet();
    da = new SqlDataAdapter(cmd);
    da.Fill(ds, "Clientes");
    table = ds.Tables["Clientes"];
    table.DefaultView.RowFilter = "Country='España' and (contactTitle='Sales
                                Agent' or contactTitle='Sales Manager')";
    foreach ( DataRowView fila in table.DefaultView)
    {
        Console.WriteLine("apellido: {0}", fila["ContactName"]);
    }
}
```



Se puede cancelar un filtro asignándole una cadena vacía a la propiedad `RowFilter`.

Sort

Esta propiedad acepta también una cadena de caracteres que representa el criterio o los criterios utilizados para la ordenación. La sintaxis es equivalente a la de la cláusula `ORDER BY`.

El siguiente ejemplo muestra los clientes ordenados primero por país y luego por apellido para un mismo país:

```
//se cancela filtro anterior
table.DefaultView.RowFilter = "";
//todas las filas son visibles ahora
//se añade un criterio de ordenación
table.DefaultView.Sort="Country ASC,ContactName ASC";
foreach(DataRowView fila in table.DefaultView)
{
    Console.WriteLine("País: {0} \t apellido:{1}"
```

```
,fila["Country"],fila["ContactName"]));
```

```
}
```

RowStateFilter

Esta propiedad determina el estado de las filas y qué versión de la fila muestra en el `DataGridView`. Hay ocho posibilidades disponibles:

CurrentRows

Presenta la versión `Current` de todas las filas añadidas, modificadas o sin cambios.

Added

Presenta la versión `Current` de todas las filas añadidas.

Deleted

Presenta la versión `Original` de todas las filas borradas.

ModifiedCurrent

Presenta la versión `Current` de todas las filas modificadas.

ModifiedOriginal

Presenta la versión `Original` de todas las filas modificadas.

None

Ninguna fila.

OriginalRows

Presenta la versión `Original` de todas las filas modificadas, suprimidas o sin cambios.

Unchanged

Presenta la versión `Current` de todas las filas sin cambios.

En el siguiente ejemplo se suprimen dos filas, que pueden visualizarse por medio de un filtro:

```
// se suprimen dos filas
table.Rows[2].Delete();
table.Rows[5].Delete();
// se cancela el filtro
table.DefaultView.RowFilter = "";
// se muestra la versión original de las filas suprimidas
table.DefaultView.RowStateFilter = DataRowState.Deleted;
foreach (DataRowView fila in table.DefaultView)
{
    Console.WriteLine("País: {0} \t apellidos: {1}", fila["Country"],
        fila["ContactName"]);
}
```

k. Buscar datos

La búsqueda de datos se puede realizar con los siguientes dos métodos: `Find` y `FindRows`. Para que funcionen estos dos métodos es imperativo haber ordenado previamente los datos con la propiedad `Sort`.

Find

Este método devuelve el índice de la primera fila que corresponde al criterio de búsqueda. Si no encuentra ninguna fila, devuelve -1. Espera como parámetro el valor que se ha de encontrar. Este valor es buscado por el campo que se utiliza como criterio de ordenación. Si el criterio de ordenación se compone de varios campos, hay que pasar al método `Find` una matriz de objetos que contenga los valores buscados para cada campo del criterio de ordenación en el orden de aparición de la propiedad `Sort`.

Este método se utiliza a menudo para buscar una fila a partir de la clave primaria.

```
public static void TestFind ()
{
    SqlCommand cmd;
    SqlConnection ctn;
    DataSet ds;
    SqlDataAdapter da;
    DataTable table;
    string codigoCliente;
    int indice;
    ctn = new SqlConnection();
    ctn.ConnectionString = "Data Source=localhost;Initial
                          Catalog=Northwind;Integrated Security=true";

    ctn.Open();
    cmd = new SqlCommand();
    cmd.Connection = ctn;
    cmd.CommandText = " SELECT * from Customers";
    ds = new DataSet();
    da = new SqlDataAdapter(cmd);
    da.Fill(ds, "Clientes");
    tabla = ds.Tables["Clientes"];
    Console.WriteLine("Introducir el código de cliente: ");
    codigoCliente = Console.ReadLine();
    tabla.DefaultView.Sort = "CustomerID ASC";
    index = tabla.DefaultView.Find(codigoCliente);
    if (indice == -1)
    {
        Console.WriteLine("No hay ningún cliente con este código");
    }
    else
    {
        Console.WriteLine("El código {0} corresponde al cliente {1}", codigoCliente,
                          tabla.DefaultView[index]["ContactName"]);
    }
    Console.ReadLine();
}
```

FindRows

Este método busca todas las filas que correspondan al criterio de búsqueda y devuelve esas filas en forma de tabla `DataRowView`.

El siguiente código busca todos los clientes de un país y de una ciudad dados:

```

public static void TestFindRows()
{
    SqlCommand cmd;
    SqlConnection ctn;
    DataSet ds;
    SqlDataAdapter da;
    DataTable tabla;
    string pais;
    string ciudad;
    DataRowView[] filasEncontradas;
    object[] criterios;
    ctn = new SqlConnection();
    ctn.ConnectionString = "Data Source=localhost;Initial
                          Catalog=Northwind;Integrated Security=true";

    ctn.Open();
    cmd = new SqlCommand();
    cmd.Connection = ctn;
    cmd.CommandText = " SELECT * from Customers";
    ds = new DataSet();
    da = new SqlDataAdapter(cmd);
    da.Fill(ds, "Clientes");
    tabla = ds.Tables["Clientes"];
    Console.WriteLine("Insertar el país: ");
    pais = Console.ReadLine();
    Console.WriteLine("Insertar la ciudad: ");
    ciudad = Console.ReadLine();
    tabla.DefaultView.Sort = "Country ASC, City ASC";
    criterios = (new object[] {pais, ciudad});
    filasEncontradas = table.DefaultView.FindRows(criterios);
    if (filasEncontradas.Length == 0)
    {
        Console.WriteLine("No se ha encontrado ningún cliente que corresponda");
    }
    else
    {
        Console.WriteLine("Los siguientes clientes corresponden ");
        foreach (DataRowView fila in filasEncontradas)
        {
            Console.WriteLine("Apellido :{0}", fila["ContactName"]);
        }
    }
    Console.ReadLine();
}

```

4. Actualización de la base de datos

Todo el trabajo realizado en los datos con los métodos vistos anteriormente habrá sido en vano si, al cerrar la aplicación, no tenemos cuidado de guardarlos.

En la mayoría de los casos los datos proceden de una base de datos. Por lo tanto, hay que actualizarla con las modificaciones que contienen un `DataSet`, un `DataTable` o un `DataRow`. El `DataAdapter` se utiliza para llenar el `DataSet`. También lo vamos a llamar para actualizar la base de datos.


Al igual que el método `Fill`, el método `Update` va a utilizar sentencias SQL para el diálogo con la base de datos. En

función de las necesidades, utilizará la sentencia contenida en el comando `InsertCommand`, `UpdateCommand` o `DeleteCommand`. Si el método `Update` necesitase un comando que no estuviera disponible, lanzaría una excepción. El método `Fill` recorre las filas del `DataTable` que debe actualizar. Y, en función del estado de la fila, (`Added`, `Deleted`, `Modified`), llama al comando `InsertCommand`, `DeleteCommand`, `UpdateCommand`. El orden en el que se efectúan las actualizaciones en la base de datos podría tener su importancia. Para controlar el orden de ejecución de las inserciones, modificaciones y supresiones, puede proceder en tres etapas, proponiendo al método `Update` un único juego de filas para actualizar. Por ejemplo, podría seleccionar sólo las filas borradas y pedir la actualización de la base de datos con este conjunto de filas y luego proceder de la misma manera con las filas modificadas e insertadas.

El método `Select` permite obtener una tabla `DataRow` que corresponde a un criterio específico. Esta tabla `DataRow` se pasa como parámetro al método `Update`.

El siguiente ejemplo lleva a cabo supresiones, modificaciones e inserciones en la base de datos.

