Guillermo "Guille" Som

Acceso a datos con ADO.NET 2.0 (sin asistentes)

Tal como comentamos en el número anterior, en esta ocasión vamos a ver cómo podemos acceder a una base de datos usando única y exclusivamente código, es decir, sin usar los asistentes que Visual Studio 2005 pone a nuestra disposición y que, en honor a la verdad, a muchos les facilitará la tarea de crear aplicaciones ADO.NET de una forma bastante sencilla.

» Acceso a datos sin asistentes

Como pudimos comprobar en el número anterior, crear un formulario para acceder a una tabla o a un procedimiento almacenado es muy fácil si nos apoyamos en los asistentes de acceso a datos que Visual Studio 2005 (y con sus limitaciones, las versiones Express) pone a nuestra disposición. Pero algunos programadores de la "vieja escuela" preferimos o nos gusta más escribir código para obtener casi la misma funcionalidad. Por tanto, en esta ocasión vamos a crear un proyecto en el que tendremos la misma funcionalidad que vimos en el número anterior, pero escribiendo todo el código de forma manual. Hacerlo de esta forma no es solo por capricho o masoquismo, sino porque así tendremos una perspectiva diferente que nos permitirá entender mejor cómo realizar las tareas comunes de acceso a datos con ADO.NET 2.0.

Detalles del proyecto que realizaremos usando solo código

Como hemos comentado anteriormente, crearemos un proyecto que nos permitirá hacer, de forma manual, lo mismo que hicimos en el número anterior. Y para facilitar la tarea, vamos a relacionar los pasos que seguiremos para conseguir nuestro propósito, aunque antes resumiremos brevemente qué funcionalidad tendremos en nuestro proyecto.

El proyecto nos permitirá acceder a una base de datos creada con SQL Server 2005 que estará definida en la instancia de `SQLEXPRESS`. Esa base de datos será la misma que usamos en el número anterior (`pruebaGuille`) y que podremos crear usando una utilidad que incluimos en el ZIP con el código de ejemplo. Crearemos dos formularios; uno de ellos nos permitirá mostrar, modificar y añadir nuevos datos, y el formato que usaremos consistirá en mostrar cada campo de la tabla (`Prueba`) en una caja de texto. Además tendremos un segundo formulario en el que mostraremos los datos en un control `DataGridView`; esos datos los obtendremos por medio de un procedimiento almacenado (`StoredProcedure1`) al que le indicaremos la fecha a partir de la que queremos mostrar dichos datos.

Preparativos para realizar la conexión a la base de datos

Para poder acceder a los datos, tendremos que conectarnos a la base de datos que será del tipo SQL Server 2005, creada en la instancia de SQL Express, aunque no necesariamente tiene por qué ser así. Para realizar dicha conexión, tendremos que establecer la cadena de conexión que le indicará a ADO.NET las características de la base de datos a acceder. Dicha cadena será la siguiente:

```
Data Source=(local)\SQLEXPRESS;
     Initial Catalog=pruebaGuille;
     Integrated Security=True
```

Particularmente, tendremos que crear un objeto de la clase `SqlDataAdapter` del espacio de nombres

Guillermo "Guille" Som es Microsoft MVP de Visual Basic desde 1997. Es redactor de dotNetManía, miembro de *Ineta Speakers Bureau Latin America*, mentor de *Solid Quality Learning* Iberoamérica y autor del libro *Manual Imprescindible de Visual Basic .NET*.
http://www.elguille.info

`SqlClient`. Entre otras cosas, a ese objeto le podemos pasar la mencionada cadena de conexión, además de la selección de los datos que queremos manipular. La selección de datos que haremos será la de todas las columnas y todas las filas de la tabla `Prueba`, quedando dicha cadena de esta forma:

```
SELECT * FROM Prueba
```

Esta cadena la podemos adaptar a los datos que queramos manipular, y no solo podemos indicarla como una sentencia "directa" sino que también podemos usar un procedimiento almacenado que hayamos definido en nuestra base de datos. En este primer ejemplo "manual" o sin asistentes usaremos un procedimiento almacenado para seleccionar los datos que se hayan modificado a partir de una fecha. En una próxima ocasión veremos más ejemplos en los que usaremos procedimientos almacenados no solo para mostrar o recuperar los datos, sino también para realizar operaciones de actualización, etc. Incluso crearemos esos procedimientos almacenados usando código de C# (y de VB), pero por ahora es preferible dejar las cosas simples.

