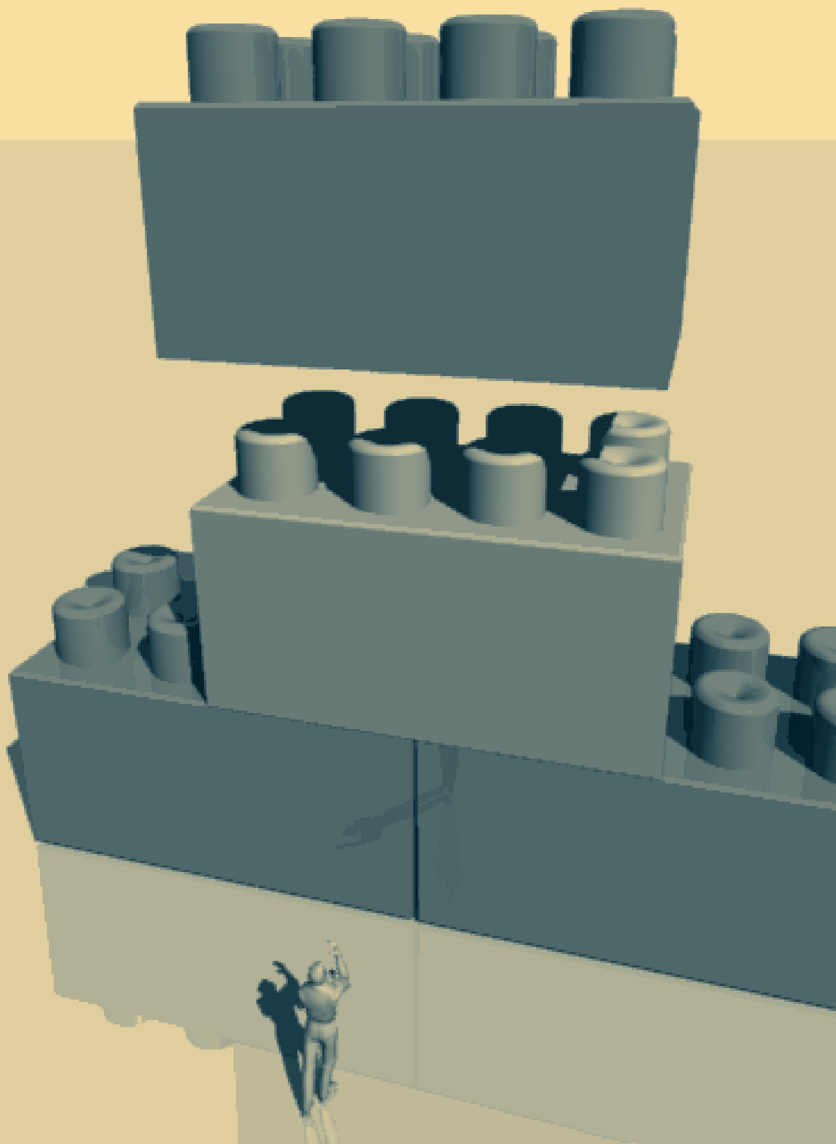
Capítulo 2
C#



EL HIJO PRÓDIGO DE MICROSOFT ES UNO DE LOS FAVORITOS PARA DESARROLLO DE APLICACIONES WEB, WINDOWS Y MOBILE. EN ESTA SECCIÓN SE PRESENTAN PRÁCTICAS INTERESANTES PARA REALIZAR EN ESTA PLATAFORMA, APLICABLES A CUALQUIER PROBLEMÁTICA DEL MUNDO REAL.

DESARROLLOS EXTENSIBLES CON C#

Creación de aplicaciones con soporte de plugins

Veremos algunas técnicas que brinda el Framework .NET para escribir aplicaciones capaces de ser extendidas sin recompilar.

Actualmente, muchas aplicaciones permiten extender su funcionalidad mediante el uso de plugins, es decir, agregados que le otorgan funcionalidad extra una vez que está terminada y puesta en producción, sin necesidad de tocar y volver a compilar el código. Desde el punto de vista del usuario final, esta posibilidad representa una interesante ventaja, ya que le permite adaptar la aplicación a sus necesidades sin depender de nuestra intervención (ideal para programas comerciales de tipo "enlatados").
También tiene ventajas para los desarrolladores de la aplicación, ya que les permite implementar y distribuir mejoras y agregados de una manera realmente muy sencilla.

Para ver en detalle todos los conceptos relacionados a la escritura de aplicaciones con soporte para plugins, vamos a desarrollar un pequeño y simple editor de texto, junto con un plugin que agregue la funcionalidad para contar las palabras que tiene.

¿Qué necesitamos?

Para que nuestra aplicación soporte el agregado de plugins, debemos tener en cuenta dos aspectos fundamentales: primero, los plugins que se desarrollen deben tener acceso a elementos de la aplicación sobre los cuales trabajar; segundo, el código del plugin debe poder determinarse en tiempo de ejecución de la aplicación (no, en tiempo de compilación), es decir que la aplicación debe estar totalmente desacoplada de los plugins que se escriban en el futuro. Para lograrlo, vamos a valernos del uso de las interfaces y el *late bining* (enlace tardío), conceptos que explicaremos a continuación.

Las interfaces

Para q ue el plugin pueda conocer algunos elementos de la aplicación y actúe sobre ellos, vamos a escribir una interfaz que abstraiga las características de un archivo de texto, y otra para el editor de texto propiamente dicho. Con el objetivo de que el plugin pueda ser compilado independientemente de la aplicación, colocaremos dichas interfaces en un assembly separado. Usando Visual Studio .NET, creamos un nuevo proyecto de tipo Biblioteca de Clases, agregamos una nueva clase (cuyo archivo llamaremos IArchivoTexto.cs) y reemplazamos el código generado por el asistente, por este otro:

```
public interface
    IArchivoTexto
{
    string Nombre{ get; set;}
    string Texto{ get; set; }
    bool Grabado{ get; set; }

    void Grabar();
}
```

Esta interfaz representa las características mínimas de un archivo de texto simple. Como dijimos al principio, el plugin necesita conocer elementos de la aplicación y, también, mostrar mensajes al usuario. Para hacerlo, vamos a escribir una interfaz que exponga el archivo de texto con el que actualmente se está trabajando en la aplicación, y un método para que el plugin muestre mensajes al usuario.

```
public interface IEditor
{
    IArchivoTexto
        Archivo{ get; }
    void MostrarMensaje( string
        mensaje, string titulo );
}
```

En principio, puede parecer que el método MostrarMensaje no es necesario, ya que dentro del código del plugin podemos hacer una invocación a MessageBox.Show con el fin de mostrar el mensaje. Sin embargo, ésta no es una buena idea, ya que el plugin queda limitado a aplicaciones Winforms. Si quisiéramos, luego, utilizarlo en una aplicación de ASP.NET, tendríamos problemas. Mediante el uso de un método, la implementación final del mecanismo para mostrar el mensaje depende de la aplicación que utiliza el plugin.

Bien, ahora que ya tenemos las interfaces necesarias para comunicar la aplicación con el plugin, podemos escribir la que se deberá implementar para crear un plugin destinado a nuestra aplicación. Como éste debe ser accesible desde alguna opción del menú, la implementación de cada uno debe indicarle a la aplicación qué texto mostrar en la interfaz con el usuario para acceder a su funcionalidad. Esto lo haremos mediante una propiedad que llamaremos TextoMenu. Para facilitar la configuración (como veremos más adelante), vamos a exigir que el plugin sea capaz de informar un nombre que lo distinga entre los demás. Por último, necesitamos pasarle al plugin el contexto del editor en el cual ejecutarse (usando la interfaz IEditor) y un método para invocar la función provista por el plugin. La interfaz IPlugin queda, entonces, de esta forma:

```
public interface IPlugin
{
    string Nombre{ get; }
    string TextoMenu{ get; }
    IEditor ContextoEditor
        { get; set; }

    void Ejecutar();
}
```

Escribir la aplicación

Ya tenemos las interfaces definidas; vamos a comenzar ahora a escribir nuestra aplicación. Usando Visual Studio .NET, creamos un proyecto de tipo Aplicación para Windows y agregamos una referencia al que generamos antes. Lo primero que tenemos que hacer es armar la interfaz general de la aplicación. Agregamos un componente MainMenu y creamos dos opciones de menú de nivel superior: Archivo (mnuArchivo) y Plugins (mnuPlugins). Es importante darle al

menú de plugins un nombre que recordemos, ya que deberemos referenciarlo más adelante. Agregamos también un TextBox y le establecemos las propiedades MultiLine en true y Dock en Fill. A esta altura, si ejecutamos el proyecto, la aplicación se verá como en la [Figura 1].

Luego, vamos a definir una clase que implemente la interfaz IArchivoTexto:

```
public class ArchivoTexto:
    IArchivoTexto
{
    public ArchivoTexto()
    {
        texto = string.Empty;
        nombre = string.Empty;
    }

    private bool grabado;
    private string nombre;
    private string texto;

    public string Nombre
    {
        get{ return nombre; }
        set{ nombre = value; }
    }

    public string Texto
    {
        get{ return texto; }
        set{ texto = value; }
    }

    public bool Grabado
    {
        get{ return grabado; }
        set{ grabado = value; }
    }

    public void Grabar()
    { // TODO: agregar
        la implementación
        ArchivoTexto.Grabar
    }
}
```

Como vemos, el texto del archivo, simplemente, será una variable de tipo string.

Para no complicar demasiado el ejemplo, vamos a hacer que el formulario principal de la aplicación implemente la interfaz IEditor, y que contenga todos los elementos necesarios para pasarles a los plugins instalados. Para que el formulario implemente la interfaz, la agregamos a su definición:

```
public class FormEditor : Form, IEditor
```

Como la interfaz IEditor expone una propiedad de tipo IArchivoTexto, declaramos un variable de tipo ArchivoTexto, privada al formulario, e implementamos la propiedad correspondiente a la interfaz:

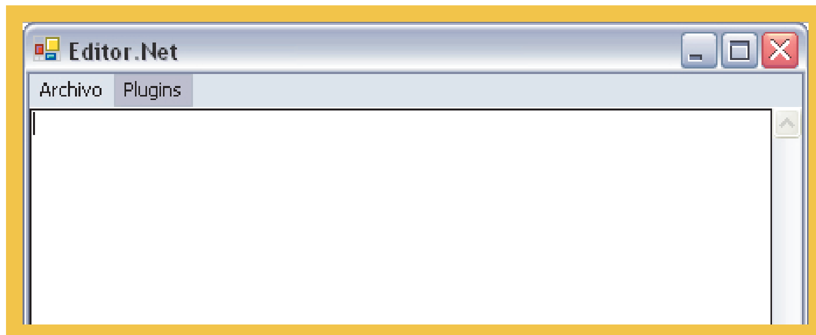


Figura 1. **Nuestra aplicación base a la cual se le podran agregar plugins**

```
private ArchivoTexto archivo =
  new ArchivoTexto();

public IArchivoTexto Archivo
{
  get{ return archivo; }
}
```

También debemos implementar el método MostrarMensaje. Como en este caso se trata de una aplicación Winforms, usamos el clásico MessageBox, con un botón Aceptar y un icono de información:

```
public void MostrarMensaje(string
  mensaje, string titulo)
{
  MessageBox.Show(mensaje, titulo,
      MessageBoxButtons.OK,
      MessageBoxIcon.Information);
}
```

Para que el texto del archivo se actualice con las modificaciones que reraliza el usuario del editor, agregamos un manejador del evento TextChanged del textbox que actualice el texto y marque el archivo como modificado:

```
private void txtTexto_TextChanged
  (object sender, System.EventArgs e)
{
  archivo.Grabado = false;
  archivo.Texto = txtTexto.Text;
}
```

El plugin

Como la aplicación está casi lista, vamos a enfocarnos en la escritura del plugin que, más tarde, agregaremos a la aplicación. Como dijimos, el plugin debe ser totalmente independiente de ésta, por lo cual lo colocaremos en un ensamblado separado, agregando un nuevo proyecto de tipo Biblioteca de Clases a nuestra solución. En él hacemos referencia al ensamblado que creamos al principio y agregamos una nueva clase que implemente la interfaz IPlugin:

```
public class ContadorPalabras:
IPlugin
{
    public ContadorPalabras()
    {}

    private IEditor editor;
    public IEditor ContextoEditor
    {
        get{ return editor; }
        set{ editor = value; }
    }
    public string Nombre
    {
        get{ return
            "ContadorPalabras"; }
    }
    public string TextoMenu
    {
        get{  return "Contar
            Palabras";
        }
    }

}
```

Como ven en el código, la clase implementa las tres propiedades de la interfaz. Como nombre usaremos “ContadorPalabras”, y en la opción del menú de la aplicación, aparecerá el texto “Contar Palabras”. Si recordamos el código de la interfaz IPlugin, veremos que a nuestra nueva clase le falta el método Ejecutar. La implementación de este método en el plugin tomará el texto del archivo contenido en el contexto del editor (que recibe mediante la propiedad ConextoEditor) y, usando el método Split de la clase String, obtendrá un arreglo con todas las palabras. Por último, mediante el método MostrarMensaje, informará al usuario la cantidad de palabras que contiene el texto que está escribiendo en el editor:

```
public void Ejecutar()
{
```

```
if( editor == null )
  throw new NullReferenceException
  ("No se ha establecido el
  contexto donde se aplica
  el plugin");
  if( editor.Archivo == null )
     throw new NullReferenceException("No se ha
        establecido el archivo para trabajar");

  int cantidadPalabras = 0;
  if( editor.Archivo.Texto.Trim().Length > 0 )
  {
     string[] palabras = editor.Archivo.Texto.Split(' ');
     cantidadPalabras = palabras.Length;
  }
  editor.MostrarMensaje(string.Format("Cantidad de
     palabras: {0}", cantidadPalabras), "Contador
     de Palabras");
}
```

Ya tenemos listo el editor de texto y, también, nuestro primer plugin, pero aún no hay forma de indicarle a la aplicación qué plugins queremos agregar.