```
DataRow[] filas;
// recupera las filas suprimidas y pide la actualización de la base de datos
filas = tabla.Select(null, null, DataRowState.Deleted);
da.Update(tabla);
// recupera las filas modificadas y solicita la actualización de la base de datos
filas = tabla.Select(null, null, DataRowState.ModifiedCurrent);
da.Update(tabla);
// recupera las filas añadidas y solicita la actualización de la base de datos
filas = tabla.Select(null, null, DataRowState.Added);
da.Update(tabla);
```

 Por supuesto, este ejemplo supone la definición previa de los comandos `InsertCommand`, `DeleteCommand`, `UpdateCommand`.

a. Generación automática de los comandos

Los comandos encargados de la actualización de la base de datos pueden ser generados automáticamente por un objeto `SqlCommandBuilder`. Deben cumplirse algunos requisitos para que el `SqlCommandBuilder` funcione correctamente:

- La propiedad `SelectCommand` debe definirse para `DataAdapter` debido a que las sentencias `INSERT`, `UPDATE`, `DELETE` partirán de ella.
- Debe haber una clave primaria en el `DataTable`.
- Los datos no deben proceder de una unión de varias tablas.

Si no se respeta una o varias de estas exigencias, se lanza una excepción durante la generación de los comandos.

Se generan los comandos respetando los criterios siguientes:

InsertCommand

Inserta una fila en la base de datos para todas las filas cuyo estado sea `Added`. Se actualizan todos los campos excepto los campos identidad, expresión o `TimeStamp`.

UpdateCommand

Actualiza en la base de datos todas las filas cuyo estado sea `Modified`. Y actualiza todos los campos excepto los campos identidad, expresión o `TimeStamp`. Se busca la fila que hay que actualizar mediante la clave primaria, pero también es necesario que los valores de los demás campos correspondan a la versión `Original` del campo en el `DataRow`.

DeleteCommand

Borra de la base de datos todas las filas cuyo estado sea Deleted. Es necesario igualmente que los valores presentes en la base de datos correspondan a la versión Original de los campos del DataRow. Los comandos generados se configurarán mediante los métodos GetInsertCommand, GetUpdateCommand, GetDeleteCommand.

El siguiente ejemplo muestra las instrucciones SQL de los tres comandos generados para la tabla **Customers**:

```
public static void TestOrdreMAJBase ()
{
    SqlCommand cmd;
    SqlConnection ctn;
    DataSet ds;
    SqlDataAdapter da;
    DataTable tabla;
    SqlCommandBuilder bldr;
    ctn = new SqlConnection();
    ctn.ConnectionString = "Data Source=localhost;Initial
                          Catalog=Northwind;Integrated Security=true";

    ctn.Open();
    cmd = new SqlCommand();
    cmd.Connection = ctn;
    cmd.CommandText = " SELECT * from Customers";
    ds = new DataSet();
    da = new SqlDataAdapter(cmd);
    da.Fill(ds, "Clientes");
    table = ds.Tables["Clientes"];
    bldr = new SqlCommandBuilder(da);
    Console.WriteLine("Instrucción SQL de UpdateCommand: {0}",
                      bldr.GetUpdateCommand().CommandText);
    Console.WriteLine("Instrucción SQL de InsertCommand: {0}",
                      bldr.GetInsertCommand().CommandText);
    Console.WriteLine("Instrucción SQL de DeleteCommand: {0}",
                      bldr.GetDeleteCommand().CommandText);
}
```

Este código muestra la siguiente información:

```
Instrucción SQL de UpdateCommand: UPDATE [Customers] SET [CustomerID] =
@p1, [CompanyName] = @p2, [ContactName] = @p3, [ContactTitle] = @p4,
[Address] = @p5, [City] = @p6, [Region] = @p7, [PostalCode] = @p8, [Country] =
@p9, [Phone] = @p10, [Fax] = @p11 WHERE (([CustomerID] = @p12)
AND ([CompanyName] = @p13) AND ((@p14 = 1 AND [ContactName] IS NULL) OR
([ContactName] = @p15)) AND ((@p16 = 1 AND [ContactTitle] IS NULL) OR ([Contact
Title] = @p17)) AND ((@p18 = 1 AND [Address] IS NULL) OR ([Address] = @p19)) AND
((@p20 = 1 AND [City] IS NULL) OR ([City] = @p21)) AND ((@p22 = 1 AND [Region]
IS NULL) OR ([Region] = @p23)) AND ((@p24 = 1 AND [PostalCode] IS NULL) OR
([PostalCode] = @p25)) AND ((@p26 = 1 AND [Country] IS NULL) OR ([Country] =
@p27)) AND ((@p28 = 1 AND [Phone] IS NULL) OR ([Phone] = @p29)) AND ((@p30 = 1
AND [Fax] IS NULL) OR ([Fax] = @p31)))
Instrucción SQL de InsertCommand: INSERT INTO [Customers] ([CustomerID],
[CompanyName], [ContactName], [ContactTitle], [Address], [City], [Region],
[PostalCode], [Country], [Phone], [Fax])
VALUES (@p1, @p2, @p3, @p4, @p5, @p6, @p7, @p8, @p9, @p10, @p11)
Instrucción SQL de DeleteCommand: DELETE FROM [Customers] WHERE
(([CustomerID] = @p1) AND ([CompanyName] = @p2) AND ((@p3 = 1 AND [Contact
```

