Guillermo "Guille" Som

Interfaces

En el artículo sobre las interfaces de *dnm.inicio.fundamentos* publicado en el nº 16 de **dotNetManía**, explicamos lo que son las interfaces y vimos algunos ejemplos de cómo usarlas. En esta ocasión veremos más ejemplos, con idea de que nos quede claro cómo utilizar las interfaces en nuestros proyectos, aunque en esta ocasión vamos a usar las interfaces de una forma diferente a lo "habitual", con la intención de que nos hagamos una idea de lo útiles que pueden llegar a ser estos tipos de datos.

» Usar las interfaces como parámetro de métodos

Las interfaces no las vamos a utilizar siempre desde la perspectiva de ofrecer funcionalidades "compatibles" con las clases de .NET Framework, ni con otras de nuestra propia cosecha, al menos desde el punto de vista de que al implementar alguna interfaz, nuestra clase tenga la "funcionalidad" aportada por dicha interfaz.

Por ejemplo, si necesitamos pasar parámetros a distintos métodos, pero no sólo nos interesa indicar el contenido de unas cadenas o unos valores numéricos, sino que también queremos pasar algo de "acción", es decir, algún método que actúe sobre esos parámetros o sobre los valores que internamente el método tenga que usar..., en estos casos también nos puede interesar utilizar las interfaces, ya que éstas nos proporcionan una forma de utilizar objetos anónimos (o previamente indefinidos), los cuales dejarán el anonimato en el momento que nuestro código ejecute dicho método.

Aclaremos un poco todo esto para que nos entendamos mejor:

Si tenemos un método declarado de esta forma:

Para VB: `Metodo(parámetro As Interfaz)`

Para C#: `Metodo(Interfaz parámetro)`

Es obvio que `parámetro` es del tipo de una interfaz, por tanto, el argumento real que pasaremos a este método será de un objeto que implemente la interfaz indicada en el tipo del parámetro. Esa interfaz tendrá cierta funcionalidad y posiblemente algunas propiedades. Todo eso lo podemos usar en ese método sin importarnos qué objeto es el que utilizamos a la hora de llamarlo, o para decirlo de otra forma: sin importarnos de qué tipo es el objeto pasado como argumento.

Si usamos las interfaces de esta forma, realmente los tipos que la implementen no lo harán para tener la funcionalidad de la interfaz, (aunque también), sino porque así podrán usarse en otros contextos. En el fondo es lo mismo, aunque al usarlas de este modo ampliamos la forma de aprovecharnos de las interfaces y de la filosofía de las mismas: proporcionar funcionalidad anónima y proveer de polimorfismo a nuestras clases y estructuras.

Dos ejemplos de interfaces como parámetros

A continuación veremos un par de ejemplos en los que utilizaremos esta otra forma de darle utilidad a las interfaces. El primer ejemplo ya lo vimos en el nº 7 de **dotNetManía** (pág. 24), en el que usábamos una clase específica para clasificar tipos que no estaban "preparados" para ser clasificados, (no vamos a entrar en detalles sobre cómo hacer una clase que pueda ser clasificada, ya que de eso nos ocupamos en el citado número de **dotNetManía**.)

El caso era que teníamos una clase "normal" que no estaba preparada para ser clasificada por el propio .NET, es decir, no implementaba la interfaz `IComparable`. Pero queríamos clasificar el contenido de una colección que contenía objetos de ese tipo no clasificable. La solución fue usar una de las sobrecargas del método `Sort`, al que se le pasa como argumento una clase que implemente una interfaz (`IComparer`). La clase pasada como argumento tiene que implementar dicha interfaz, y se encargará de clasificar (o comparar) dos elementos de un tipo determinado.

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

Al estar declarado el parámetro como un tipo concreto de una interfaz, esa sobrecarga del método `Sort` se asegura de que el objeto usado como argumento tiene la funcionalidad que se espera, por tanto, será seguro usarlo, y se supone que hará el trabajo que debe hacer.

> **NOTA**
>
> Como ya sabemos las clases que implementan una interfaz solamente están obligadas a definir los miembros de esa interfaz, pero no hay nada que nos obligue a escribir código dentro de esos miembros. Por supuesto, implementar una interfaz para después no darle "funcionalidad" no tiene mucho sentido, pero es conveniente saber que el compilador no chequeará que esos miembros realmente hagan algo.