Configuración de plugins

Para que la aplicación reconozca los plugins, vamos a agregar una sección al archivo de configuración, donde indicaremos los nombres y la ubicación de cada uno de los que queremos utilizar. Antes de continuar, agregaremos un archivo de configuración a nuestra aplicación principal, utilizando el asistente de Visual Studio, que incorporará al proyecto un archivo llamado App.Config.

Por predefinición, los valores que podemos agregar al archivo de configuración son pares de la forma llave y valor, que deben ir ubicados dentro del tag <appSettings>. Para nuestro caso, esta opción no es suficiente, ya que tenemos más datos para configurar. Afortunadamente, el framework .NET nos provee de un mecanismo por el cual podemos escribir nuestras propias secciones dentro del archivo de configuración, y nos ofrece una clase que se encarga de leer esa sección y "traducirla" para que la aplicación pueda utilizar esos datos de configuración. La forma de hacerlo es implementando la interfaz System.Configuration.IConfigurationSectionHandler. Esta expone un único método que recibe un nodo XML con el elemento de nuestra sección personalizada del archivo de configuración, y debe devolver un objeto que represente los datos de configuración.

Entonces, debemos escribir una clase que represente los datos de la configuración. Para simplificar el ejemplo, la nuestra sólo tendrá una lista de los plugins instalados:

```
  public class ConfiguracionPlugins
  {
     public ConfiguracionPlugins()
     {}

     private ArrayList pluginsInstalados =
        new ArrayList();
  public ArrayList PluginsInstalados
     {
        get{ return pluginsInstalados; }
     }
  }
```

Definición

Late binding (o "enlace tardío") es la habilidad de crear instancias de objetos cuyo tipo real no se conoce en tiempo de compilación, pero sí, en tiempo de ejecución. La técnica de late binding permite utilizar objetos creándolos a partir del nombre de la clase al que pertenecen, lo cual dota a las aplicaciones de una gran flexibilidad. Cuando el tipo real del objeto se conoce en tiempo de compilación, se trata de un "early binding" (o "enlace temprano").

Ahora que ya tenemos definida la clase para acceder a la configuración, debemos escribir clase que se encargue de leer el archivo de configuración xml y de crear el objeto correspondiente. Agregamos una nueva clase a nuestro proyecto, la llamamos ConfiguracionPluginsSectionHandler y modificamos su declaración para indicar que implementa la interfaz IConfigurationSectionHandler. Como vimos antes, esta interfaz expone un solo método denominado Create, que recibe tres parámetros, de los cuales nos interesa únicamente el último, que, al momento de la invocación del método, contiene el nodo XML correspondiente a nuestra sección personalizada.

El formato de la sección

Como vamos a tener que manipular un poco el documento XML, antes de escribir el código debemos tener bien definido el formato de nuestra sección. Queremos que nuestra aplicación pueda tener muchos plugins instalados, de modo que vamos a crear un elemento XML llamado <Plugins>, que podrá contener a uno o más elementos <plugin>. El elemento <plugin> tendrá atributos para indicar el nombre de cada uno, el ensamblado donde se encuentra y el nombre de la clase que lo implementa. Nuestra sección personalizada quedará de la siguiente manera:

```
<Plugins>
  <Plugin nombre="ContadorPalabras"
     ensamblado="MisPlugins"
     clase="MisPlugins.ContadorPalabras" />
</Plugins>
```

Pasamos, entonces, a escribir el código del método Create de nuestra clase ConfiguracionPluginsSectionHandler. Mediante una consulta XPath, vamos a conseguir la lista de nodos <Plugin> y, para cada uno detectado, obtendremos sus atributos, que utilizaremos para crear, mediante *late binding* y *relection* una instancia de un objeto IPlugin y agregarlo a la lista de plugins del objeto Configuracion. Para poder usar reflection, debemos agregar a nuestra clase una cláusula using System.Reflection. El código del método Create queda así:

```
public object Create(object parent, object
  configContext, System.Xml.XmlNode section)
{
  ConfiguracionPlugins config =
     new ConfiguracionPlugins();
  XmlNodeList plugins = section.SelectNodes
     ("./Plugin");
  foreach(XmlNode nodo in plugins)
  {
     string nombre = nodo.Attributes["nombre"].Value;
     string ensamblado =
        nodo.Attributes["ensamblado"].Value;
     string clase = nodo.Attributes["clase"].Value;
```

```
        Assembly assembly = Assembly.Load(ensamblado);
                IPlugin plugin = (IPlugin)assembly.
                  CreateInstance( clase );
        config.PluginsInstalados.Add(plugin);
    }

    return config;
  }
```

Ya estamos muy cerca de lograr que nuestra aplicación soporte plugins. Para que el CLR sepa qué clase debe usar para leer una sección personalizada del archivo de configuración, debemos indicárselo. Para hacerlo, agregamos al archivo de configuración de la aplicación una sección denominada <configSettings>, dentro de la cual especificamos el nombre de nuestra sección y el tipo del objeto que se encarga de leerla y "traducirla" para que la utilice la aplicación. Es importante que la sección <configSettings> esté antes que cualquier otra, porque de lo contrario, obtendremos un error de compilación. Ahora el archivo App.config queda así:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
   <section name="Plugins" type="EditorNet.
      ConfiguracionPluginsSectionHandler, EditorNet" />
  </configSections>

  <Plugins>
      <Plugin nombre="ContadorPalabras" ensamblado=
         "MisPlugins" clase="MisPlugins.ContadorPalabras" />
  </Plugins>
</configuration>
```

Instalación del plugin

Para que el usuario de la aplicación pueda utilizar los plugins, éstos deben ser accesibles desde la interfaz gráfica. Con este objetivo, vamos a crear dinámicamente, durante el arranque del programa, nuevas opciones en el menú principal. Escribimos un método al que podemos llamar "InstalarPlugins", en el que obtenemos la lista de plugins utilizando el método ConfigurationSettings.GetConfig (aquí es donde el CLR invoca a nuestra clase ConfiguracionPluginsSectionHandler, tal como se lo indicamos en el archivo de configuración), y para cada plugin configurado, creamos un MenuItem y lo agregamos al menú Plugins. Como necesitamos proveer de un manejador del evento Click del menú para invocar al plugin, vamos a utilizar un único Event Handler y nos valdremos del parámetro sender, que nos informa cuál es el control que generó el evento. Para mantener la relación entre opciones de menú y plugins, utilizamos una tabla hash, donde iremos colocando los MenuItem y los Plugins, usando el menú como llave:

```
private Hashtable pluginsMenu = new Hashtable();

private void InstalarPlugins()
{
  ConfiguracionPlugins plugins = ConfigurationSettings.
      GetConfig("Plugins") as ConfiguracionPlugins;
  foreach(IPlugin plugin in plugins.PluginsInstalados)
  {
      MenuItem menu = new MenuItem();
      menu.Text = plugin.TextoMenu;
      menu.Click += new EventHandler(ClickMenuPlugin);
```

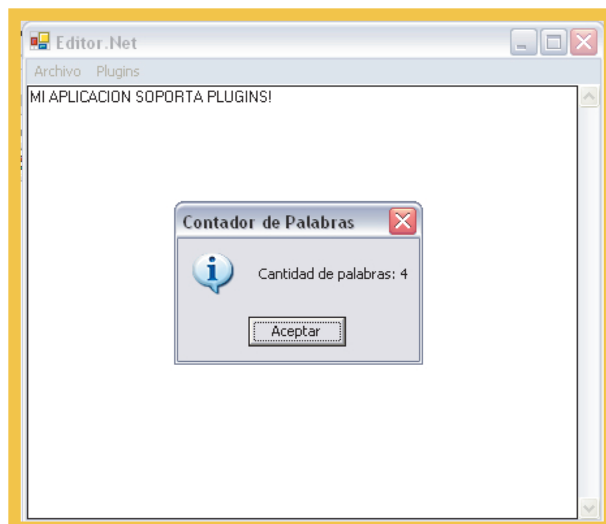


Figura 2. Funcionamiento del plugin creado.

```
      pluginsMenu.Add(menu, plugin);
      mnuPlugins.MenuItems.Add( menu );
   }
 }

 private void ClickMenuPlugin(object sender, EventArgs e)
 {
   MenuItem menu = (MenuItem)sender;
   IPlugin plugin = (IPlugin)pluginsMenu[menu];
   try
   {
      plugin.ContextoEditor = this;
      plugin.Ejecutar();
   }
   catch(Exception ex)
   {
      MessageBox.Show("El plugin a generado una
         excepción con el mensaje: " + ex.Message,
         "Error en Plugin", MessageBoxButtons.OK,
         MessageBoxIcon.Error);
   }
 }
```

Agregamos un manejador del evento Load del formulario y ahí escribimos una llamada al método InstalarPlugins, para que se haga cuando la aplicación comienza. Si ahora ejecutamos la aplicación, veremos que, bajo el menú Plugins, hay una nueva opción con el texto "Contar Palabras". Si escribimos algo en el editor y hacemos clic en el menú, obtendremos un mensaje con la cantidad de términos, tal como se puede ver en la [Figura 2]. Un último detalle: si al momento de ejecutar el plugin nos da un error, debemos copiar el ensamblado MisPlugins.dll a la carpeta donde está el ejecutable de la aplicación.

Conclusiones

A lo largo de este artículo, vimos los pasos necesarios para construir una aplicación cuya funcionalidad pueda extenderse mediante el agregado de plugins. Lo más importante es analizar con sumo detalle qué elementos de la aplicación principal queremos que sean accesibles desde el plugin, para diseñar y escribir las interfaces de manera apropiada.

APLICANDO TESTING UNITARIO CON VISUAL STUDIO

Desarrollo guiado por pruebas

Esta técnica propone que los programadores escriban sus propias pruebas, las implementen antes de escribir cada clase y, luego, las programen con el fin de pasarlas.

El desarrollo guiado por pruebas (TDD, por las siglas de *Test Driven Development*) es una de las prácticas recomendadas por la más famosa de las metodologías ágiles: *Extreme Programming*. Si bien el empleo conjunto y simultáneo de todas las técnicas que la componen es uno de los requisitos indispensables en la aplicación de esta metodología, en este artículo vamos a concentrarnos, solamente, en TDD, ya que es posible utilizarla en muchos contextos, más allá de las características del proceso que se haya decidido adoptar.

Uno de los propósitos de TDD es agilizar el ciclo de escritura de código y realización de pruebas de unidad. Recordemos que las pruebas de unidad son las que se efectúan sobre una parte de un programa para comprobar que éste cumple su función específica. En los métodos tradicionales de desarrollo de software, estas pruebas son realizadas por personas especializadas, cuya única tarea es asegurar la calidad del producto final.

El hecho de que quienes estén a cargo de escribir el código y quienes deban probarlo sean grupos distintos suele originar cierto nivel de competencia: los programadores se esfuerzan por escribir código correcto, que luego deberán entregar a sus "oponentes" para que éstos traten de demostrar lo contrario, encontrando los casos en los que el programa falla. Si bien la competencia resulta provechosa en ciertas circunstancias, un equipo de desarrollo debería de estar conformado por individuos que realizaran un esfuerzo conjunto en pos de un objetivo común, sin enfrentarse entre ellos.

Además, el trámite que supone enviar el código para que sea probado demora el avance del proyecto. Antes de entregar su código, el programador espera a estar seguro de haberlo revisado lo suficiente. Y es recién en ese momento cuando se inician las pruebas y se documentan los errores. Para subsanarlos, el programador original, que ha estado esperando o ha comenzado a trabajar en otra parte del proyecto, tiene que volver a sumergirse en el código que había dejado de lado.

Poniendo el carro delante del caballo

La propuesta del desarrollo guiado por pruebas es radicalmente distinta de la explicada en la sección anterior. Según esta práctica, es el programador quien realiza las pruebas de unidad de su propio código, y las implementa **antes** de escribir el código que debe ser probado.

Cuando recibe el requerimiento de implementar una parte del sistema, y una vez que comprendió cuál es la funcionalidad pretendida, debe empezar por pensar qué pruebas tendrá que pasar ese fragmento de programa (o unidad) para que se lo considere correcto. Luego, procede a programar; pero no el código de la unidad que le tocó, sino el que se va a encargar de llevar a cabo las pruebas. Cuando está satisfecho de haber escrito todas las pruebas necesarias (y no antes), comienza a programar la unidad, **con el objetivo de pasar las pruebas** que programó.

La forma de trabajo del programador también cambia mucho durante la escritura del código funcional propiamente dicho. En vez de trabajar durante horas o días hasta tener la primera versión en condiciones de ser probada, va creando pequeñas versiones que puedan ser compiladas y pasen, por lo menos, algunas de las pruebas. Cada vez que hace un cambio y vuelve a compilar, también repite la ejecución de las pruebas de unidad. Y trata de que su programa vaya pasando más y más pruebas hasta que no falle en ninguna, que es cuando se considera que está listo para ser integrado con el resto del sistema.

Una de las grandes diferencias con el estilo de trabajo tradicional es que el programador ya no compite con otro u otros, sino que lo hace consigo mismo. Se enfrenta con el desafío de escribir código que pase las pruebas que él mismo ha programado. Y, siguiendo las premisas de muchos de los métodos ágiles, sólo programará lo que sea estrictamente necesario para ese fin.

Pasemos a un ejemplo muy simple, para ilustrar cómo funciona este proceso. Supongamos que nos toca desarrollar un componente .NET que se encargue de ordenar un arreglo de valores enteros y que, como parte del diseño, se ha decidido hacerlo mediante el algoritmo de **bubble sort** u ordenamiento por burbujeo. Para los que no recuerden o no conozcan este algoritmo, consiste en recorrer el arreglo intercambiando entre sí los elementos contiguos que estén en el orden inverso al buscado, y recomenzar ese procedimiento tantas veces como sean necesarias, hasta que no queden pares por intercambiar.

NUnit, otra alternativa

Una alternativa más económica a usar Visual Studio Team System 2005 para TDD consiste en comprar la edición Standard o Professional y complementarla con un framework de tests de unidad de uso libre, como NUnit (www.nunit.org). Estos frameworks pueden interactuar, incluso, con las ediciones Express de Visual Studio, que también son gratuitas.

Como estamos empleando TDD, tenemos que empezar por decidir qué pruebas vamos a hacer. Elegimos probar con varios arreglos de cinco elementos que siempre serán los números del 1 al 5. Y se nos ocurren las siguientes pruebas:

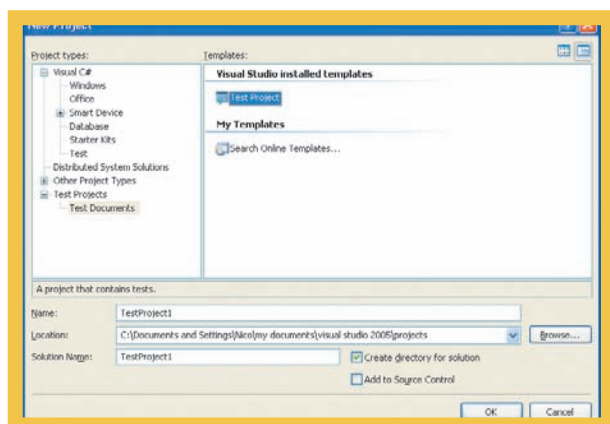
- Un arreglo ya ordenado.
- Uno que tenga solamente un par de elementos contiguos fuera de orden.
- Uno que tenga el último elemento al principio (y el resto en orden).
- Uno que tenga el primer elemento al final.
- Uno con el orden de sus elementos completamente invertido.

Tal vez no sea suficiente, pero yo me siento conforme. Pienso que si logro escribir un programa que pase todas estas pruebas, habré cumplido con lo pedido.

El paso siguiente es implementar las pruebas en forma de un programa que examine nuestro componente (que todavía no existe). Dado que todo desarrollador guiado por pruebas (y quien quiera hacer pruebas de unidad) tiene que crear frecuentemente programas de este tipo, los entornos modernos de desarrollo ofrecen herramientas integradas que facilitan el trabajo. Veamos cómo hacerlo en Microsoft Visual Studio Team System 2005, que es el entorno que yo uso para trabajar en .NET. Pero los pasos por seguir no difieren mucho si se usa otro ambiente de última generación.

Entre las extensiones que brinda Team System a la instalación básica de Visual Studio 2005, se agrega un tipo de proyecto entre los que se sugieren en el momento de crear uno nuevo: Test Project, como se muestra en la [Figura 1].

Para crear casos de prueba de unidad, Visual Studio se basa fuertemente en los atributos que .NET permite asociar como metadatos a distintos elementos de un programa. Las aplicaciones disponibles para otras plataformas de desarrollo suelen imponer convenciones para denominar las clases y los métodos que realizarán las pruebas; por ejemplo, comenzando su nombre con la cadena **test**. Pero la herramienta que estamos usando evita esta rigidez, ya que podemos usar los nombres que prefiramos, siempre y cuando agreguemos los atributos que marcan esos elementos del programa como responsables para probar. Los atributos que vamos a usar en este ejemplo son dos: [TestClass] y [TestMethod]. El primero designa una clase como contenedora de un grupo o batería de pruebas, y el segundo marca cada método a cargo de realizar una prueba. Al crear un nuevo proyecto de tipo Test Project, el ambiente le agrega un archivo que contiene la plantilla de una clase de pruebas. Aunque no es necesario hacerlo así, antes de empezar a programar las pruebas, vamos a crear el esqueleto de la clase que se va a encargar de hacer el ordenamiento, para tenerlo disponible con IntelliSense. Es una clase muy simple, con un nombre asignado por defecto por Visual Studio y un método que será el que efectúe el ordenamiento:



[Figura 1] Creación del nuevo proyecto para test.

EL PROGRAMADOR IMPLEMENTA LAS PRUEBAS DE SU PROPIO CÓDIGO, ANTES DE ESCRIBIRLO.

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

namespace Bubble1
{
  class Class1
  {
     public void Sort(int[] arr)
     {
     }
  }
}
```