La cadena de conexión la vamos a guardar en los datos de configuración de la aplicación. Para ello usaremos la ficha "Configuración" de las propiedades de la aplicación. Tal como podemos ver en la figura 1, en el nombre asignaremos `conexionGuille` y en el ámbito seleccionaremos "Aplicación"; esto hará que se añada a nuestro proyecto un fichero llamado `app.config` que será el que contenga todos los valores añadidos con el ámbito de aplicación; esos valores serán de solo lectura.

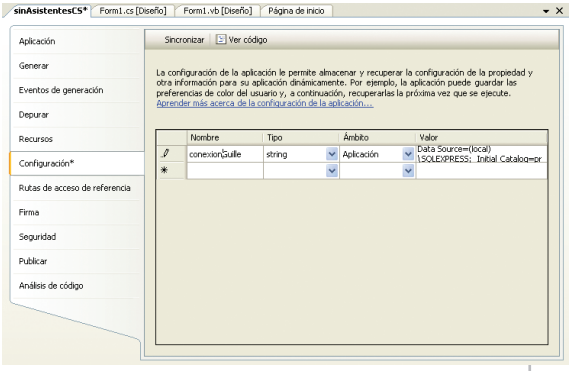

Figura 1. Asignar la cadena de conexión a las propiedades de configuración.

Para acceder a los valores de configuración podemos agregar una importación del espacio de nombres `Properties` que estará incluido en el espacio de nombres de nuestra aplicación. Por ejemplo, si el espacio de nombres predeterminado es `sinAsistentesCS`, tendremos que añadir la siguiente línea al principio de los ficheros de código en los que vayamos a usar esos datos de configuración:

```
using sinAsistentesCS.Properties;
```

Esto nos permitirá acceder a ese valor de la siguiente forma:

```
string cadenaConexion =
  Settings.Default.conexionGuille;
```

Si no añadimos esa importación tendremos que acceder a los valores de configuración indicando el espacio de nombres en el que está definida la clase `Settings`:

```
string cadenaConexion =
  Properties.Settings.Default.conexionGuille;
```

NOTA

En Visual Basic 2005, los datos de configuración están incluidos en `My.Settings`, y podemos acceder a la cadena de conexión usando:

```
Dim cadenaConexion As String =
        My.Settings.conexionGuille
```

Realizar la conexión a la base de datos

Una vez que tenemos preparada la cadena de conexión a la base de datos que queremos usar, es hora de preparar las clases de ADO.NET que nos permitirán realizar la conexión y el acceso real a esos datos. Inicialmente necesitaremos un objeto del tipo `SqlDataAdapter`, que será el que se encargue de conectar a la base de datos. También necesitaremos un objeto de tipo `DataSet` o `DataTable`, el cual recibirá los datos que indiquemos en la cadena de selección. Debido a que solo accederemos a los datos de una tabla, nos resultará más simple usar un objeto del tipo `DataTable`, ya que un objeto `DataSet`, entre otras cosas, puede contener varias tablas, las relaciones entre ellas, etc.; por tanto no lo necesitaremos en esta ocasión.

En el fuente 1 podemos ver el código necesario para realizar la conexión a la base de datos indicada en el parámetro de configuración `conexionGuille` que anteriormente asignamos en las propiedades de la aplicación; esa cadena de conexión la indicamos en el cons-

```
string cadenaConexion =
            Properties.Settings.Default.conexionGuille;
string sel = "SELECT * FROM Prueba";

da = new SqlDataAdapter(sel, cadenaConexion);
dt = new DataTable();

da.Fill(dt);
```

Fuente 1. Creamos los objetos que usaremos para acceder a los datos de la base de SQL Server.

tructor del objeto `SqlDataAdapter`, al que también le indicamos la cadena de selección con idea de que sepa qué datos debe traer de la base de datos para asignarlos al objeto `DataTable`, acción que indicamos mediante el método `Fill` del adaptador.