```
Name] IS NULL) OR ([ContactName] = @p4)) AND ((@p5 = 1 AND [Contact
Title] IS NULL) OR ([ContactTitle] = @p6))
AND ((@p7 = 1 AND [Address] IS NULL) OR ([Address]= @p8)) AND ((@p9 = 1 AND
[City] IS NULL) OR ([City] = @p10)) AND ((@p11 = 1 AND [Region] IS NULL) OR
([Region] = @p12)) AND ((@p13 = 1 AND [PostalCode] IS NULL) OR ([Postal
Code] = @p14)) AND ((@p15 = 1 AND [Country] IS NULL) OR ([Country] = @p16))
AND ((@p17 = 1 AND [Phone] IS NULL) OR ([Phone] = @p18)) AND ((@p19 = 1 AND [Fax]
IS NULL) OR ([Fax] = @p20)))
```

Lo menos que se puede decir es que este código no dice mucho.

Tranquilícese. En el párrafo relativo a los accesos concurrentes, vamos a aclarar la presencia de estos innumerables parámetros en estas tres instrucciones SQL. De momento, lo importante es que estas instrucciones realicen correctamente la actualización de la base de datos.

Lo vamos a comprobar realizando una inserción, una modificación y una supresión en la tabla **Customers** de clientes españoles. Veamos el estado de la tabla antes de efectuar nuestras modificaciones.

select * from customers where country = 'Spain'

	CustomerID	CompanyName	ContactName	ContactTitle	Address
►	BOLID	Bólido Comidas p...	Martín Sommer	Owner	C/ Araquil, 67
	FISSA	FISSA Fabrica In...	Diego Roel	Accounting Man...	C/ Morazarzal, 86
	GALED	Galería del gastr...	Eduardo Saavedra	Marketing Manager	Rambla de Catal...
	GODOS	Godos Cocina Tí...	José Pedro Freyre	Sales Manager	C/ Romero, 33
	ROMEY	Romero y tomillo	Alejandra Camino	Accounting Man...	Gran Vía, 1

A continuación ejecutemos este código:

```
public static void TestMAJBase()
{
    SqlCommand cmd;
    SqlConnection ctn;
    DataSet ds;
    SqlDataAdapter da;
    DataTable table;
    DataRow fila;
    SqlCommandBuilder bldr;
    ctn = new SqlConnection();
    ctn.ConnectionString = "Data Source=localhost;Initial
                          Catalog=Northwind;Integrated Security=true";

    ctn.Open();
    cmd = new SqlCommand();
    cmd.Connection = ctn;
    cmd.CommandText = " SELECT * from Customers where country='Spain'";
    ds = new DataSet();
    da = new SqlDataAdapter(cmd);
    da.Fill(ds, "Clientes");
    tabla = ds.Tables["Clientes"];
    //borra Alejandra Camino
    tabla.Rows[5].Delete();
    // cambia la dirección de José Pedro Freyre
    tabla.Rows[4]["Address"] = "calle Pablo Picasso, 9";
    // añade un nuevo cliente
```