Y ya estamos listos para escribir nuestra batería de pruebas, con todas las que enumeramos anteriormente:

```
using System;
using System.Text;
using Microsoft.VisualStudio.QualityTools.
  UnitTesting.Framework;

namespace TestProject1
{
  /// <summary>
  /// Pruebas para el bubble sort
  /// </summary>
  [TestClass]
  public class Batería1
  {
     private int[] resp = new int[]{ 1, 2, 3, 4, 5 };

     private Bubble1.Class1 sorter =
        new Bubble1.Class1();

     /// <summary>
     /// Prueba con un arreglo ordenado.
     /// </summary>
     [TestMethod]
     public void PruebaOrdenados()
     {
        int[] arr = new int[] { 1, 2, 3, 4, 5 };
        sorter.Sort(arr);
        CollectionAssert.AreEqual(resp, arr);
     }

     /// <summary>
```

```
        /// Prueba con un solo swap.
        /// </summary>
        [TestMethod]
        public void PruebaUnSwap()
        {
            int[] arr = new int[] { 1, 3, 2, 4, 5 };
            sorter.Sort(arr);
            CollectionAssert.AreEqual(resp, arr);
        }

        /// <summary>
        /// Prueba con el último al principio.
        /// </summary>
        [TestMethod]
        public void PruebaUlt()
        {
            int[] arr = new int[] { 5, 1, 2, 3, 4 };
            sorter.Sort(arr);
            CollectionAssert.AreEqual(resp, arr);
        }

        /// <summary>
        /// Prueba con el primero al final.
        /// </summary>
        [TestMethod]
        public void PruebaPrim()
        {
            int[] arr = new int[] { 5, 2, 3, 4, 1 };
            sorter.Sort(arr);
            CollectionAssert.AreEqual(resp, arr);
        }

        /// <summary>
        /// Prueba con el arreglo invertido.
        /// </summary>
        [TestMethod]
        public void PruebaInv()
        {
            int[] arr = new int[] { 5, 4, 3, 2, 1 };
            sorter.Sort(arr);
            CollectionAssert.AreEqual(resp, arr);
        }

    }
}
```

Además de los métodos de prueba (marcados con el atributo [TestMethod]), nuestra clase tiene los siguientes miembros:

- Un ejemplo del arreglo ordenado, para compararlo con el resultado de cada prueba:

```
private int[] resp=new int[]{ 1,2,3,4,5 };
```

- Un campo que representa a la clase que vamos a probar:

```
private Bubble1.Class1 sorter = new Bubble1.Class1();
```

Nuestros métodos de prueba siempre empiezan creando el arreglo que van a pasar como argumento. Ésta es la única instrucción que varía entre uno y otro de estos métodos:

```
int[] arr = new int[]{ 5,4,3,2,1 };
```

Después invocan al método de ordenamiento:

```
sorter.Sort(arr);
```

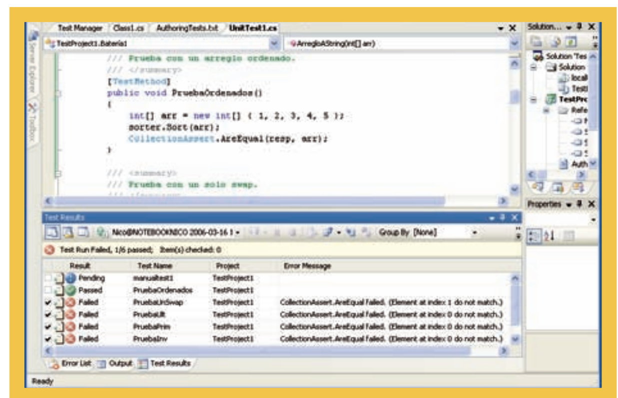
Y terminan comparando el resultado con el arreglo de ejemplo:

```
CollectionAssert.AreEqual(resp, arr);
```

La clase **CollectionAssert** es una de las que provee Visual Studio, dentro del namespace **Microsoft.VisualStudio.QualityTools.UnitTesting.Framework**, para evaluar los resultados de las pruebas. Se usa para comparar dos colecciones (en este caso, arreglos). La primera es el resultado esperado y la segunda, el obtenido. Si quisiéramos comparar objetos que no constituyeran colecciones, usaríamos, en cambio, la clase **Assert**.

Y como estamos usando TDD, no debemos esperar a tener todo nuestro código escrito para que sea probado. Ya mismo compilamos todo el proyecto y corremos por primera vez la batería de pruebas. Como el programa todavía no hace nada, lo más probable es que no pase ninguna prueba, pero de todas maneras seguimos el método a ultranza y nos fijamos cuáles pruebas son las que fallan, para tratar de solucionar el problema (por ahora, el inconveniente es que no hay ningún código de ordenamiento).

Aquí es donde podemos apreciar el poder de contar con un entorno que soporta pruebas de unidad. El **assembly** que creamos –y que contiene tanto la clase Bubble1.Class1 (el componente que estoy desarrollando), como la clase TestProject1.Batería1 (nuestra batería de pruebas)– no es ejecutable, es una biblioteca de vinculación dinámica (DLL). Para hacer las pruebas, normalmente tendríamos que crear un programa ejecutable adicional. Pero no hace fal-



[Figura 2] Resultado del test que se muestra luego de compilar el proyecto.

ta, porque Visual Studio se encarga de mostrarnos los resultados. Cuando compilamos la solución y la ejecutamos, aparecen en una ventana parecida a la que se usa para mostrar los errores de compilación, como se ve en la **[Figura 2]**.

¡El resultado es mejor de lo esperado! Aunque no escribimos nada de código para hacer el ordenamiento, ya conseguimos pasar la primera prueba (aparece con una marca verde). Por supuesto, no era una tarea muy difícil, porque el argumento ya es un arreglo ordenado. Pero eso no significa que la prueba esté de más: si nos equivocamos al escribir el algoritmo, tal vez nuestro supuesto método para ordenar, en realidad, desordene una secuencia previamente ordenada. De hecho, debemos tener cuidado para que eso no suceda al ir avanzando. Ésta es una constante en TDD: debemos reparar el código para que pase las pruebas que fallaron, pero sin que

la nueva versión deje de superar las que ya habían sido exitosas.

Previsiblemente, el método vacío no pasa el resto de las pruebas (marcadas en rojo, con una cruz). Si leemos la columna "Error Message" para la segunda de nuestras pruebas, veremos que los dos arreglos (el esperado y el obtenido) difieren en su segundo elemento (recordemos que los arreglos empiezan en la posición 0). Esta descripción puede no ser lo suficientemente ilustrativa para descubrir qué fue lo que no funcionó. Para remediarlo, vamos a agregar a la clase **Batería1** un método que construya una cadena de caracteres que muestre el contenido de un arreglo:

```
/// <summary>
/// Presentar un arreglo de enteros,
     con sus elementos pegados.
/// </summary>
public string ArregloAString(int[] arr)
{
  StringBuilder sb = new StringBuilder("[");
  foreach (int i in arr)
     sb.Append(i.ToString());
  return sb.Append("]").ToString();
}
```

Ahora podemos pasarle un tercer argumento al método **Collection Assert.AreEqual**, consistente en una cadena de caracteres:

```
CollectionAssert.AreEqual(resp, arr, "Obtenido: " +
    ArregloAString(arr));
```

Si volvemos a ejecutar las pruebas, el mensaje de error mostrado para la segunda es más ilustrativo: "CollectionAssert.AreEqual failed. Obtenido: [13245]". Y así sucesivamente para el resto de las pruebas. Entonces, nuestro objetivo ahora es superar las cuatro pruebas restantes.

Superando pruebas

Para avanzar, escribimos la primera versión operativa del método **Bubble1.Class1.Sort(int[] arr)**. Se trata de un ciclo que recorre el arreglo invirtiendo los pares contiguos desordenados:

```
public void Sort(int[] arr)
{
  for (int i = 0; i < arr.Length - 1; i++)
     if (arr[i] > arr[i + 1])
     {
        int aux = arr[i];
        arr[i] = arr[i + 1];
        arr[i + 1] = aux;
     }
}
```

El ciclo recorre todos los elementos del arreglo (excepto el último) y compara cada uno con el siguiente. En caso de que no estén en orden, usa una variable auxiliar para intercambiar sus valores.

Result	Test Name	Project	Error Message
Pending	manualtest1	TestProject1	
Passed	PruebaOrdenados	TestProject1	
Passed	PruebaUnSwap	TestProject1	
Passed	PruebaUlt	TestProject1	
Failed	PruebaPrim	TestProject1	CollectionAssert.AreEqual failed. Obtenido: [23415](Element at index 0 do not match.)
Failed	PruebaInv	TestProject1	CollectionAssert.AreEqual failed. Obtenido: [43215](Element at index 0 do not match.)

[Figura 3] Resultado luego de realizar una recopilación del proyecto.

Como indica TDD, no bien terminamos de hacer la modificación, recompilamos y realizamos las pruebas. El resultado es el que se muestra en la [Figura 3].

¿Qué ocurrió? Nuestro nuevo método de ordenamiento es mejor que el anterior, al menos si lo medimos en la cantidad de pruebas que supera.