Crear los comandos de actualización, inserción y eliminación

Si nuestra intención es poder añadir, eliminar y modificar los datos que manipularemos, tenemos que indicar los comandos correspondientes para estas tres acciones. Esos comandos (que no son otra cosa que instrucciones de Transact-SQL), los podemos crear de forma automática por medio de la clase `SqlCommandBuilder`, a cuyo constructor le pasaremos un objeto del tipo `SqlDataAdapter`. Los mencionados comandos se crearán basándose en la cadena de selección que hemos utilizado, pero de esos detalles no tenemos que preocuparnos, ya que lo único que tenemos que hacer es algo como esto:

```
SqlCommandBuilder cb = new
          SqlCommandBuilder(da);
```

Si usamos una tabla en la que tenemos un índice automático, debemos asignar a la propiedad `MissingSchemaAction` del `DataAdapter` el valor `AddWithKey`, de forma que se genere automáticamente el valor del campo autoincremental. La asignación la haremos de esta forma:

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

Dar funcionalidad a la aplicación

Una vez que tenemos los datos en el objeto `DataTable`, ya solo nos queda mostrar los datos, añadir nuevos, eliminar y actualizar los cambios y, cuando creamos conveniente, guardar esos cambios en la base de datos.

Debido a que no estamos usando un asistente, será nuestro trabajo la creación completa de la interfaz del usuario, por tanto tendremos que agregar los controles en los que mostraremos los datos, así como los botones que nos permitirán navegar por los registros de la tabla que vamos a utilizar.

Nuestro formulario de prueba tendrá el aspecto mostrado en la figura 2, que como podemos comprobar es muy parecido al que el propio asistente de Visual Studio 2005 crearía. Aunque en esta aplicación de ejemplo hay ciertos cambios, y no nos referimos solo a cambios en las imágenes usadas, o en el número de botones.

En esta aplicación usaremos un control `CheckBox` para indicar si queremos asignar automáticamente los cambios que realicemos a los datos de la tabla; de esta forma, cada vez que nos desplacemos entre los registros, se guardarán los cambios que hayamos hecho. Pero si no queremos disponer de esta funcionalidad "auto-asignadora", podemos quitar la marca de ese control y será responsabilidad nuestra si guardamos o no los cambios que realicemos. Para ese caso, tenemos el botón de actualizar (la imagen con un disquete). Por supuesto, esas asignaciones se harán solo si los datos realmente han cambiado, comprobación que hacemos cada vez que cambia el contenido de las cajas de textos o el control `DateTimePicker`. Cuando los eventos de cambio de esos controles se disparan, lo que hacemos es comprobar si realmente el contenido ha cambiado (ver el fuente 2), de forma que si se vuelve a dejar el mismo contenido que había antes, el registro no debe marcarse como modificado.

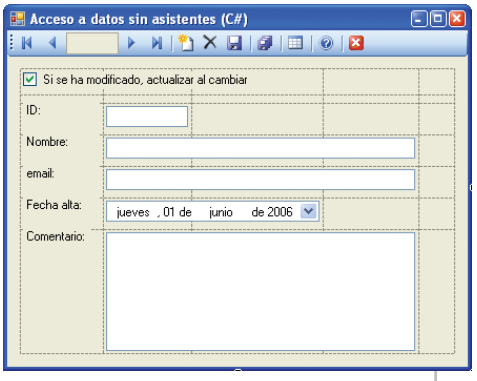

Figura 2. El formulario principal en modo de diseño

```
private bool comprobarCambiado()
{
    bool cambiado = false;

    if( txtNombre.Text != dt.Rows[filaActual]["Nombre"].ToString())
        cambiado = true;
    if( txtEmail.Text != dt.Rows[filaActual]["e-mail"].ToString() )
        cambiado = true;
    if( Convert.ToDateTime( txtFechaAlta.Text).Date.CompareTo(
      Convert.ToDateTime( dt.Rows[filaActual]["FechaAlta"]).Date) != 0 )
        cambiado = true;
    if( txtComentario.Text != dt.Rows[filaActual]["Comentario"].ToString())
        cambiado = true;

    return cambiado;
}
```

Fuente 2. La función que comprueba si se ha modificado el contenido de las cajas de texto usadas para introducir los datos