```

fila = tabla.NewRow();
fila["CustomerID"] = "ENIEC";
fila["CompanyName"] = "Eni Escuela Informática";
fila["ContactName"] = "Marcel García";
fila["ContactTitle"] = "Director";
fila["Address"] = "calle Cáceres, 24";
fila["Country"] = "Spain";
fila["City"] = "Madrid";
tabla.Rows.Add(fila);
bldr = new SqlCommandBuilder(da);
da.Update(tabla);
}

```

Comparemos el contenido actual con el anterior de la tabla **Customers** para verificar que, efectivamente, se han tenido en cuenta las modificaciones.

select * from customers where country = 'Spain'

	CustomerID	CompanyName	ContactName	ContactTitle	Address
	BOLID	Bólido Comidas p...	Martín Sommer	Owner	C/ Araquil, 67
►	ENIEC	Eni Escuela Infor...	Marcel García	Director	calle Cáceres, 24
	FISSA	FISSA Fabrica In...	Diego Roel	Accounting Man...	C/ Morazarzal, 86
	GALED	alería del gastr...	Eduardo Saavedra	Marketing Manager	Rambla de Catal...
	GODO5	Godos Cocina Tí...	José Pedro Freyre	Sales Manager	calle Pablo Picaso, 9

b. Utilización de los comandos personalizados

La utilización de los comandos personalizados permite elegir el tipo de acción realizado durante la actualización de la base de datos. Por ejemplo, la eliminación de una fila puede traducirse por la asignación de un valor particular en un campo del registro. En este caso la sentencia **SQL** ejecutada en el `DeleteCommand` será más bien una instrucción **UPDATE** que una **DELETE**.

Por ejemplo, el siguiente código crea un comando personalizado para la eliminación:

```

SqlCommand delCmd;
SqlParameter codigoCliente;
delCmd = new SqlCommand();
delCmd.Connection = ctn;
delCmd.CommandText = " UPDATE Customers set Archive=1 where
                        CustomerID=@num";
codigoCliente = new SqlParameter("@num", SqlDbType.NChar, 5,
                                ParameterDirection.Input, false, 0, 0, "CustomerID",
                                DataRowVersion.Current, null);
delCmd.Parameters.Add(codigoCliente);
da.DeleteCommand = delCmd;
bldr = new SqlCommandBuilder(da);

```



Los comandos personalizados son compatibles con los comandos generados automáticamente por el `SqlCommandBuilder`, ya que éste sólo genera un comando si la propiedad `InsertCommand`, `DeleteCommand` o `UpdateCommand` es igual a **null** en el `DataAdapter`. Si hay un comando, no se reemplaza por el `SqlCommandBuilder`.

c. Gestión de los accesos concurrentes

En un entorno multiusuario hay dos técnicas para gestionar las actualizaciones: el bloqueo optimista y el bloqueo pesimista.

El bloqueo pesimista es el más exigente para los usuarios, ya que, para evitar los conflictos al modificar la base de datos, en cuanto un usuario desea actualizar un registro, éste es bloqueado en la base de datos. Hasta que la modificación del registro esté finalizada, el bloqueo sigue activo y bloquea de ese modo el acceso de otros usuarios. La idea de esta solución es evitar la aparición de conflictos.

El bloqueo optimista no es realmente un bloqueo, ya que los registros están disponibles prácticamente de forma permanente. Es en el momento de la actualización cuando se efectúa una verificación para comprobar si los datos presentes en la base de datos son idénticos a los utilizados para rellenar el `DataSet`. Si son distintos, ello se debe a que otro usuario los modificó entre el llenado del `DataSet` y la petición de actualización de la base de datos. Conviene tomar entonces una decisión relativa a las actualizaciones.

Se pueden considerar tres soluciones:

- Abandonar las actualizaciones.
- Sobrescribir la versión existente.
- Preguntar al usuario lo que desea hacer.

Se puede poner en marcha esta solución con dos técnicas corrientes.

La primera consiste en utilizar en la tabla un campo de tipo **TimeStamp**. La particularidad de este tipo de campo es que tiene un valor que cambia automáticamente con cada modificación efectuada en el registro. Por lo tanto, basta con efectuar una comparación del valor presente en la base con el valor presente en el `DataSet`. Si hay una diferencia, ello se debe a que la base de datos ha sido modificada desde que se cargó el `DataSet`. Para poder considerar esta solución, hace falta que la base de datos sea capaz de gestionar este tipo de campo. También, esta solución aumenta el volumen de los datos, ya que por ejemplo para SQL Server este tipo de campo utiliza 8 bytes.

La segunda solución consiste en generar esto a nivel de la aplicación conservando los datos originales y comparándolos con los datos presentes en la base de datos en el momento de la actualización. Ésta es la solución adoptada por los comandos generados automáticamente por el objeto `SqlCommandBuilder`.

Analicemos el código de una consulta de modificación generada por este objeto.