El código no implementa el algoritmo de **bubble sort** completo; sólo hace una pasada por el arreglo. El algoritmo original pide que se vuelva a comenzar, siempre y cuando haya habido un cambio, cosa que no estamos haciendo. Se acomodó, entonces, el arreglo que tenía una única inversión; y también, el que tenía el último elemento al principio, porque la pasada que estamos haciendo se encarga de ir invirtiendo su posición con la de cada uno de los otros elementos, hasta dejarlo en su lugar. Pero en el arreglo que tiene el primer elemento al final, sólo se acercó este valor un paso a su posición correcta; y en el arreglo invertido no se consiguió más que llevar el 5 a su sitio. Lo que nos faltó hacer fue encerrar el ciclo que hace una pasada en otro, que lo repita mientras haya algún cambio. Ésta es la nueva apariencia del método de ordenamiento:

```
public void Sort(int[] arr)
{
  bool cambió = true;

  while (cambió)
  {
     cambió = false;
     for (int i = 0; i < arr.Length - 1; i++)
        if (arr[i] > arr[i + 1])
        {
           int aux = arr[i];
           arr[i] = arr[i + 1];
           arr[i + 1] = aux;
           cambió = true;
        }
  }
}
```

Agregamos una variable local (cambió), a la que se le da el valor **false** antes de cada pasada, y sólo valor true si se hace algún cambio. Como punto de interés adicional, vale la pena notar que, como lo habíamos hecho con el nombre de la clase de prueba (Batería1), aprovechamos el hecho de que .NET permite usar todos los caracteres Unicode en los identificadores y no solamente los del código ASCII básico. Por haber programado en otros lenguajes, muchos estamos acostumbrados a omitir caracteres en nuestros identificadores, como vocales acentuadas o eñes, pero es aconsejable no hacerlo, en aras de la claridad (por ejemplo, no es lo mismo "cambio" que "cambió").

Result	Test Name	Project	Error Message
Pending	manualtest1	TestProject1	
Passed	PruebaOrdenados	TestProject1	
Passed	PruebaUnSwap	TestProject1	
Passed	PruebaUlt	TestProject1	
Passed	PruebaPrim	TestProject1	
Passed	PruebaInv	TestProject1	

[Figura 4] Resultado final de nuestro proyecto con todos los tests aprobados.

Si compilamos y probamos el nuevo código, obtenemos el resultado de la [Figura 4].

Conclusión

¡Ya pasamos todas las pruebas! Si nos regimos por el criterio de la técnica de desarrollo guiado por pruebas, ésta es justamente la definición de que nuestro código es válido: según TDD, una unidad es correcta cuando supera todas sus pruebas.

USANDO PUNTEROS A FUNCIONES

Delegados en C#

Para asegurar el uso de punteros a funciones, C# 2.0 implementa los delegados, que le agregan la característica de orientación a objetos. En este artículo, analizamos su funcionalidad y su uso.

Cuando queremos enterarnos de los cambios en un sistema que estamos observando, solemos verificar su estado y, tal vez, guardar el valor de la variable observada. Con cierta frecuencia, repetimos esta acción, verificando una y otra vez el valor obtenido y comparándolo con los anteriores. De la comparación entre dos valores distintos, podemos inferir cambios en el estado del sistema observado.

Esta forma de registrar el comportamiento de un sistema no es la mejor en materia de recursos. Es evidente que, cuando aumenta la frecuencia de observación (medida como la cantidad de veces que obtenemos el valor de una variable del sistema en una unidad de tiempo), crece la sensibilidad a los cambios. Esto quiere decir que observaremos más cambios (en realidad, estaremos observando la misma cantidad, pero con una resolución mucho mayor).

Obviamente, esta situación plantea una desventaja muy importante, que, en su forma más extrema, nos lleva a utilizar más recursos de lo debido, con lo cual se provocan cambios en el comportamiento del sistema observado como consecuencia de observarlo.

Otra forma de observar cambios en un sistema consiste en usar un mecanismo de observador / observado (implementado en el observer pattern, por ejemplo). Este tipo de mecanismo permite la suscripción a eventos generados dentro de un sistema. De este modo, si se producen los eventos que se están observando, el **mismo** sistema nos avisará sobre los cambios, momento en el cual medimos la variable de interés para determinar el nuevo estado.

Por lo general, la segunda forma es mejor en cuanto al uso de recursos. Así, un mecanismo de suscripción suele resultar mucho mejor para la observación de cambios de estado de un sistema. Muchos lenguajes soportan esta funcionalidad a través de la implementación de métodos callback (*callback methods*), que reciben direcciones a otros métodos como parámetros.

El uso de delegados promueve la separación entre la funcionalidad implementada y el cliente de esa funcionalidad, lo cual mejora el encapsulamiento y disminuye el acoplamiento entre ellos.

¿Qué es un delegado?

Un delegado es un tipo del CLR que deriva de System.Delegate y que se utiliza para manejar punteros a funciones.

Características de los delegados

Un delegado puede encapsular métodos de instancia o métodos estáticos. Estos métodos son llamados entidades "invocables" (*callable entity*). Si la entidad almacena un método de instancia, se dice que contiene el método y la instancia donde éste se encuentra. En cambio, si almacena un método estático, entonces sólo contiene el método.

Método de instancia → Delegado (método + instancia)
Método estático → Delegado (método)

Un delegado está implementado en la clase System.Delegate, marcado como **sealed**, por lo que no se permite derivar desde el tipo System.Delegate.

El conjunto de métodos encapsulados por un delegado se conoce como lista de invocación (*invocation list*). Cuando el delegado encapsula sólo un método, la lista de invocación contiene una única entrada.

El poder de la ignorancia

Una ventaja de los delegados es que no conocen la clase de métodos que encapsulan. Sólo se requiere que sean **consistentes** con el tipo del delegado, lo cual los convierte en una excelente opción para la **invocación anónima**. Veremos este tema más adelante.

De esta manera, el conocimiento que tiene el cliente acerca de la implementación es nulo; simplemente, consume la funcionalidad implementada.

¿Cómo definir un delegado?

Hay tres pasos que permiten la definición de un delegado:

1. Declarar
2. Instanciar
3. Invocar

1. Declarar

La forma más sencilla de declarar un delegado es la siguiente:

```
delegate void Delegado1();
```

Por supuesto que los delegados pueden tener tipo de retorno y/o parámetros de función. Veremos cómo se utilizan más adelante.

2. Instanciar

```
Delegate1 d = new Delegate1(Funcion1);
```

En este caso, **Funcion1** es una función que no devuelve ningún valor. Deben coincidir los tipos devueltos de la función y el delegado (a esto nos referimos

antes cuando hablamos de consistencia). Funcion1 está definida, previamente, como:

```
static void Funcion1(){
  // codigo
}
```

3. Invocar

Sólo se necesita llamar al tipo definido:

```
d();
```

Todo junto

El siguiente ejemplo implementa los tres pasos en una aplicación de consola.

```
delegate void Delegate1(); // 1-Declarar
class Program{
  static void Funcion1(){
      Console.WriteLine("Funcion1 acaba de ser
         invocada");
  }
  static void Main(string[] args) {
      Delegate1 d = new Delegate1(Funcion1);
      // 2-Instanciar
      d(); // 3-Invocar
  }
}
```

Nótese cómo el método **Funcion1()** es invocado a través del delegado, y no, directamente. Si bien el ejemplo es muy sencillo, no tiene mucho sentido, ya que la misma funcionalidad podría lograrse llamando a **Funcion1()**. Por lo tanto, veamos otro caso.

Otro ejemplo

Este ejemplo implementa una clase **Contador**, que contiene dos propiedades: **Desde** y **Hasta**, que definen un rango de acción para el contador. Existe, además, un método **Comenzar()**, que "cuenta" empezando en **Desde** hasta llegar a **Hasta** e informa a los observadores (clientes) sobre el nuevo valor. Con cada iteración, el estado interno del Contador cambia. Como dijimos antes, la clase informa del cambio pasando el valor. Más adelante veremos una modificación a este código.

Volviendo al ejemplo, cada vez que se produce la iteración, avisamos a todos los consumidores de la clase (clientes) que están observando los cambios de estado, que hubo una modificación en el sistema.

Archivo: Contador.cs

```
delegate void Paso(int x);
class Contador {
  private int _Desde, _Hasta;
  public int Desde
  {
      get { return _Desde;}
      set { _Desde = value; }
  }
  public int Hasta
{
      get { return _Hasta; }
      set { _Hasta = value; }
  }
  public void Comenzar(Paso p)
  {
     for (int i = this.Desde; i < this.Hasta; i++){
        p(i);
  }
}
```

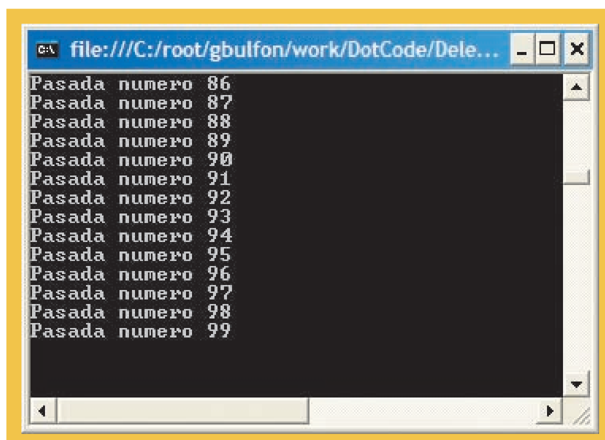
Para utilizarla:

```
class Test
{
  public static void Aviso(int x) {
     Console.WriteLine("Pasada numero {0}",
        x.ToString());
  }
}
class Program
{
  static void Main(string[] args) {
     // preparar
     Contador c = new Contador();
     c.Desde = 10; c.Hasta = 100;
     // delegar
     Paso pTemp = new Paso(Test.Aviso);
     // Paso era el delegado
     c.Comenzar(pTemp); // con cada pasada,
        la funcion Aviso es invocada
  }
}
```

De este modo, podemos proveer la funcionalidad deseada a través de una clase o método (en nuestro caso, la clase **Test**) para los eventos generados en otra clase (clase **Contador**).

Veamos ahora una modificación del ejemplo anterior, que agrega una propiedad a Contador, llamada **Value**. Ésta irá cambiando con cada pasada del método **Comenzar()** y representará los distintos estados de **Contador**. Así, en vez de pasar el valor como parámetro, se cambiará el valor de la propiedad **Value** y se informará a los observadores que hubo un cambio en el objeto.

Los observadores de este ejemplo tienen "cierto" conocimiento sobre Contador, por lo que averiguan su estado consultando la propiedad **Value**.



[Figura 1] Salida obtenida por el programa.

Contador.cs

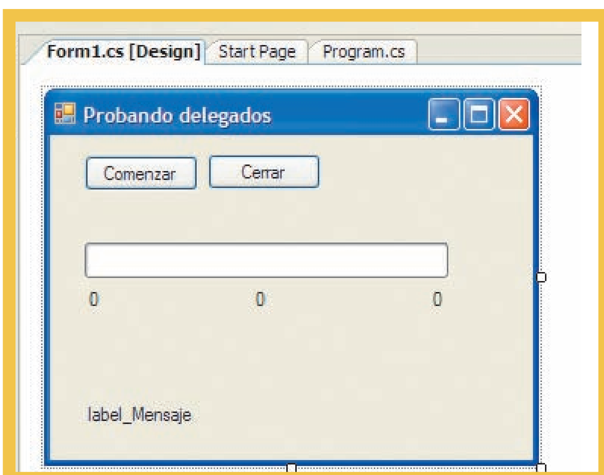
```
delegate void Paso(Contador contador);
class Contador
{
  private int _Desde, _Hasta;
  private int _Value;
  public int Value
  {
     get { return _Value; }
     set { _Value = value; }
  }
  public int Desde { get { return _Desde; }
     set { _Desde = value;}}
  public int Hasta { get { return _Hasta; }
     set {_Hasta = value;}}
  public void Comenzar(Paso p){
     for (int i = this.Desde; i < this.Hasta; i++){
        this.Value = i;
        p(this); // delegar
     }
  }
}
```

Test.cs

```
class Test
{
  public static void Aviso(Contador c)
  {
     Console.WriteLine("Pasada numero {0}", c.Value);
  }
}
```

Program.cs

```
class Program
{
  static void Main(string[] args)
  {
     // preparar
     Contador c = new Contador(); // instanciar
     c.Desde = 10; c.Hasta = 100;
     // delegar
     Paso pTemp = new Paso(Test.Aviso);
```



[Figura 2] Así se ve representado el formulario.

```
     c.Comenzar(pTemp);
     // para detener
     Console.Read();
  }
}
```

Más ejemplos

Un código como el siguiente utiliza delegados (delegate o System.Delegate) para mantener informado al usuario sobre el avance de una tarea que ocurre dentro de una clase, por medio del uso de una barra de progreso. Así, **Aviso()** podría actualizar controles de interfaz, como en el ejemplo siguiente:

```
class Proceso {
 public delegate void Paso(int x);
 private int _Desde, _Hasta;
 public int Desde
 {
     get { return _Desde; }
     set { _Desde = value; }
 }
 public int Hasta
 {
     get { return _Hasta; }
     set { _Hasta = value; }
 }
 public void Comenzar(Paso p)
 {
     for (int i = this.Desde; i < this.Hasta; i++)
     {
 p(i);
        for (int j = 0; j < 100000; j++) ; // delay
     }
  }
}
```

El parámetro del método Comenzar es un delegado que se utiliza de la siguiente manera:

```
private void ActualizarProgressBar(int x){
 if (x > progressBar1.Minimum && x <
     progressBar1.Maximum)
  progressBar1.Value = x;
}
private void button1_Click(object sender, EventArgs e)
{
  Proceso proceso = new Proceso();
  proceso.Desde = 0;proceso.Hasta = 1000;
  this.progressBar1.Minimum = proceso.Desde;
  this.progressBar1.Maximum = proceso.Hasta;
  proceso.Comenzar
     (new Proceso.Paso(ActualizarProgressBar));
  this.button_Cerrar.Enabled = true;
}
private void button_Cerrar_Click
  (object sender, EventArgs e)
{
  Close();
}
```

El método **ActualizarProgressBar()** actualiza el valor de la barra para informar acerca del estado del proceso (interno a la clase Pro-

ceso) que se está ejecutando. Quienes quieran practicar un poco más pueden intentar agregar delegados para informar a las clases clientes la terminación del proceso.

