Guillermo "Guille" Som

Interfaces

Yo implemento, tu implementas,... ¡implementemos todos!

En este primer artículo de esta nueva seccción explicamos las *Interfaces*. Las interfaces son de esos elementos de .NET Framework que juegan un papel bastante importante, pero que, a pesar de estar por todas partes, aparentemente pasan desapercibidas...

» Y es cierto, tanto que juegan un parte importante como que pasan desapercibidas; de hecho las estamos usando continuamente y en muchos de los casos ni tan siquiera somos conscientes de ello. Esto es así, al menos en la mayoría de los casos, porque es el propio .NET el que juega ese papel de implementador de interfaces; nosotros simplemente las usamos, pero me atrevo a asegurar que en la mayoría de los casos no somos conscientes de que estamos trabajando codo a codo con las interfaces. Por tanto, ya va siendo hora de que sepamos que existen y cómo podemos aprovecharnos de su existencia... en el buen sentido de la palabra.

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

Las interfaces son...

En su forma más simple, las interfaces son simplemente definiciones de buenas intenciones, es decir, sólo indican cómo deberían definirse ciertos miembros de nuestras clases y estructuras. Pero si lo examinamos un poco más a fondo, realmente son más que esas buenas intenciones, ya que, cuando entran en juego, no basta con tener buenas intenciones, sino que debemos seguir al pie de la letra lo que esas interfaces definen. Y este es el punto fuerte de las interfaces, no sólo definen cómo deberían ser las cosas, sino que nos obligan a que las cosas sean de una forma determinada.

Seguramente estaremos hartos de leer que las interfaces son un contrato, es decir, son algo que cuando decidimos usarlas, debemos hacerlo al pie de la letra, no basta con tener la intención de..., sino debemos convertir esa intención en algo tangible.

Como hemos comentado, el propio .NET Framework utiliza las interfaces cuando quiere asegurarse de que algo se puede hacer, por ejemplo, si queremos clasificar elementos en un array o una colección, (ver nº 7 de **dotNetManía**), esos elementos deben implementar la interfaz `IComparable`, de esta forma, el propio .NET se asegura de que esas clases clasificables se comportarán de una forma determinada, de modo que pueda utilizarse como está previsto que se utilicen. Esa seguridad que tiene .NET de hacer las cosas como está previsto se la da el hecho de que el programador que ha creado la clase usada para ser clasificada ha firmado un contrato, ese contrato le dice que debe utilizar los miembros de la interfaz correspondiente de la forma adecuada; en este ejemplo, debe implementar el método `CompareTo` para que coincida exactamente con la forma que está definido en la interfaz `IComparable`. Por tanto, si .NET se encuentra con una clase que define los miembros de esa interfaz, tiene la certeza de utilizarlos como está previsto.

¿Qué contiene una interfaz?

Debido a la naturaleza propia de las interfaces, es decir, que solamente representan buenas intenciones, éstas no contienen código ejecutable, sólo definen qué código debe tener la clase o estructura que decida implementarla. Dicho esto, podemos concluir que las interfaces sólo definen los miembros, realmente la "firma" de los mismos, es decir, de qué tipo son, si reciben parámetros, cuántos y de qué tipo, y en el caso de las propiedades, además del tipo de las mismas, si éstas son de sólo lectura, predeterminadas (indizadores en C#), etc. Veamos un ejemplo para aclarar conceptos. En los fuentes 1 y 2 tenemos la definición de una interfaz, tanto para VB como para C#.

```
Public Interface IPrueba
    Sub Mostrar(ByVal saludo As String)
    Property Nombre() As String
    Default ReadOnly Property Item(ByVal index As Integer) As String
    Function Total() As Integer
    Event DatosCambiados(ByVal mensaje As String)
End Interface
```

Fuente 1. Definición de una interfaz en Visual Basic

```
public delegate void DatosCambiadosEventHandler(string mensaje);
public interface IPrueba
{
      void Mostrar(string saludo);
      string Nombre{get; set;}
      string this[int index]{get;}
      int Total();
      event DatosCambiadosEventHandler DatosCambiados;
}
```

Fuente 2. Definición de una interfaz en C#

Como podemos comprobar, en las dos interfaces hemos definido todos los tipos de miembros que cualquier clase o estructura pueden tener: métodos, propiedades, indizadores (propiedades predeterminadas en VB) y eventos. En el caso de los eventos, la diferencia está en cómo se definen en VB y en C#, en este último, los eventos siempre están ligados con los delegados, mientras que en Visual Basic no tiene por qué hacerse esa asociación, aunque internamente sí que están bastantes ligados con los delegados, de hecho la declaración de la interfaz del fuente 1 la podemos hacer tal como mostramos en el fuente 3.

Aunque en cualquier caso, siempre se podrán definir de la forma habitual (de VB), de hecho, cuando el propio IDE de Visual Studio crea la plantilla para implementar esta interfaz, lo hace de la misma forma que si hubiésemos usado la definición hecha en el fuente 1.

¿Cómo usar una interfaz?