En el fuente 1 tenemos el código para C#[1] de la clase "intermedia" que podemos usar para clasificar objetos de un tipo que no implementa la interfaz `IComparable`. De esta clase sólo usaremos la parte expuesta por la interfaz `IComparer`, es decir, el método `Compare`; si esta clase tuviera más miembros, éstos se ignorarán, al menos al usar el objeto como argumento del método `Sort`.

```
public class ColegaComparer : IComparer
{
      public int Compare(object x, object y)
      {
            Colega c1 = ((Colega)x);
            Colega c2 = ((Colega)y);
            return string.Compare(c1.Apellidos, c2.Apellidos);
      }
}
```

Fuente 1. Código de la clase que implementa la interfaz IComparer

Pero no solo tenemos que usar interfaces definidas en el propio .NET Framework, ya que las interfaces usadas pueden ser de cualquier tipo, lo importante aquí es que el método al que queremos pasar un objeto acepte un tipo de interfaz, y, como es lógico suponer, el objeto pasado debe implementar esa interfaz.

El segundo ejemplo que usaremos para pasar interfaces a métodos, lo vamos a basar en una interfaz que podemos implementar en nuestros formularios. La razón de hacer esto es porque algunas veces nos podemos encontrar con la situación de que queremos dar cierta funcionalidad a nuestros formularios, pero puede que esa nueva funcionalidad sea diferente en cada uno de ellos.

Una de las formas de proporcionar ese nuevo funcionamiento es crear una clase basada en la clase `Windows.Forms.Form`, (que es la clase base de cualquier formulario de .NET), y añadirle la nueva funcionalidad con elementos (miembros) virtuales (reemplazables). Después, sólo tendríamos que crear nuevos formularios basados en esa clase y así dispondríamos de todas las nuevas características añadidas en nuestra clase base.

Pero es posible que la implementación (el código a usar) de esa nueva funcionalidad sea diferente dependiendo de lo que queramos que nuestro formulario haga, por tanto, no tiene mucho sentido escribir un código que después lo vamos a reemplazar en la mayoría de las veces que usemos esa clase. En este tipo de situaciones es donde nos podemos beneficiar del uso de las interfaces, ya que una interfaz no tiene código ejecutable, solamente define los miembros que tenemos que implementar, y cada implementación puede ser diferente unas de otras porque lo único importante es que (en nuestro caso) el formulario implemente esos miembros y le de la funcionalidad que creamos conveniente.

Por ejemplo, puede ser que queramos crear un formulario para realizar búsquedas, y queremos que no dependa de ningún formulario en particular. Por tanto lo podemos usar con formularios que utilicen diferentes tipos de controles para almacenar la información en la que realizaremos la búsqueda. Para darle esa independencia al formulario que pedirá los datos que queremos buscar, podemos utilizar una interfaz que defina los miembros que cada formulario tendrá que implementar, de forma que nuestro formulario de buscar utilice la parte del formulario que implementa la interfaz para llamar a los métodos.

Ni qué decir tiene que ese formulario de búsqueda no será el encargado de buscar nada, ya que no sabe dónde tendrá que realizar dicha búsqueda, por tanto serán los formularios que implementen la interfaz los que se encarguen de todo el trabajo.

En la figura 1 tenemos el formulario buscar. En el control combo estarán las últimas palabras usadas en las búsquedas anteriores.

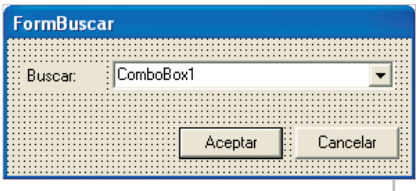

Figura 1. El formulario buscar en tiempo de diseño.

En el fuente 2 tenemos la definición de la interfaz `IBuscar` que además de un método `Buscar` al que se le pasará como argumento la palabra que hay que buscar, también define dos métodos:

- `LeerCfg` devuelve un array con las palabras usadas en búsquedas anteriores
- `GuardarCfg` tiene un parámetro en el que se indicará la palabra buscada en la última ocasión.

En el formulario buscar hemos definido un método `Show` que se usará para mostrarlo y al que habrá que indicarle un objeto que implemente la interfaz `IBuscar`.