Sobre la lista de invocación

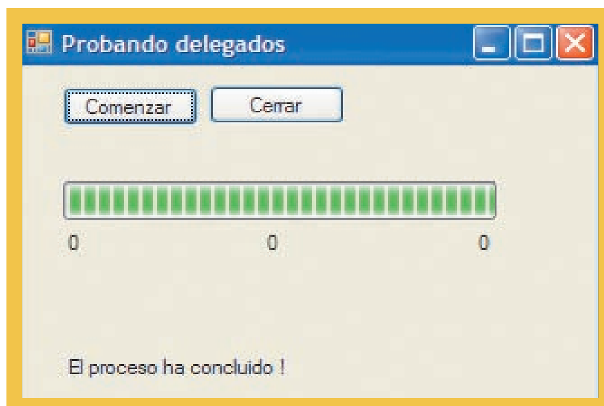
En la sección sobre caracteristicas de los delegados nombramos la lista de invocación (*invocation list*).

Veamos un ejemplo que muestra esta lista de un delegado para los métodos estáticos y de instancia.

El ejemplo utiliza las propiedades **Method** y **Target** definidas en la clase **System.Delegate** para mostrar, en una aplicación de consola, que un delegado almacena sólo el método cuando éste es estático, y el método más la instancia cuando éste pertenece a una de ellas:

Program.cs

```
public delegate void D(); // Declarar
class ContadorSimple
{
  public void Comenzar(D d)
  {
     for (int i = 0; i < 10; i++)
  {
        d();
     }
  }
}
class Test
{
  public void AvisoDinamico()
  {
     Console.WriteLine("Metodo invocado");
  }
}
class Program
{
  static void Aviso()
 {
     Console.WriteLine("Metodo Aviso() invocado");
 }
  static void Main(string[] args)
  {
     // instanciar
     D d1 = new D(Aviso);
     D d2 = new D(Aviso);
     Test t = new Test();
     D d3 = new D(t.AvisoDinamico);
     d1 += d2;
     d1 += d3;
     ContadorSimple cs = new ContadorSimple();
     cs.Comenzar(d1); // invocar
     #region Invocation List
     System.Delegate[] il = d1.GetInvocationList();
     Console.WriteLine("\nLista de invocacion
        del delegado d1");
     Console.WriteLine("-------------------------");
     foreach(System.Delegate d in il){
        if(d.Target == null)
           Console.WriteLine("\tMetodo estatico :{0}
              ",d.Method.Name);
        else
           Console.WriteLine("\tMetodo de la instancia
              {0} :{1} ", d.Target,d.Method.Name);
     }
     #endregion
     Console.ReadLine();
   }
 }
```



[Figura 3] Así se verá cuando el proceso finalice.

La clase **ContadorSimple** contiene el método **Comenzar**, cuya firma incluye un delegado como parámetro. La clase **Test** se utiliza para crear un delegado con un método de instancia, mientras que el método **Aviso** se emplea para crear un delegado con un método estático.

Tanto **d1** como **d2** contienen referencias a métodos estáticos. Se concatenan en **d1** para mostrar la lista de invocación de **d1** (de lo contrario, aparecería sólo un método).

El delegado **d3** se usa con una referencia a un método de instancia. Es concatenado a **d1** por la misma razón que lo hicimos con **d2.**

El array **il** contiene la lista de invocación de d1 obtenida utilizando el método **GetInvocationList()**, perteneciente a **System.Delegate**, y que devuelve un array de objetos del tipo **System.Delegate**.

```
System.Delegate[] il = d1.GetInvocationList();
```

Si **Target** es **null**, el delegado contiene sólo referencias a métodos estáticos, y es utilizado en la sentencia **foreach** para identificar esa condición:

```
if(d.Target == null)
```

Resumen

Hemos visto que un delegado es un mecanismo para implementar llamadas a funciones. Por supuesto, su uso puede ampliarse a cualquier observador. También aprendimos cómo tratar los delegados y vimos que un delegado contiene una copia de la instancia, si el método asociado es de instancia, o sólo el método, si éste es estático.

Delegados en Visual Basic

En Visual Basic.NET también es posible crear y utilizar delegados. Para definir un delegado, se utiliza la sintaxis **Delegate Sub Paso(ByVal x As Integer)**. Luego, es necesario crear una instancia de dicho delegado y, por útimo, cuando queremos invocarlo, utilizar **objeto_delegado.Invoke(parametros)**.

DOMINANDO LOS ARCHIVOS PLANOS POR MEDIO DE METADATA

Procesamiento automático de archivos de texto

Pese a que XML sigue creciendo, por lo general usamos archivos de texto con registros de ancho fijo o delimitados. Podemos recurrir a la librería FileHelpers para procesar estos archivos sin mucho código.

Todos los que hemos tenido que importar o exportar datos con archivos de texto plano sabemos lo tediosa y rutinaria que resulta esta tarea (sobre todo, cuando, luego de un tiempo, algún jefe pide cambiarlo). En este artículo veremos cómo olvidarnos para siempre de los problemas de manipulación de strings y archivos; y cómo, con un par de líneas de código, podemos resolver todos los problemas de parsing, conversión, encodings, valores nulos, manejo de errores y mucho más.

Lo que nos permite dejar atrás todos estos problemas son los FileHelpers, una librería Open Source que desarrollé para la manipulación de archivos de texto con formato, y que usan el potencial de .NET para obtener información del código en tiempo de ejecución (metadata).

Comencemos a trabajar

Empecemos con un ejemplo sencillo para clarificar las cosas. Supongamos que tenemos que leer y escribir archivos con el siguiente formato:

```
2732,Juan Perez,435.00,11052002
554,Pedro Gomez,12342.30,06022004
112,Ramiro Politti,0.00,01022000
924,Pablo Ramirez,3321.30,24112002
..
```

La solución que a la mayoría se nos ocurre es:

- Abrir el archivo.
- Leer línea por línea.
- Hacer un Split para obtener cada campo.
- Convertir cada campo al formato correcto.
- Asignarlos a los campos de una clase.
- Agregar este último objeto al arreglo y seguir.
- Para escribir el archivo, se hace el mismo proceso, pero a la inversa.

Imaginemos la cantidad de código necesario para hacer esto... Ahora veamos cómo hacerlo por medio de FileHelpers.

Primero, definimos la clase que representa cada registro, así:

```
using FileHelpers;

[DelimitedRecord(",")]
public class Cliente
{
        public int Codigo;
        public string Nombre;
        public decimal Saldo;
        [FieldConverter(ConverterKind.Date, "ddMMyyyy")]
        public DateTime FechaAlta;
}
```

Para los más desorientados, esas etiquetas encerradas entre [ ] son los atributos en C#, que nos permiten agregar esa metadata que los FileHelpers necesitan para identificar el formato de cada registro. Al mirar el ejemplo, queda claro que cada cliente es un registro delimitado por "," y que el último campo es un campo fecha que tiene el formato ddMMyyyy. He aquí otra ventaja de utilizar la librería: el código queda autodocumentado; si lo hubiésemos hecho a la antigua, el formato del archivo no estaría explicito en ningún lado y habría que deducirlo del código. Ahora sigamos con el código necesario para leer y escribir estos archivos con los FileHelpers... y abracadabra:

```
// Le decimos qué clase queremos manejar

FileHelperEngine engine = new FileHelperEngine(typeof(Cliente));
// Leemos los datos del archivo

Cliente[] clientes = (Cliente[]) engine.ReadFile
("clientes.txt");

// Si queremos escribir un archivo
engine.WriteFile("salida.txt", clientes);
```

Como vemos, es **una línea de código para cada operación**. Por debajo la librería se encarga de todo lo que antes describimos, pero de manera totalmente transparente, lo cual nos deja más tiempo para desarrollar el resto del programa. En el CD que acompaña a la revista o a través de Internet pueden encontrar el código completo de estos ejemplos.

Cómo usarlo desde VB.NET

Los amigos de VB.NET no tienen que ponerse mal, porque la librería puede ser usada desde este lenguaje, mediante la siguiente sintaxis:

```
Imports FileHelpers
<DelimitedRecord(",")> _
Public Class Cliente

  Public Codigo As Integer
  Public Nombre As String
  Public Saldo As Decimal
  <FieldConverter(ConverterKind.Date, "ddMMyyyy")> _
  Public FechaAlta As DateTime
End Class
```

Y para usar la librería:

```
'-> Estas dos lineas son el uso de la librería
Dim engine As New FileHelperEngine(GetType(Cliente))
Dim clientes As Cliente()
'-> Para Leer
clientes = DirectCast(engine.ReadFile("clientes.txt"),
Cliente())
'-> Para Escribir
engine.WriteFile("salida.txt", clientes)
```

Registros de ancho fijo

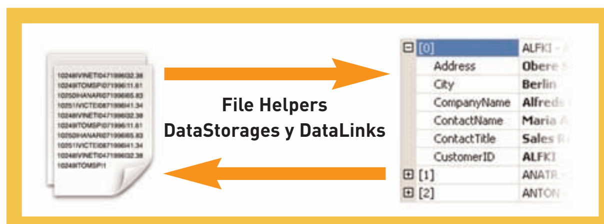
Ahora veamos cómo resuelve la librería los archivos con registros de ancho fijo. Supongamos que tenemos los mismos datos pero, ahora, con este formato:

```
01732Juan Perez           004350011052002
00554Pedro Gomez          123423006022004
00112Ramiro Politti       000000001022000
00924Pablo Ramirez        033213024112002
```

La gran diferencia entre los datos en este formato y el anterior es que, para el campo Saldo, no tenemos el punto decimal; está implícito que los seis primeros dígitos son la parte entera, y los dos últimos, la decimal. Ya veremos luego cómo hacer para resolver este pequeño problema.

Para manipular estos archivos, necesitamos, entonces, una clase parecida a ésta:

```
[FixedLengthRecord()]
public class Cliente2
```



[Figura 1] Ejemplo básico del uso de lectura y escritura de datos.

```
{
  [FieldFixedLength(5)]
  pulic int Codigo;
  [FieldFixedLength(20)]
  [FieldTrim(TrimMode.Right)]
  public string Nombre;
  [FieldFixedLength(8)]
  [FieldConverter(typeof(TwoDecimalConverter))]
  public decimal Saldo;
  [FieldFixedLength(8)]
  [FieldConverter(ConverterKind.Date, "ddMMyyyy")]
  public DateTime FechaAlta;
}
```

Como vemos, la definición está un poco más cargada, ya que necesitamos más metadata para poder describirla por completo.

Cada campo de una clase definida como **FixedLengthRecord** debe tener el atributo **FieldFixedLength** sobre cada uno de ellos. Como se ve en el ejemplo, es muy claro y simple modificar la cantidad de caracteres que abarca cada campo.

En la clase hay, además, un atributo **FieldTrim(**TrimMode.Right**)**, que provee la librería para eliminar los caracteres en blanco que queremos descartar durante la lectura.

Finalmente, lo más extraño puede que sea el atributo FieldConverter (**typeof** (TwoDecimalConverter)). Éste les dice a los FileHelpers que, para convertir un string al valor del campo o viceversa, use el conversor especificado y no el que tiene predefinido.

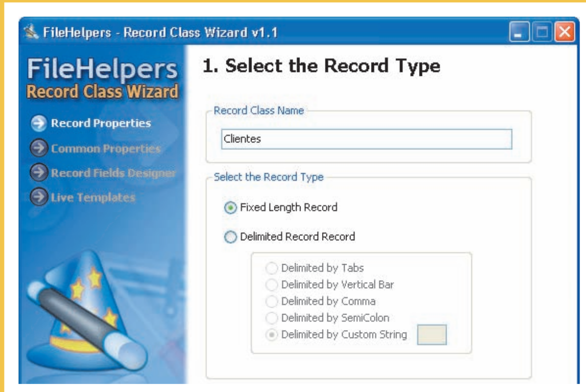
Todo conversor personalizado debe heredar de la clase ConverterBase y sobrescribir sus dos métodos –StringToField y FieldToString–; en este caso, la implementación del conversor sería:

Clases más importantes de la librería

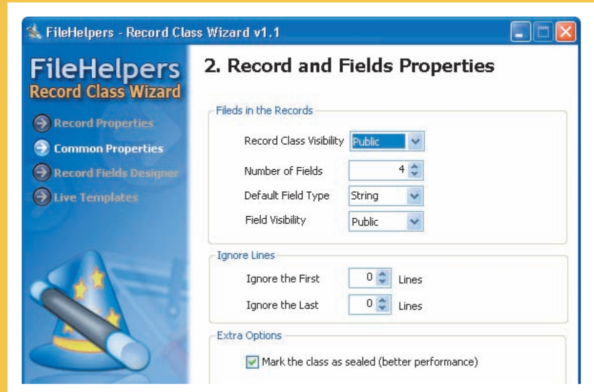
Clase	Descripción
FileHelperEngine	La clase básica de la librería para leer y escribir registros en una sola operación.
FileHelperEngine<T>	Igual que la anterior, pero con uso de Generics (.NET 2.0).
FileHelpersAsyncEngine	Permite el procesamiento registro por registro.
FileHelpersAsyncEngine<T>	Igual que la anterior, pero con uso de Generics (.NET 2.0).
MasterDetailEngine	Permite manipular archivos con dos tipos de registros que tienen una relación maestro detalle.
FileTransformationEngine	Permite convertir un archivo de un formato en uno nuevo, con un formato diferente.
CommonEngine	Una clase con métodos estáticos que simplifican el procesamiento, ya que no se necesita crear ningún objeto.
ExcelStorage, AccessStorage y SqlServerStorage	Permiten importar y exportar registros en esas fuentes de datos.