Moverse por los registros de la tabla, actualizando si es necesario

Como podemos comprobar en el código del fuente 2, tenemos una variable (`filaActual`) que controla la fila o registro con el que estamos trabajando en cada momento, y es por medio de esa variable con la que

navegamos usando los primeros cuatro botones de la barra de herramientas. En cada método de evento de esos botones hacemos la asignación correspondiente para mantener adecuadamente el valor de la fila actual, y después simplemente llamamos a un método que será el que se encargue de mostrar la información en las cajas de texto correspondientes. En ese mismo método (que podemos ver en el fuente 3), hacemos ciertas comprobaciones adicionales, con idea de que el valor de la fila actual esté en el rango adecuado, además de comprobar si tenemos que guardar los datos o no.

```
private void btnPrimero_Click(object sender, EventArgs e)
{
    filaActual = 0;
    mostrarDatos();
}

private void btnAnterior_Click(object sender, EventArgs e)
{
    filaActual–;
    mostrarDatos();
}

private void btnSiguiente_Click(object sender, EventArgs e)
{
    filaActual++;
    mostrarDatos();
}

private void btnUltimo_Click(object sender, EventArgs e)
{
    filaActual = dt.Rows.Count - 1;
    mostrarDatos();
}

private void mostrarDatos()
{
    if( dt.Rows.Count < 1 )
        return;

    if( filaActual < 0 )
        filaActual = 0;

    if( filaActual >= dt.Rows.Count - 1 )
        filaActual = dt.Rows.Count - 1;

    if(dt.Rows[filaActual].RowState == DataRowState.Deleted)
    {
        // TODO: Comprobar a qué fila podemos desplazarnos
        return;
    }

    comprobarActualizar();
    filaAnt = filaActual;

    txtID.Text = dt.Rows[filaActual]["ID"].ToString();
    txtNombre.Text = dt.Rows[filaActual]["Nombre"].ToString();
    txtEmail.Text = dt.Rows[filaActual]["e-mail"].ToString();
    txtFechaAlta.Text = dt.Rows[filaActual]["FechaAlta"].ToString();
    txtComentario.Text = dt.Rows[filaActual]["Comentario"].ToString();

    txtActual.Text = (filaActual + 1).ToString();
    modificado = false;
}
```

Fuente 3. Los métodos de navegación y el encargado de mostrar los datos en los controles

Como podemos comprobar, será en el método `mostrarDatos` en el que tendremos que escribir el código necesario para mostrar los datos en los controles. Como es de suponer, si la fila que intentamos mostrar está eliminada, simplemente salimos sin mostrar nada, aunque debemos comprobar a qué fila debemos desplazarnos; esa parte la veremos a la hora de eliminar los registros. En ese mismo método asignamos el valor a otra variable (`filaAnt`) que nos será de utilidad en el caso de que haya cambiado el contenido de los controles, y la usaremos para indicar en qué fila debemos hacer la asignación de dichos cambios. Esta comprobación la hacemos en el método `comprobarActualizar`, desde el que llamamos al método `actualizarDatos`, que recibe como parámetro la fila que hay que actualizar, tal como podemos comprobar en el fuente 4.

```
private void comprobarActualizar()
{
    if(chkActualizarAuto.Checked && modificado &&
       (filaActual != filaAnt))
        actualizarDatos(filaAnt);
}

private void actualizarDatos(int fila)
{
    dt.Rows[fila]["Nombre"] = txtNombre.Text;
    dt.Rows[fila]["e-mail"] = txtEmail.Text;
    dt.Rows[fila]["FechaAlta"] =
              Convert.ToDateTime(txtFechaAlta.Text);
    dt.Rows[fila]["Comentario"] = txtComentario.Text;

    modificado = false;
    btnGuardarEnBase.Enabled = true;
}
```

Fuente 4. El código para asignar los cambios a la fila correspondiente y el que comprueba si se deben actualizar los datos

Como vemos, también hay un solo método en el que hacemos todo lo necesario para asignar los cambios a la fila que se ha modificado. El método `actualizarDatos` recibe un parámetro con el índice de la fila modificada porque nos servirá también en caso de que pulsemos en el botón "Actualizar", aunque en ese caso, el registro que debemos asignar es el de la fila actual (ver fuente 5). Por esa razón, en vez de usar una de las variables "globales" hemos optado por usar un parámetro.

```
private void btnActualizar_Click(object sender,
EventArgs e)
{
    actualizarDatos(filaActual);
}
```