Ese argumento se asignará a una variable interna del tipo de la interfaz, con idea de usarla para llamar a los métodos correspondientes del objeto pasado. Lo primero que hacemos, después de asignar el objeto recibido a la variable interna, es llamar al método

[1] En el texto sólo aparecen los fuentes en C# pero puede descargarse también los de VB.NET desde nuestra Web www.dotnetmania.com

```
// Interfaz para proporcionar métodos para buscar
public interface IBuscar
{
  string[] LeerCfg();
  void GuardarCfg( string palabra);
  bool Buscar( string texto);
}

// El formulario de buscar
private IBuscar elForm;
public void Show(IBuscar formulario)
{
  elForm = formulario;
  // leemos los datos de la configuración
  string[] lista = elForm.LeerCfg();
  cboBuscar.Items.Clear();
  if( lista == null )
    cboBuscar.Text = "";
  else
  {
    foreach( string s in lista)
      cboBuscar.Items.Add(s);
    cboBuscar.SelectedIndex = 0;
  }
  //
  this.Show();
}

private void btnAceptar_Click(object sender, EventArgs e)
{
  // Llamamos al formulario para que busque el texto
  // y si lo encuentra cerramos este formm en caso
  // contrario damos la oportunidad de buscar otra cosa
  if( elForm.Buscar(cboBuscar.Text) )
  {
    // Llamamos al método guardarCfg
    // por si quiere guardar la palabra buscada
    elForm.GuardarCfg(cboBuscar.Text);
    Hide();
  }
  else
    cboBuscar.Focus();
}

private void btnCancelar_Click(object sender, EventArgs e)
{
  Hide();
}
```

Fuente 2. Código de la interfaz y el formulario buscar

```
public class Form1 : System.Windows.Forms.Form, IBuscar
{

public bool Buscar(string texto)
{
  // Realizar una búsqueda en el contenido del textbox
  // Devolverá True si ha encontrado lo buscado,
  // y se encargará de resaltar el texto, etc.
  int pos = RichTextBox1.Find(texto);
  if( pos > -1 )
      RichTextBox1.Select(pos, texto.Length);
  return pos > -1;
}

public void GuardarCfg(string palabra)
{
  // El código para guardar la configuración en este formulario
  // El parámetro será la palabra que se ha indicado en buscar
  // Tendremos que decidir si la guardamos o que...
}

public String[] LeerCfg()
{
  // El código para leer la configuración en este formulario
  // y devolverla como un array de tipo String
  return new String[] {"Porque", "no engraso", "los ejes",
                       "me llaman", "abandonao", "gustan", "suenen"};
}

private void btnBuscar_Click(object sender, EventArgs e)
{
  // Mostrar el formulario de búsqueda
  FormBuscar fBuscar = new FormBuscar();
  fBuscar.Show(this);
}
```

Fuente 3. Código del formulario que implementa la interfaz.

`LeerCfg` para asignar al combo las palabras usadas con anterioridad.

Cuando el usuario pulsa en el botón "Aceptar", se llama al método `Buscar` de la interfaz, y si devuelve un valor verdadero querrá decir que el texto indicado ha sido hallado, por tanto llamamos al método `GuardarCfg` para que el formulario sepa que esa es la palabra que debe añadir a su lista de palabras internas. En caso de que devuelva un valor falso, se sigue mostrando el formulario de búsqueda hasta que se encuentre algo o se cancele.

Tal como comentamos en la nota, no tenemos por qué implementar código en los métodos definidos por la interfaz, y tal como veremos en el código implementado en el formulario que usa `IBuscar`, no se hace nada en el método `GuardarCfg`, pero como las interfaces "obligan" a que tengamos que implementar todos los miembros definidos en ellas, estaremos obligados a crear al menos el cuerpo del método para que nuestro código compile.

La aplicación de ejemplo utiliza dos formularios que implementan la interfaz `IBuscar`: en uno de ellos se utiliza un control *RichTextBox* y en el otro un control *TextBox*. Para buscar en el primero utilizamos el método `Find`, mientras que en el segundo llamamos al método `IndexOf` de la propiedad `Text` de la caja de textos. Esto simplemente es para que veamos que realmente al código del formulario buscar le da igual qué tipo de control será el que se use para hacer la búsqueda, ya que será el propio formulario el que se encargará de hacer esa búsqueda.

En el fuente 3 tenemos las implementaciones de los tres métodos de la interfaz `IBuscar`, además del método que muestra el formulario de búsqueda, al que hay que indicarle como argumento la instancia actual del formulario, aunque ese objeto se "filtrará" para dejar pasar sólo la parte del mismo que implementa la interfaz.

Esperamos que con lo aquí mostrado tengamos otra visión de cómo usar las interfaces, pero esto no lo es todo, en otra ocasión veremos cómo las interfaces tienen un papel muy importante cuando queremos usar ensamblados de .NET desde aplicaciones COM, por ejemplo desde Visual Basic 6, en las que queremos mantener la compatibilidad binaria para no forzar a recompilar los ejecutables cuando añadimos nuevos elementos a ese ensamblado, pero eso, será en otra ocasión, cuando veamos el tema de la interoperabilidad.

¡Nos vemos!