Creando las clases y el código con los FileHelpers Wizard

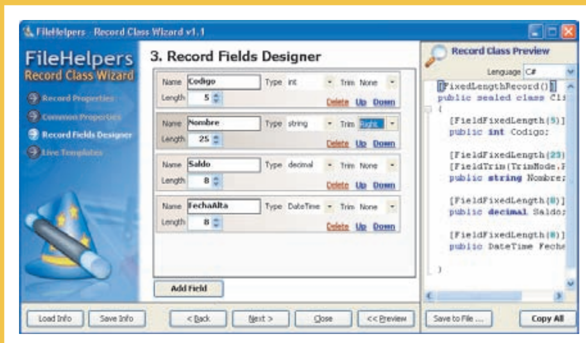
Guía paso a paso sobre la creación del código y las clases con FileHelpers.



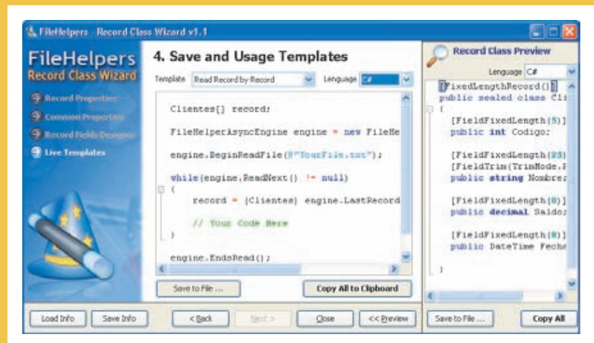
1 En la primera pantalla debemos especificar el nombre de la clase y el tipo de archivo por procesar.



2 Luego, la cantidad de campos, el tipo predefinido para cada uno y cuántas líneas hay antes o después de los registros.



3 Aquí definimos los nombres y tipos de cada campo, la longitud y su orden.
Contamos con una vista previa en tiempo real.



4 Finalmente, podemos utilizar las plantillas predefinidas con el fin de generar el código para leer o escribir archivos de varias formas y, sobre todo, en el lenguaje que necesitamos.

```
internal class TwoDecimalConverter: ConverterBase
{
  public override object StringToField(string from)
  {
     decimal res = Convert.ToDecimal(from);
     return res / 100;
  }
  public override string FieldToString(object from)
  {
     decimal d = (decimal) from;
     return Math.Round(d * 100).ToString();
  }
}
```

Bastante sencillo, sobre todo, porque a partir de ahora podemos reusar nuestro conversor tantas veces como queramos, sin cambiar una sola línea.

Manejo de errores

Hasta aquí, todo perfecto, pero todos sabemos que, al leer cada registro, pueden aparecer diferentes errores de formato. Para manejarlos, los FileHelpers tienen diferentes opciones, que se

Características de los FileHelpers

- → Funciona para .NET en las versiones 1.1, 2.0 y Compact Framework.
- → Funciona con todos los lenguajes de .NET que soporten atributos: C#, VB.NET, Delphi, etc.
- → Tiene versiones Generics para las principales clases.
- → Se puede utilizar, además, para procesar strings o cualquier otro stream.
- → Soporta completamente el formato CSV de Excel.
- → La librería tiene muchísimas opciones, revisen la documentación para conocerlas.
- → En la documentación encontrarán muchos ejemplos acerca de cómo usar la librería.

aplican tanto a las operaciones de lectura como a las de escritura. Todos los motores de procesamiento cuentan con un objeto ErrorManager, al cual le podemos indicar cómo debe reaccionar ante un error con la propiedad ErrorMode, que es un enumerado y puede recibir estos valores:

- ErrorMode.ThrowException
  Éste es el comportamiento predefinido, e indica que, cuando se encuentra un error, se levanta una excepción y se detiene el procesamiento. No es del todo adecuado, ya que perdemos los datos leídos hasta el momento y no podemos continuar.
- ErrorMode.IgnoreAndContinue
  En este modo la librería sólo devuelve los registros que no contienen errores, y descarta completamente los demás.
- ErrorMode.SaveAndContinue
  Es el comportamiento más avanzado. Funciona como el anterior, pero en vez de descartar los errores, los almacena internamente para que, luego, podamos recuperarlos.

Para recuperar estos errores usamos la propiedad ErrorCount y Errors de la clase ErrorManager, de la siguiente manera:

```
engine.ErrorManager.ErrorMode = ErrorMode.
  SaveAndContinue;
Cliente[] clientes = (Cliente[]) engine.ReadFile
  ("clientes.txt");
if (engine.ErrorManager.ErrorCount > 0)
{
  engine.ErrorManager.SaveErrors("errores.txt");
}
```

Si, por ejemplo, procesamos este archivo:

```
1732,Juan Perez,435.00,11052002
C34,Pedro Gomez,12342.30,06022004
112,Ramiro Politti,0.00,01022000
```

Nos genera el siguiente archivo:

```
2|C34,Pedro Gomez,12342.30,06022004|Error Converting
'C34' to type: 'Int32'.
```

Usos avanzados

La librería no se limita, simplemente, al procesamiento de archivos planos, sino que, además, permite importar y exportar datos de otros formatos o servidores.

Entre otros, permite obtener e insertar registros en archivos de Microsoft Excel y Access, como así también conectarse a una base de datos SQL Server y extraer o insertar registros en alguna tabla. En la documentación pueden encontrar información acerca de cómo usar estas características.

Otra funcionalidad importante introducida en las últimas versiones es la capacidad de convertir archivos de un formato al otro. Si volvemos al ejemplo inicial, basta con agregar a la clase de ancho fijo el siguiente método:

```
[TransformToRecord(typeof(Cliente2))]
public Cliente2 CrearSimilar()
{
  Cliente2 res = new Cliente2();
  res.Codigo = this.Codigo;
  res.Nombre = this.Nombre;
  res.Saldo = this.Saldo;
  res.FechaAlta = this.FechaAlta;
  return res;
}
```

Lo que hacemos es indicarles a los FileHelpers, con el atributo TransformToRecord(typeof (Cliente2)), que el método que se debe usar en caso de necesitar una conversión es el que está debajo. Luego, basta con crear una instancia del objeto resultado y copiarle los valores propios.

Entonces, para transformar un archivo entre estos dos formatos, sólo hacen falta estas dos líneas:

```
FileTransformEngine engine = new FileTransformEngine
  (typeof(Cliente), typeof(Cliente2));
engine.TransformFile1To2(@"entrada1.txt",
  "salida2.txt");
```

También tenemos la posibilidad de manejar archivos con formato del estilo maestro-detalle. La clase responsable de hacerlo es MasterDetailEngine, que está en el namespace FileHelpers.MasterDetail; en la documentación encontrarán la manera de usarlo.

Sugerencias o nuevas ideas

La librería está en constante evolución y a la espera de nuevos requerimientos, así que no duden en comunicarse conmigo para hacerme llegar sus ideas, consultas o sugerencias. También pueden ingresar en los foros de la librería, donde cada programador comparte sus experiencias y plantea sus problemas.

> Podemos encontrar XML por todos lados, pero para la comunicación entre sistemas se siguen usando los archivos de texto plano.

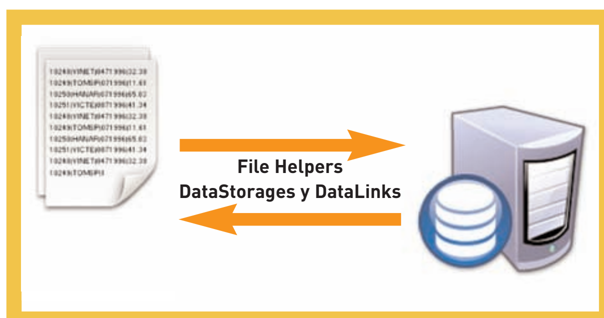
Recursos relacionados

Página oficial de los FileHelpers
→ http://filehelpers.sourceforge.net

Foros con ideas y preguntas
→ http://filehelpers.sourceforge.net/forums

Algunas guías usadas para el desarrollo de la librería:
→ http://msdn.microsoft.com/netframework/programming/classlibraries

Cualquier sugerencia es bienvenida; envíen un mail a marcosmeli@gmail.com



[Figura 2] Ejemplo del uso avanzado de lectura y escritura de datos.

USO DEL CONTROL OBJECTDATASOURCE EN ASP.NET.

Vincular objetos de negocio a controles de presentación

Veremos cómo realizar de manera clara y amigable la representación de objetos de negocio y exponer los datos que manejan junto con la funcionalidad, enlazándolos a un control de interfaz gráfica.

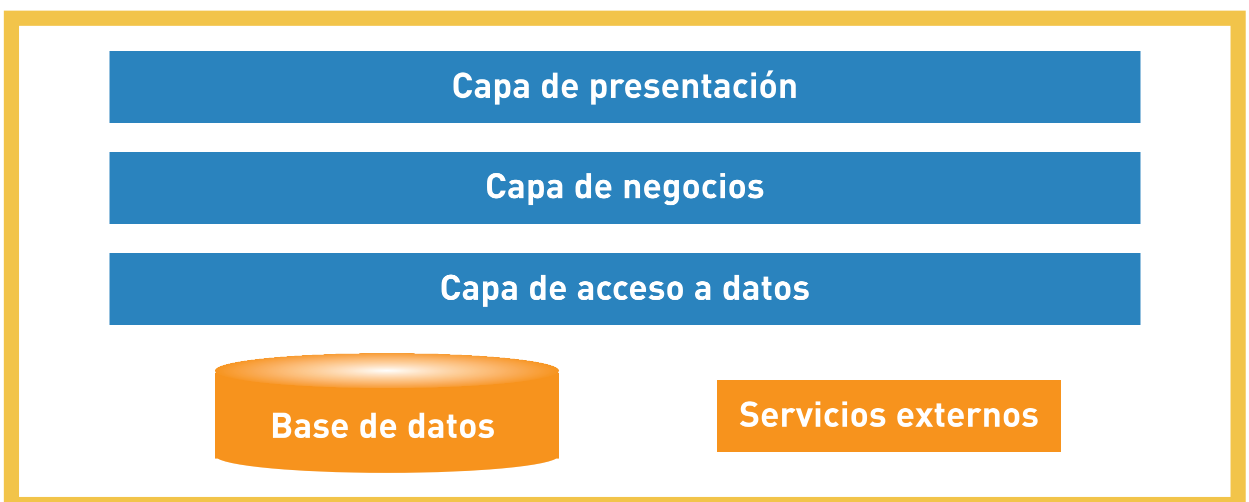
En este artículo trataremos una forma de vincular los controles con interfaz gráfica de ASP.NET 2.0 a un objeto de negocio, de forma muy simple y limpia, prácticamente sin tener que escribir código. Esto otorga una manera muy prolija de interconectar controles a los datos mediante una capa de negocio intermedia, entre la capa de presentación y el origen de datos.

Morfología de la solución

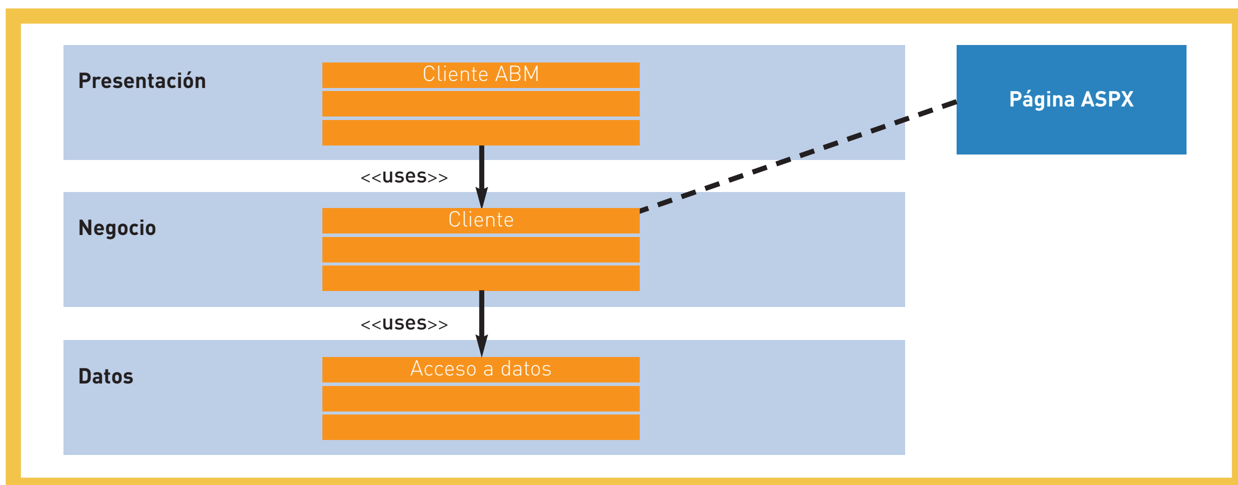
Siempre que pensamos en una aplicación escalable, lo primero que nos planteamos es de qué manera estará conformada la estructura, o si se quiere, el esqueleto de la aplicación. Este planteo lleva a que el equipo de arquitectos distribuya la estructura en diferentes capas (tiers), que cumplen una función específica dentro de la aplicación. Por ejemplo, una aplicación, básicamente, contará con una capa de presentación, que muestra la información que el usuario necesita por pantalla y la funcionalidad establecida para el tipo de información que va a mostrar; una capa de negocio, encargada de realizar todas las reglas de negocio inherentes a la solución que está dando la aplicación; y, por último, una capa de acceso a datos, que se ocupa de toda la operatoria de administración de los datos que se persistan en la base.