Fuente 5. El método del evento del botón para actualizar

Eliminar y crear nuevos registros

Las otras dos acciones que utilizaremos para eliminar datos o para crear nuevos en principio no tienen

mayores complicaciones; al menos la de crear nuevos datos, ya que simplemente añadiremos una nueva fila a la colección de filas de la tabla. Esa acción, tal como vemos en el fuente 6, consiste en obtener una nueva fila (`DataRow`) que nos proporciona el objeto `DataTable`, asignar los valores predeterminados que creamos conveniente y finalmente añadirla a la colección `Rows`.

```
private void btnNuevo_Click(object sender, EventArgs e)
{
    // Comprobar si los datos actuales hay que
guardarlos automáticamente
    comprobarActualizar();

    DataRow dr = dt.NewRow();
    dr["Nombre"] = "Nuevo";
    dr["FechaAlta"] = DateTime.Now;
    dt.Rows.Add(dr);
    btnGuardarEnBase.Enabled = true;
    habilitarBotones();
    btnUltimo.PerformClick();
}
```

Fuente 6. El código para crear nuevos registros

A la hora de eliminar registros tampoco tendremos que hacer demasiado, ya que como vemos en el fuente 7, solo tenemos que llamar al método `Delete` de la fila actual.

```
private void btnBorrar_Click(object sender, EventArgs e)
{
    // Eliminar la fila actual
    // Comprobar si los datos actuales hay que
guardarlos automáticamente
    comprobarActualizar();
    // Confirmar el borrado
    if( MessageBox.Show("...") == DialogResult.Yes )
    {
        dt.Rows[filaActual].Delete();

        btnGuardarEnBase.Enabled = true;

        modificado = false;
        btnSiguiente.PerformClick();
    }
}
```

Fuente 7. El método para eliminar registros

Como vemos, el problema de eliminar registros no está en la forma de eliminarlos, sino a la hora de navegar entre los registros, ya que es posible que nos encontremos con un registro eliminado, en cuyo caso no debemos mostrarlo. En realidad, el problema nos lo podremos encontrar cuando queramos navegar al primero o al último y éstos estén eliminados, o incluso si hay varios seguidos que están eliminados. Por tanto, en el método `mostrarDatos` debemos tener en cuenta estas posibilidades y no "intentar" mostrar los datos de los registros eliminados.

```
if( dt.Rows[filaActual].RowState == DataRowState.Deleted )
{
  // Comprobar a qué fila podemos desplazarnos
  // Si la primera está borrada, buscar la siguiente que no lo esté
  if( filaActual == 0 )
  {
      for( int i = filaActual + 1; i < dt.Rows.Count; i++ )
      {
          if( dt.Rows[filaActual].RowState == DataRowState.Deleted )
          {
              filaActual++;
          }
          else
          {
              filaActual--;
              break;
          }
      }
      btnSiguiente.PerformClick();
      return;
  }
  else
  {
      // Si la última está borrada, buscar la anterior que no lo esté
      if( filaActual == dt.Rows.Count - 1 )
      {
          for( int i = filaActual - 1; i >= 0; i-- )
          {
             if( dt.Rows[filaActual].RowState == DataRowState.Deleted )
             {
                 filaActual--;
             }
             else
             {
                 filaActual++;
                 break;
             }
          }
          btnAnterior.PerformClick();
          return;
      }
  }

  if( (filaActual - filaAnt) >= 0 )
      btnSiguiente.PerformClick();
  else
      btnAnterior.PerformClick();

  return;
}
```

Fuente 8. Comprobación a incluir en el método mostrarDatos, de qué fila está disponible cuando hemos eliminado algunas

Para que todo funcione correctamente, debemos cambiar la comprobación de si la fila está eliminada que mostramos en el fuente 3 por la del código fuente 8, de esa forma no entraremos en un bucle sin fin que finalmente resultaría en un error de desbordamiento de la pila (*stack overflow*).