```
UPDATE [Customers] SET [CustomerID] = @p1, [CompanyName] = @p2, [ContactName] =
@p3, [ContactTitle] = @p4, [Address] = @p5, [City] = @p6, [Region] = @p7,
[PostalCode] = @p8, [Country] = @p9, [Phone] = @p10, [Fax] = @p11 WHERE
((( [CustomerID] = @p12) AND ([CompanyName] = @p13) AND ((@p14 = 1 AND
[ContactName] IS NULL) OR ([ContactName] = @p15)) AND ((@p16 = 1 AND
[ContactTitle] IS NULL) OR ([ContactTitle] = @p17)) AND ((@p18 = 1 AND
[Address] IS NULL) OR ([Address] = @p19)) AND ((@p20 = 1 AND [City] IS NULL) OR
([City] = @p21)) AND ((@p22 = 1 AND [Region] IS NULL) OR ([Region] =
@p23)) AND ((@p24 = 1 AND [PostalCode] IS NULL) OR ([PostalCode] =
@p25)) AND ((@p26 = 1 AND [Country] IS NULL) OR ([Country] =
@p27)) AND ((@p28 = 1 AND [Phone] IS NULL) OR ([Phone] = @p29)) AND ((@p30 = 1 AND
[Fax] IS NULL) OR ([Fax] = @p31)))
```

Este comando utiliza un número impresionante de parámetros: treinta y uno, en realidad. Intentemos explicar el papel de cada uno de estos parámetros.

Se usan los parámetros de @p1 al @p11 para fijar los nuevos valores del registro.

Por su parte, se utiliza el parámetro @p12 para identificar el registro que se ha de actualizar efectuando un test sobre la clave primaria.

Los otros parámetros sólo están presentes aquí para gestionar el bloqueo optimista. Para cada campo se verifica que

los datos presentes en la base de datos son idénticos a los del `DataSet`. Para los campos con restricción de no nulos, la verificación es muy sencilla. Es el caso del campo `CompanyName`, que se compara con el parámetro `@p13`.

Para los campos que no tienen restricción de nulos, hay que hacer uso de un pequeño truco. Sólo es posible comprobar que un campo es nulo en la base de datos, pero no puede haber comparación con el valor **Null**. Para paliar esta limitación, se usan los parámetros `@p14,@p16,@p18,@p20,@p22 @p24,@p26,@p28,@p30` con objeto de representar un **Null** para un campo del `DataSet`. Si su valor es igual a 1, entonces el campo es igual a **Null** en el `DataSet`. Por lo tanto, se combina la comprobación sobre el valor de estos parámetros con el operador `And` para verificar la nulidad simultánea del campo correspondiente en la base de datos. Durante la ejecución del comando se sustituyen los parámetros por los valores presentes en el `DataSet`. En función del parámetro, se utilizan versiones diferentes de la fila.

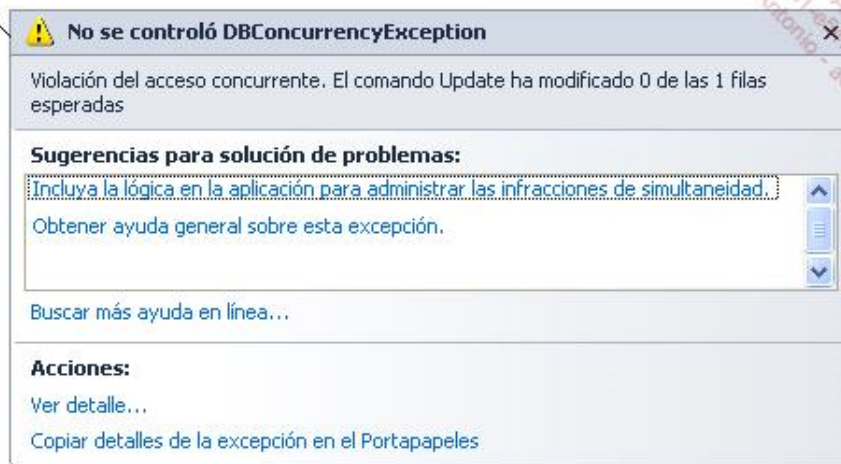
En cuanto a los parámetros del `@p1` al `@p11`, se usan las versiones `Current`. Son los datos que se quiere transferir a la base de datos. Para los restantes parámetros se usan las versiones `Original` de los parámetros.

Se determina la versión de la fila que se va a utilizar en el momento de crear el parámetro antes de su inserción en la colección **Parameters**. Para ello se utiliza el siguiente constructor :