Hasta antes de la versión 2 de .NET, la forma de enlazar los controles de presentación a los objetos de negocio no estaba bien definida. La manera de hacerlo era creando la clase que representaba al objeto de negocio con sus funciones y propiedades, y, una vez que el objeto de negocio estaba definido, se lo utilizaba en la capa de presentación, enlazándolo directamente a los controles. De este modo, de acuerdo con lo que uno deseaba hacer, había que programar en un determinado manejador (handler) del control de presentación la funcionalidad para realizar una determinada acción; luego, para volver a presentar los datos, era necesario volver a enlazarlos al control de presentación mediante el método DataBind. Esta práctica, si bien funcionaba, obligaba al desarrollador a agregar más lógica de la deseada en la capa de presentación, puesto que, en cada manejador del control de presentación, era necesario poner la lógica que permitiera ejecutar las acciones de negocio sobre el origen de datos, como insertar un nuevo registro, actualizar uno existente, borrar o realizar una selección que siguiera cierto criterio de búsqueda, etc.

Hoy en día, en la versión 2 de ASP.NET nos encontramos con un nuevo control denominado ObjectDataSource, que opera en la capa de presentación. No



[Figura 1] Representación básica de la separación por capas de una aplicación.



[Figura 2] Páginas de presentación acoplada a los objetos de negocio.

posee una interfaz gráfica, o sea, no expone por sí solo información al cliente, pero se encarga de establecer el vínculo (binding) entre un control determinado y un objeto de negocio. Es así que, de manera declarativa, permite que un control realice acciones sobre un objeto de negocio, sin necesidad de código ni vínculos entre el control y el objeto.

El ObjectDataSource funciona, internamente, mediante reflection, por medio del cual crea una instancia del objeto de negocio para poder invocar los métodos necesarios para realizar las operatorias de negocio relacionadas a la inserción, actualización, borrado y selección de información del origen de datos.

Para obtener información sobre un objeto de negocio particular, cuenta con la propiedad **TypeName**, que recibe el nombre del tipo de objeto de negocio que utilizará para vincular al control. Por ejemplo, si el objeto de negocio se llama Clients, y se encuentra en el espacio de nombre Product.Customer, a la propiedad TypeName hay que pasar el FQN (*Fully Qualify Name*) del objeto, siendo éste "Product.Customer.Clients". Una vez pasado el nombre del tipo de objeto de negocio, hay que definir las operaciones que maneja dicho objeto. Para el fin antes mencionado, ObjectDataSource cuenta con cuatro propiedades más: **SelectMethod**, **UpdateMethod**, **DeleteMethod**, **InsertMethod**. Cada una de ellas ejecuta los métodos correspondientes a cada acción en el objeto de negocio.

La vinculación del ObjectDataSource al control se realiza mediante la propiedad "**datasourceid**" del control que necesita utilizar el ObjectDataSource.

Nota: Un dato importante de esta forma de vincular controles a objetos de negocio es que el DataBind esta implícito; por lo tanto, no hace falta que el programador tenga que pensar en vincular los datos nuevamente al finalizar una operación.

Al devolver información, la propiedad SelectMethod del ObjectDataSorce lo hace a través de un DataSet o de una clase que implementa la interfaz IEnumerable. De no ser así, el objeto de negocio es encapsulado por el RunTime a una colección de tipo IEnumerable. Si el método posee parámetros, se pueden agregar con la colección SelectParameters, que será enlazado a los valores que sea necesario pasar.

Los métodos invocados por ObjectDataSource pueden ser de instancia o estáticos. Si son estáticos, los eventos ObjectCreating, ObjectCreated y ObjectDisposing son pasados por alto sin ser ejecutados.

Nota: para enlazar parámetros, el nombre y el tipo debe coincidir con el nombre y tipo que expone la firma de la función.

Caso práctico

Una vez que hemos explicado el funcionamiento de ObjetctDataSource, es tiempo de pasar a una demostración, para tener una idea más acabada de su uso y, así, poder implementarlo. Para esto, realizaremos un ejemplo que sintetice su funcionamiento.

En este caso, crearemos un objeto de negocio que manejará el ABM (alta, baja y modificación) de clientes de una compañía, como así también, un objeto de entidad que representará a un cliente específico dentro de la capa de negocio. Por lo tanto, crearemos una clase denominada Clients, que tendrá, dentro de sus miembros, las cuatro operatorias básicas de negocio relacionadas al ABM de un cliente: GetClients, AddClient, UpdateClient y DeleteClient. También generaremos una clase Client en la que se almacenará la información correspondiente al cliente, mediante las propiedades Id, FirstName, LastName y Address.

```
namespace Product.Customer
{
 public class Clients
 {
 public Clients()
 {
 }
 public Client[] GetClients()
 {
     Client[] client = new Client[…];
     //TODO: objeto de acceso a datos.
     return client;
 }
 public void AddClient(string firstname, string
   lastname, string address)
 {
     Client client = new Client();
     client.FirstName = firstname;
     client.LastName = lastname;
     client.Address = address;
     AddClient(client);
```

```
    }
    public void AddClient(Client client)
    {
        //TODO: objeto de acceso a datos para inserción.
    }
    public void UpdateClient(int id, string firstname,
            string lastname, string address)
    {
        Client client = new Client();
        client.Id = id;
        client.FirstName = firstname;
        client.LastName = lastname;
        client.Address = address;
        UpdateClient(client);
    }
    public void UpdateClient(Client client)
    {
     //TODO: objeto de acceso a datos para actualización.
    }
    public void DeleteClient(int id)
    {
     //TODO: objeto de acceso a datos para borrado.
    }
    }
    public class Client
    {
     …
    }
```

Una vez creado el objeto entidad y el objeto de negocio en su capa correspondiente –el cual asumimos que trabaja contra la capa de acceso a datos para realizar la persistencia, modificación, borrado y selección–, pasaremos a crear la página ASPX que contendrá tanto el control de presentación de datos, como el ObjectDataSource que vinculará el objeto de negocio al control de presentación. Para hacerlo, seguimos este procedimiento. En primer término, agregamos un control que soporte el enlace con ObjectDataSource. Para saber si ese control soporta el enlace, sólo tenemos que ver si posee la propiedad DataSourceID; de ser así, significa que lo soporta. Para representar nuestro ejemplo usaremos el control FormView, que permite listar un conjunto de registros, pero de a uno por vez, mediante el uso de una plantilla que puede estar constituida por subcontroles, como Labels, TextBox, etc.

```
<asp:FormView ID="frmViewCustomer" runat="server"
DataSourceID="objDs" DataKeyNames="Id"
AllowPaging="true">
<ItemTemplate>
<table>
<tr><td>
      <asp:Button ID="btnEdit" runat="server"
      Text="Editar" CommandName="edit" />
         <asp:Button ID="btnNew" runat="server"
         Text="Nuevo" CommandName="new" />
            <asp:Button ID="btnDelete"
            runat="server" Text="Delete"
            CommandName="delete" />
         </td></tr>
         <tr><td>
            Nombre:
            <asp:Label runat="server" ID="firstName"
            Text='<%# Bind("FirstName") %>'></asp:Label>
         </td></tr>
         <tr><td>
            Apellido:
            <asp:Label runat="server" ID="lastName"
            Text='<%# Bind("LastName") %>'></asp:Label>
         </td></tr>
         <tr><td>
            Dirección:
            <asp:Label runat="server" ID="Address"
            Text='<%# Bind("Address") %>'></asp:Label>
         </td></tr>
         </table>
</ItemTemplate>
<EditItemTemplate>
…
</EditItemTemplate>
<InsertItemTemplate>
…
</InsertItemTemplate>
</asp:FormView>
```

Nótese que, en forma declarativa, en el tag inicial asp:FormView está declarada la propiedad DataSourceID, a la cual se le asigna el nombre objDs. Éste es el que se le asignó al control ObjectDataSource, que se encargará de enlazar el control de presentación (FormView) al objeto de negocio Clients. A continuación, vemos de qué manera realizar el vínculo entre el objeto de negocio y el control ObjectDataSource:

```
 <asp:ObjectDataSource ID="objDs" runat="server"
     SelectMethod="GetClients"
     UpdateMethod="UpdateClient"
     DeleteMethod="DeleteClient"
     InsertMethod="AddClient"
     TypeName="Product.Customer.Clients">
 <InsertParameters>
```

Propiedad	Descripción
SelectMethod	Carga u obtiene el nombre del método que invoca el ObjectDataSource para obtener información del origen de datos.
UpdateMethod	Carga u obtiene el nombre del método que invoca el ObjectDataSource para actualizar información.
DeleteMethod	Carga u obtiene el nombre del método que invoca el ObjectDataSource para eliminar información del origen e datos.
InsertMethod	Carga u obtiene el nombre del método que invoca el ObjectDataSource para insertar nuevos registros al origen de datos.

```
    <asp:Parameter Name="firstname" Type="string" />
    <asp:Parameter Name="lastname" Type="string" />
    <asp:Parameter Name="address" Type="string" />
   </InsertParameters>
   <UpdateParameters>
    <asp:Parameter Name="id" Type="Int32" />
    <asp:Parameter Name="firstname" Type="string" />
    <asp:Parameter Name="lastname" Type="string" />
    <asp:Parameter Name="address" Type="string" />
   </UpdateParameters>
   <DeleteParameters>
    <asp:Parameter Name="id" Type="int32" />
   </DeleteParameters>
   </asp:ObjectDataSource>
```

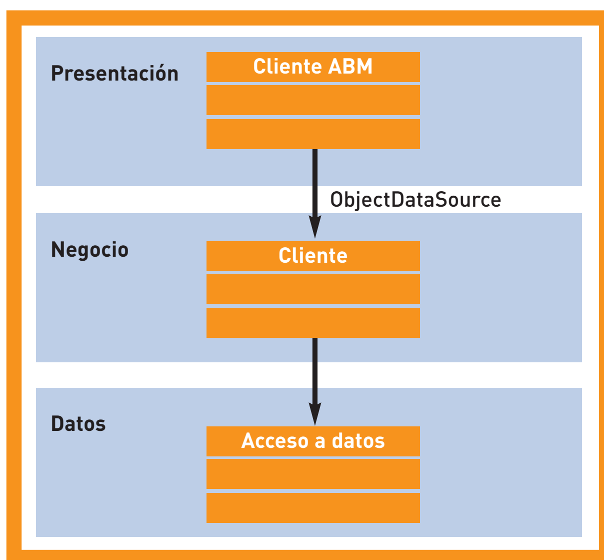
El control ObjectDataSource posee propiedades para vincular el control al objeto de negocio y, además, a los miembros del objeto de negocio; algunas de ellas son: SelectMethod, UpdateMethod, DeleteMethod, InsertMethod y TypeName. La última propiedad que se muestra, TypeName, se encarga de enlazar el objeto de negocio al ObjectDataSource; el resto vincula cada miembro del objeto de negocio a una acción específica.

Además de la vinculación al objeto de negocio, ObjectDataSource posee subtags, como InserParameters, que son utilizados para relacionar, a una determinada acción, la firma de la función que se utilizará para efectuar una acción específica.

Eventos ObjectCreating, ObjectCreated y ObjectDisposing

Al momento de utilizar el control ObjectDataSource, hay que tener en cuenta que cada invocación que hace el control a un método en particular genera un ciclo de vida del objeto en forma independiente respecto de la siguiente invocación; o sea, por cada llamado se creará el objeto, lo utilizará el ObjectDataSource y, luego, el objeto de negocio será destruido. Por lo tanto, si el diseño de un objeto de negocio fue pensado para realizar una gran cantidad de tareas –lo cual implica que sea grande y pesado para procesar–, tal vez el uso del ObjectDataSource deje de ser una buena forma para tratar el tema de acoplamiento entre la capa de presentación y la de negocio. O, quizá, sea momento de volver a pensar en el diseño de los objetos de negocio, para hacerlos más modulares y, como consecuencia, más pequeños. En fin, en caso de que no sea posible rediseñar un objeto de negocio pesado y, aun así, se quiera utilizar ObjectDataSource, una de las maneras de controlar el ciclo de vida de los objetos que crea el objeto de negocio podría ser mediante el uso de los enventos ObjectCreating, ObjectCreated y ObjectDisposing, que permiten controlar el ciclo de vida de los objetos que son creados por ObjectDataSource para invocar su funcionalidad de negocio.

La manera en que funcionan estos eventos es un tema bastante largo como para incluirlo en unas pocas líneas, pero, por lo menos, tengan en cuenta que hay una solución al problema, aunque la desventaja que tiene respecto del uso de objetos de negocio pequeños es que la forma de uso declarativa de ObjectDataSource ya dejaría de servir. Mejor dicho, ya no podría ser la única forma de encarar la solución, puesto que es necesario programar los manejadores (handler) de los eventos para tener control sobre el ciclo de vida de los objetos en cuestión mediante el uso de una caché que guarde los objetos usados y, por medio de estos eventos, preguntar si el objeto ya existe. De existir, hay que recuperarlo y pasárselo al ObjectDataSource; de lo contrario, habrá que crear uno nuevo.



[Figura 3] Desacople de la capa de presentación respecto de la capa de negocio mediante el control ObjectDataSource.

Recordar que...

El ObjectDataSource es un control que funciona muy bien, aunque, al momento de usarlo, como programadores, deberíamos tener en cuenta ciertos aspectos que harán a su buen desempeño y evitarán cualquier posible inconveniente que podría surgir como consecuencia de su uso.