Guardar los cambios realizados en la base de datos

La forma en que estamos tratando los datos es de forma desconectada, es decir, esos datos los estamos mani-

pulando en nuestro equipo, pero los cambios no se están reflejando en la base de datos. Por tanto, si queremos que todos esos cambios se hagan efectivos en la base de datos, debemos pulsar en el botón con el icono de varios disquetes. Desde el método gestor de ese evento le indicaremos al `DataAdapter` que use las instrucciones de actualización, eliminación o inserción que correspondan, lo cual logramos llamando al método `Update` del objeto `SqlDataAdapter` pasándole como parámetro la tabla que contiene las filas modificadas, y finalmente le indicaremos a la tabla que acepte los cambios. Todo esto lo hacemos por medio de dos líneas de código, tal como podemos ver en el fuente 9.

```
private void btnGuardarEnBase_Click(object sender, EventArgs e)
{
  try
  {
      // Comprobar si los datos actuales hay que guardarlos automáticamente
      comprobarActualizar();

      da.Update(dt);
      dt.AcceptChanges();

      btnGuardarEnBase.Enabled = false;
  }
  catch(Exception ex)
  {
      MessageBox.Show("Error al guardar los datos en la base\n" +
                      ex.Message, "Error");
  }
}
```

Fuente 9. Debemos guardar en la base de datos los cambios realizados localmente

Mostrar los datos a partir de una consulta realizada con un procedimiento almacenado

Por razones obvias de espacio no podremos ver con detalle la parte que nos falta de este proyecto: mostrar los datos por medio de una consulta usando un procedimiento almacenado de SQL Server; pero al menos veremos el código que tendremos que usar, que ejecutaremos desde otro formulario (ver figura 3) en el que tendremos un control `DataGridView` y los controles para navegar entre los registros mostrados, además de la caja de texto en la que indicaremos la fecha a usar para la consulta y el botón "Buscar" que será el encargado de asignar al grid los datos a mostrar.

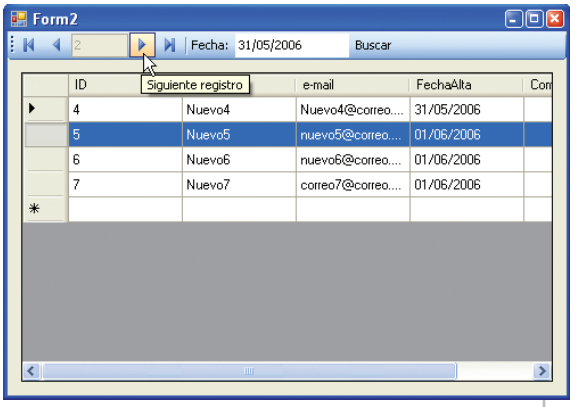

Figura 3. El formulario usado para mostrar los datos a partir de una fecha

```
private void btnBuscar_Click(object sender, EventArgs e)
{
 string cadenaConexion = Properties.Settings.Default.conexionGuille;
 try
 {
  SqlDataAdapter da = new SqlDataAdapter("StoredProcedure1",
                                         cadenaConexion);
  da.SelectCommand.CommandType = CommandType.StoredProcedure;
  da.SelectCommand.Parameters.Add("@Param1", SqlDbType.DateTime);
  da.SelectCommand.Parameters["@Param1"].Value =
                                 Convert.ToDateTime(txtFecha.Text);

  DataTable dt = new DataTable();
  da.Fill(dt);

  if( dt.Rows.Count > 0 )
  {
      dgvDatos.DataSource = dt;
      habilitarBotones(true);
  }
  else
  {
      dgvDatos.DataSource = null;
      habilitarBotones(false);
  }
  filaActual = filaAnt = 0;
  mostrarDatos();

 }
 catch(Exception ex)
 {
    MessageBox.Show("Error\n\n" + ex.Message, "Error");
 }
}
```

Fuente 10. El método para buscar datos usando un procedimiento almacenado

En el fuente 10 tenemos el código de ese método de búsqueda.

Conclusiones

Como hemos visto, no es tan complicado crear código para acceder a datos sin apoyarnos en los asistentes de Visual Studio 2005, y la ventaja es que sobre este código tenemos más control que en el caso del código generado de forma automática. Sí, tiene un poco de más trabajo, pero en realidad lo más laborioso es el tema del diseño del formulario de introducción de datos, el cual aunque lo generara el propio Visual Studio con sus asistentes seguramente acabaríamos modificando para adaptarlo a nuestro gusto. Pero gracias a los nuevos controles y nuevas ayudas del diseñador de formularios de Visual Studio 2005, esto será una tarea relativamente fácil.

Como siempre, en el ZIP con el código completo del ejemplo usado en este artículo encontrarás tanto la versión para C# como para Visual Basic 2005.