```
public SqlParameter(
    string parameterName,
    SqlDbType dbType,
    int size,
    ParameterDirection direction,
    bool isNullable,
    byte precision,
    byte scale,
    string sourceColumn,
    DataRowVersion sourceVersion,
    Object value
)
```

Permite asociar de forma automática una versión de un campo en particular a un parámetro. Éste es el papel de los argumentos `sourceColumn` y `sourceVersion`. La ejecución del comando devuelve el número de registros actualizados realmente. Si el comando no pudiera actualizar los campos, el valor devuelto por al ejecución del comando sería igual a cero. En este caso se lanza una excepción:

```
da.Update(table);
```



Una solución menos contundente permite vigilar las actualizaciones a lo largo de la ejecución de las instrucciones SQL. Para ello, hace falta gestionar el evento `RowUpdated` del `DataAdapter` que se activa tras la llamada de cada `InsertCommand`, `DeleteCommand`, `UpdateCommand`. El parámetro de tipo **RowUpdatedEventArgs** permite saber, a través de

la propiedad `RecordsAffected`, cuántos registros han sido actualizados.

Si no se ha actualizado ningún registro, puede elegir la acción que se debe ejecutar modificando la propiedad `Status` con uno de los valores de la enumeración `UpdateStatus`:

Continue

La actualización se lleva a cabo como si nada hubiese pasado.

ErrorsOccurred

Se lanza una excepción.

SkipAllRemainingRows

Se detiene la actualización de la fila actual.

SkipCurrentRow

La actualización continúa con las filas restantes.

En principio la decisión sobre el tipo de acción que se ha de realizar debe dejarse en manos del usuario final, como en siguiente ejemplo:

```
private void da_RowUpdated(object sender,
System.Data.SqlClient.SqlRowUpdatedEventArgs e)
{
    string respuesta;
    if (e.RecordsAffected == 0)
    {
        Console.WriteLine("Se ha producido un error durante la actualización
                           de la base de datos");
        Console.WriteLine("Usted desea:");
        Console.WriteLine("1 - continuar con las actualizaciones");
        Console.WriteLine("2 - Cancelar las actualizaciones");
        Console.WriteLine("3 - Cancelar la actualización de esta fila pero
                           seguir con las otras");
        respuesta = Console.ReadLine();
        switch (respuesta)
        {
            case "1":
                e.Status = UpdateStatus.Continue;
                break;
            case "2":
                e.Status = UpdateStatus.SkipAllRemainingRows;
                break;
            case "3":
                e.Status = UpdateStatus.SkipCurrentRow;
                break;
        }
    }
}
```

5. Las transacciones

Las transacciones sirven para agrupar en una entidad un conjunto de comandos SQL. Este agrupamiento va a garantizar que, si alguna de las instrucciones del grupo incluida en la transacción fracasa, la base de datos podrá volver a su estado inicial. El ejemplo clásico es la transferencia de una cuenta bancaria a otra. Imagínese que tiene en su base de datos una tabla para las cuentas de los particulares y otra para las de las empresas.

La transferencia de una cuenta de empresa a una cuenta de particular (el pago de su salario) podría llevarse a cabo con las siguientes instrucciones:

```
public static void TestTransaction ()
{
    SqlCommand cmdPart;
    SqlCommand cmdEmp;
    SqlConnection ctn;
    SqlParameter numParticular;
    SqlParameter numEmpresa;
    SqlParameter importePart;
    SqlParameter importeEmp;
    ctn = new SqlConnection();
    ctn.ConnectionString = "Data Source=localhost;Initial
                          Catalog=Northwind;Integrated Security=true";