En primer lugar, es preciso saber que el ObjectDataSource crea y destruye las instancias del objeto de negocio creado por cada método llamado. Esto significa que no mantiene en memoria el objeto de negocio por el tiempo de vida del request que se genera como consecuencia de una solicitud al Server. Por lo tanto, es recomendable usarlo sólo con objetos de negocio que no sean demasiado pesados. La alternativa para mantener el control sobre el ciclo de vida de los objetos es utilizar los eventos ObjectCreating, ObjectCreated y ObjectDisposing.

Conclusión

El uso de ObjectDataSource es una buena opción, porque deja el código de la capa de presentación mucho más limpio y, por lo tanto, más amigable al momento de realizar el mantenimiento del producto, considerando que la vinculación entre el objeto de negocio y el control de presentación puede realizarse íntegramente en forma declarativa.

Evento	Descripción
ObjectCreating	Ocurre antes de crear el objeto que es identificado en la propiedad TypeName.
ObjectCreated	Ocurre después de crear el objeto identificado en la propiedad TypeName.
ObjectDisposing	Ocurre antes de destruir el objeto identificado en la propiedad TypeName.

USO DE METADATOS

Reflection con C#

Veremos cómo descubrir información sobre un programa con el uso de la reflexión en las herramientas de creación de código.

Muchos se preguntarán, ¿qué son los metadatos? Trataré de explicarlo brevemente.

Se trata de datos altamente estructurados que describen la información, el contenido, la calidad, la condición y otras de sus características. Sin ir más lejos, es "información sobre información" o "datos sobre los datos".

Las técnicas de reflexión resultan especialmente útiles en el caso de las herramientas de diseño, que soportan la construcción automatizada de código basándose en las selecciones realizadas por el usuario a partir de los metadatos de los tipos empleados.

La idea de la reflexión nació ante la necesidad de descubrir información de un programa. Gracias a las APIs de reflexión de .NET, podremos saber cómo es la jerarquía interna de un software.

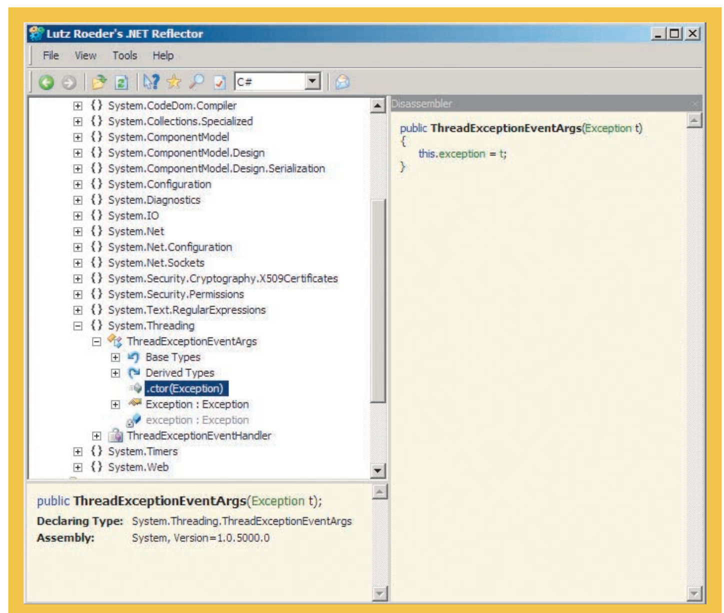
Reflector

Veremos un ejemplo práctico de cómo usar la reflexión. Hay otras herramientas más interesantes que proporcionan hasta algo del código fuente de un programa compilado, lo cual puede ser peligroso. Una de las herramientas se llama **Reflector** y se puede bajar gratuitamente desde www.aisto.com/roeder/dotnet. Reflector es un navegador de clases para componentes .NET. Soporta vistas a nivel de espacios de nombres y ensamblados, búsqueda de tipos y miembros, además de documentación XML, árboles de llamadas y gráficos. También es decompilador de Visual Basic, Delphi y C#, árboles de dependencias y más.

Ejemplo práctico

Veremos un ejemplo sencillo en el que trataremos de identificar todas las clases, tipos, delegados y algunos otros datos más que el programa contiene, informando por pantalla, en consola. Empezaremos por el primer archivo, que deja una clase disponible con todo tipo de miembros de clase para hacer actuar una reflexión sobre ella; ésta es su única misión:

```
public class Reflected
{
   public     int         MyField;
   protected ArrayList Array;        // aca defi-
nimos el array

   public Reflected()
```



[Figura 1] Aplicación para hacer reflexión de manera muy práctica y rápida. Incluye decompilador en varios lenguajes, como Delphi, C# y Visual Basic .NET.

```
   {
      Array = new ArrayList();   // creamos el objeto
      Array.Add("Datos que iran en el array"); // agregamos un dato
   }
   public float MiPropiedad
   {
      get
      {
         return MiEvento();
      }
```

Ensamblados y nombres de espacio

Para compilar correctamente, necesitamos dos cosas: una referencia física hacia el ensamblado y una referencia a un nombre de espacio en el código fuente. Ambas son requeridas por el compilador para poder identificar exitosamente y validar sus tipos en el código compilado. Para proveer de un ensamblado físico tenemos que usar la propiedad CompilerParameters.ReferencedAssemblies. Ésta apunta hacia la DLL o EXE que contiene los tipos que queremos incluir en el código.También debemos agregar las importaciones correctas a los nombres de espacio. Debemos recordar que podemos tener más de un nombre de espacio que precisemos incluir para un solo ensamblado.

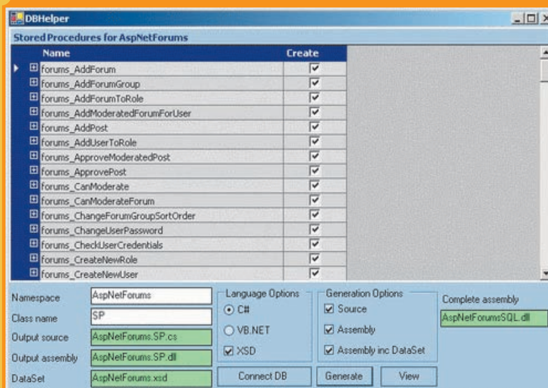
```
    }

    public object this[int indice]
    {
        get                          // el método get
        {
            if (indice <= indice)
            {
        return Array[indice]; //obtenemos el valor
                    //segun el valor del indice
            }
            else
            {
                return null;
            }
        }
        set                          // el método set
        {
            Array.Add(value);        // agregamos el valor
        }
    }

    public float MyInstanceMethod()
    {
        Console.WriteLine("Invocando Instancia MyMethod.");

        return 0.02f;
    }

    public static float MyStaticMethod()
    {
        Console.WriteLine("Invocando MyMethod Estático.");

        return 0.02f;
    }

    public delegate float Delegado();

    public event Delegado MiEvento =
        new Delegado(MyStaticMethod);

    public enum Enum { valOne, valTwo, valThree };
}
```

Como vemos, el código anterior tiene una clase principal llamada Reflected, en la que residen algunos de sus componentes, como un delegado, propiedades, métodos, enumerativos y un objeto array. Éste será el listado, es decir, el programa que será analizado y se "mostrará" a sí mismo.

Ahora veamos el siguiente código:

```
class Reflecting
  // una nueva clase que hará el trabajo "sucio"
{
    static void Main(string[] args)
    {
        Reflecting reflect = new Reflecting();

        Assembly Ensamblado =
            Assembly.LoadFrom("Reflecting.exe");

        reflect.GetReflectionInfo(Ensamblado);

        Console.ReadLine();
    }

    void GetReflectionInfo(Assembly Ensamblado)
    {
        Type[] typeArr = Ensamblado.GetTypes();

        foreach (Type type in typeArr)
        {
            Console.WriteLine("\nType: {0}\n", type.FullName);

            ConstructorInfo[] MyConstructors =
                type.GetConstructors();
            foreach (ConstructorInfo constructor
                in MyConstructors)
            {
                Console.WriteLine("\tConstructor:
                    {0}", constructor.ToString());
            }
            Console.WriteLine();

            FieldInfo[] MyFields = type.GetFields();
            foreach (FieldInfo field in MyFields)
            {
                Console.WriteLine("\tField:
                    {0}", field.ToString());
            }
            Console.WriteLine();

            MethodInfo[] MyMethods = type.GetMethods();
            foreach (MethodInfo method in MyMethods)
            {
                Console.WriteLine("\tMethod:
                    {0}", method.ToString());
            }
            Console.WriteLine();

            PropertyInfo[] MyProperties =
                type.GetProperties();
```

Más reflexión



La reflexión, además de aplicarse a programas, también permite hacerlo a bases de datos. En www.codeproject.com/cs/database/dbhelper.asp, encontraremos una aplicación, llamada DBHelper, que refleja los datos de una base y genera el código para realizar operaciones básicas sobre ella.

```
            foreach (PropertyInfo property in MyProperties)
            {
                Console.WriteLine("\tProperty:
                    {0}", property.ToString());
            }
            Console.WriteLine();

            EventInfo[] MyEvents = type.GetEvents();
            foreach (EventInfo anEvent in MyEvents)
            {
                Console.WriteLine("\tEvent:
                    {0}", anEvent.ToString());
            }
            Console.WriteLine();
        }
    }
```

En el método Main creamos un objeto llamado reflecting, al cual llamaremos reflect. También generaremos un objeto Assembly, que usaremos con el método LoadFrom para cargar el EXE que vamos a crear, denominado Reflecting.exe. Es necesario hacer esto para, luego, obtener los datos del ejecutable mediante el método GetReflectionInfo, al cual le pasamos como parámetro el Ensamblado.

A continuación, necesitamos crear un array para "absorber" los tipos; entonces, con el método GetTypes rellenaremos el array Ensamblado.

El corazón del proceso se ejecuta en cada uno de los foreach, los cuales van obteniendo la información de cada tipo, según qué tipo sea.

Miremos una parte del foreach, para entender todo el resto:

```
ConstructorInfo[] MyConstructors =
  type.GetConstructors();
foreach (ConstructorInfo constructor
  in MyConstructors)
{
  Console.WriteLine("\tConstructor:
  {0}", constructor.ToString());
}
```

Vemos que obtiene el tipo constructor, por ejemplo, y luego, en el foreach pregunta si la información devuelta es del tipo constructor. Si es así, escribe en consola la información perteneciente a ese constructor. Así se hace con cada propiedad, método, clase, objeto, etc.

La salida del programa es como la siguiente:

```
Type: Reflecting

  Constructor: Void .ctor()

  Method: Int32 GetHashCode()
  Method: Bolean Equals(System.Object)
  Method: System.String ToString()
  Method: System.Type GetType()

Type: Reflected
```

También podemos utilizar el método GetModules() sobre el tipo Assembly, para obtener todos los módulos relacionados con ese ensamblado y cargarlos como hicimos antes en un array de tipos.

Activación dinámica de código

¿Para qué sirve esta técnica? Sin ir más lejos, puede servir para crear código dinámicamente en situaciones que requieran evaluar inconvenientes no estudiados en etapa de diseño. Esto da la posibilidad de generar ensamblados posteriores al principal y, así, agregar funcionalidad extra a un sistema de manera dinámica.

Por ejemplo, el protocolo SOAP (*Simple Object Access Protocol*), que es independiente del protocolo de transporte, se utiliza mucho con el http. Pero resulta que necesitamos un nuevo protocolo y, con estas técnicas, podríamos crear nuevas DLLs en las que estarían empaquetados dichos protocolos sin tener que recompilar el código. Veamos un ejemplo:

```
  void DynamicallyInvokeMembers(Assembly Ensamblado)
  {
      Type classType = Ensamblado.GetType("Reflected");

      PropertyInfo myProperty =
          classType.GetProperty("MyProperty");

      MethodInfo propGet = myProperty.GetGetMethod();

      object reflectedObject =
          Activator.CreateInstance(classType);

      propGet.Invoke(reflectedObject, null);

      MethodInfo myMethod =
          classType.GetMethod("MyInstanceMethod");

      myMethod.Invoke(reflectedObject, null);
  }
}
```

Como podemos notar, el método DynamicallyInvokeMembers() usa el objeto Assembly para obtener el objeto Type de la clase Reflected. Luego, utiliza el objeto Type para obtener la propiedad MyProperty.

A continuación, se obtiene un objeto MethodInfo llamando al método GetGetMethod() del objeto PropertyInfo. El método GetGetMethod() recupera una copia del método get de una propiedad, el cual, para nuestro objetivo de la reflexión, es considerado, a su vez, como un método.

Luego, la clase Reflected se instancia usando el método Activator.CreateInstance(). El objeto instanciado se utiliza, entonces, como primer parámetro del método Invoke() del objeto MethodInfo. Esto identifica el objeto sobre el que se debe invocar el método. El segundo parámetro del método Invoke() es la lista de parámetros que se deben enviar al método, que sería un array de objetos si hubiese parámetros. En nuestro caso, no hay parámetros para enviar al método, de manera que el segundo parámetro del método Invoke() se configura como null.

Las dos líneas siguientes del código muestran cómo se invoca dinámicamente un método de instancia. La sintaxis es la misma que la que acabamos de ver para el medio de acceso get de la propiedad. Sin embargo, aquí no es necesario el paso intermedio que se usa para las propiedades, sino que el método puede obtenerse directamente con GetMethod() del objeto Type.

Conclusión

Hemos visto que la reflexión nos permite descubrir información bastante avanzada sobre un programa, y el mismo Microsoft ha lanzado productos destinados a proteger los ensamblados. Este tipo de "investigaciones" facilita muchas tareas, pero también hace que terceros puedan mirar nuestro código y hasta robar partes de él.

Por otro lado, podemos activar código dinámicamente, crearlo y guardar nuevos programas generados según nuevas condiciones que se presenten dentro de un sistema dado, como compiladores y motores de guiones.