    ctn.Open();
    cmdEmp = new SqlCommand();
    cmdEmp.Connection = ctn;
    cmdEmp.CommandText = "Update CuentasEmpresas set saldo=saldo-
                          @importe where numCuenta=@numCuenta";
    numEmpresa = new SqlParameter("@numCuenta", SqlDbType.Int);
    numEmpresa.Value = 1234;
    cmdEmp.Parameters.Add(numEmpresa);
    importeEmp = new SqlParameter("@importe", SqlDbType.Decimal);
    importeEmp.Value = 3000;
    cmdEmp.Parameters.Add(importeEmp);
    cmdEmp.ExecuteNonQuery();
    cmdPart = new SqlCommand();
    cmdPart.Connection = ctn;
    cmdPart.CommandText = "Update CuentasParticulares set
                          sueldo=sueldo+@importe where
                          numCuenta=@numCuenta";
    numParticular = new SqlParameter("@numCuenta", SqlDbType.Int);
    numParticular.Value = 5678;
    cmdPart.Parameters.Add(numParticular);
    importePart = new SqlParameter("@importe", SqlDbType.Decimal);
    importePart.Value = 3000; cmdPart.Parameters.Add(importePart);
    cmdPart.ExecuteNonQuery();
    ctn.Close(); }
```

¿Qué ocurriría si durante la ejecución de este código de repente el servidor de la base de datos estuviese indisponible por culpa de red? La operación de débito podría haberse llegado a ejecutar sin que la de crédito haya hecho lo propio correctamente. Puede haber un riesgo de sufrir un problema importante en el funcionamiento de la aplicación (y en el pago de su propio sueldo). Las transacciones van a permitir hacer frente a este problema agrupando la ejecución de instrucciones SQL para garantizar que estén todas ejecutadas, o bien ninguna.

Se gestionan las transacciones a nivel de conexión. Por lo tanto, es ésta la que nos va a permitir iniciar una transacción. El método `BeginTransaction` nos devuelve una instancia de la clase `SqlTransaction`. Para cada ejecución de comando, podríamos indicar entonces si la ejecución debería ocurrir en el contexto de la transacción o fuera de ella. Al final de la transacción, podemos validar todas las instrucciones que le hemos confiado o, por el contrario, cancelarlas todas. El método `Commit` valida la transacción, mientras que el método `RollBack` la cancela.

Para dar mayor seguridad al código anterior, podríamos utilizar la siguiente versión:

```
public static void TestTransaction2()
{
    SqlCommand cmdPart;
    SqlCommand cmdEmp;
    SqlConnection ctn;
    SqlParameter numParticular;
    SqlParameter numEmpresa;
    SqlParameter importePart;
    SqlParameter importeEmp;
    SqlTransaction trans;
    ctn = new SqlConnection();
    ctn.ConnectionString = "Data Source=localhost;Initial Catalog=Northwind;
Integrated Security=true";
    ctn.Open();
    trans = ctn.BeginTransaction();
    try
    {
        cmdEmp = new SqlCommand();
        cmdEmp.Connection = ctn;
        cmdEmp.CommandText = "Update CuentasEmpresas set saldo=saldo-
                               @importe where numCuenta=@numCuenta";
        numEmpresa = new SqlParameter("@numCuenta", SqlDbType.Int);
        numEmpresa.Value = 1234;
        cmdEmp.Parameters.Add(numEmpresa);
        importeEmp = new SqlParameter("@importe", SqlDbType.Decimal);
        importeEmp.Value = 3000; cmdEmp.Parameters.Add(importeEmp);
        // lugar de ejecución del comando en la transacción
        cmdEmp.Transaction = trans;
        cmdEnt.ExecuteNonQuery();
        cmdPart = new SqlCommand();
        cmdPart.Connection = ctn;
        cmdPart.CommandText = "Update CuentasParticulares set
                               sueldo=sueldo+@importe where
                               numCuenta=@numCuenta";
        numParticular = new SqlParameter("@numCuenta", SqlDbType.Int);
        numParticular.Value = 5678;
        cmdPart.Parameters.Add(numParticular);
        importePart = new SqlParameter("@importe", SqlDbType.Decimal);
        importePart.Value = 3000;
        cmdPart.Parameters.Add(importePart);
        cmdPart.Transaction = trans;
        cmdPart.ExecuteNonQuery();
        trans.Commit();
    }
    catch (Exception ex)
    {
        trans.Rollback();
        Console.WriteLine("Se han cancelado todas las operaciones");
    }
    ctn.Close();
}
```

 Si la conexión se interrumpe, la instrucción `RollBack` o `Commit` no llegará al servidor. En este caso, el servidor toma la iniciativa de ejecutar un `RollBack` sobre todas las transacciones en curso si se pierde la conexión con el cliente.