Tal como podemos comprobar en los fuentes 1 y 2, al definir una interfaz sólo indicamos la "firma" que tendrán los miembros sin indicar siquiera el ámbito de éstos, ya que siempre serán públicos, de hecho no tendía ningún sentido definir miembros privados en una interfaz.

```
Public Delegate Sub DatosCambiadosEventHandler(ByVal mensaje As String)
Public Interface IPrueba
    ...resto de miembros...
    Event DatosCambiados As DatosCambiadosEventHandler
End Interface
```

Fuente 3. Definición de una interfaz de VB usando delegados

El nombre de las interfaces

Por definición, o convención, los nombres de las interfaces siempre empezarán con la letra I mayúscula seguida del nombre propiamente dicho. Aunque esto es sólo una sugerencia, ya que realmente el nombre de una interfaz puede ser cualquier nombre válido.

Una vez que hemos definido el contenido de una interfaz y queremos usarla en una clase, estructura u otra interfaz, lo haremos de forma distinta dependiendo del lenguaje. En el caso de C#, para indicar que una clase (o tipo) implementa dicha interfaz, lo haremos de la misma forma que al derivar la clase a partir de otra: usando dos puntos después del nombre del tipo, seguido con el nombre de la interfaz:

```
public class Prueba : IPrueba {...}
```

En Visual Basic utilizaremos la instrucción `Implements` seguida del nombre de la interfaz:

```
Public Class Prueba
      Implements IPrueba
      ...
End Class
```

Si estamos usando Visual Studio como editor de nuestro código, éste puede crear las plantillas de los miembros de la interfaz automáticamente, de forma que nosotros sólo tengamos que escribir el código correspondiente a cada uno de estos miembros.

La forma de definir esos miembros dependerá también del lenguaje que estemos utilizando, en el caso de C#, no hay que hacer nada especial, simplemente definir los miembros en la clase y asunto arreglado. Pero si el lenguaje usado es Visual Basic, esas definiciones hay que hacerlas de forma explícita, es decir, debemos indicar que las definiciones de esos miembros realmente son las definiciones de los miembros de la interfaz. Esto lo haremos usando, a continuación de la definición de cada uno de ellos, la instrucción `Implements` seguida del nombre de la interfaz y el miembro que implementa:

```
Public Property Nombre() As String _
            Implements IPrueba.Nombre
```

Realmente esta forma de hacer de VB puede parecer, (de hecho lo es), una forma de rizar el rizo, pero la explicación o excusa que hay (con la que nos podemos consolar), es que en VB no tenemos porqué usar el mismo nombre indicado en la interfaz, por ejemplo, el método `Mostrar` lo podríamos definir de esta otra forma:

```
Public Sub Imprimir(_
          ByVal saludo As String)_
          Implements IPrueba.Mostrar
```

En este caso, el método `Imprimir` de la clase `Prueba` realmente es la definición del método `Mostrar` de la interfaz `IPrueba`. Sobre esto, aunque está bien que podamos hacerlo, si queremos mantener nuestras mentes claras, sobre todo la de los que prefieren usar C#, deberíamos respetar siempre el nombre indicado en la interfaz.

Múltiples interfaces en nuestros tipos

.NET es un lenguaje de herencia simple, es decir, una clase de .NET no se puede derivar de más de una clase, (salvo con la excepción de la herencia implícita de la clase `Object`), pero en el caso de las interfaces, podemos decir que los lenguajes de .NET son de herencia múltiple. Esto es así porque una clase (o tipo) puede implementar más de una interfaz; por supuesto, en ese caso, siempre deberemos definir todos los miembros de todas las interfaces que implemente.

Para indicar que una clase implementa más de una interfaz lo haremos de la siguiente forma. Para C#:

```
public class Prueba2 : IPrueba,
                       IPrueba2 {...}
```

Y de esta otra para VB:

```
Public Class Prueba2
     Implements IPrueba, IPrueba2
     ...
End Class
```

Múltiples interfaces con miembros idénticos

Un problema con el que nos podemos encontrar es que dos interfaces distintas tengan miembros con la misma firma (y el mismo nombre). Bueno, problema, lo que se dice problema, no lo es, ya que la definición la haremos como de costumbre, si bien aquí volvemos a encontrarnos con la forma que tiene cada lenguaje de hacer frente a esa duplicidad.

En el caso de C#, no tenemos que hacer nada en especial, sólo definir un miembro que cumpla con la firma correspondiente y ese mismo miembro lo podemos usar para las dos interfaces.

```
public void Mostrar(string saludo)
{
     // Esta implementación servirá
     // para las dos interfaces
```

En el caso de Visual Basic, podemos hacerlo de dos formas:

La primera, (que es la que usa el IDE de Visual Studio), es creando dos miembros para implementar cada uno de los de cada interfaz; por supuesto esos miembros deben tener nombres diferentes:

```
Public Sub Mostrar(_
  ByVal saludo As String) _
     Implements IPrueba.Mostrar

Public Sub Mostrar1(_
  ByVal saludo As String)
     Implements IPrueba2.Mostrar
```

La segunda es definiendo sólo un miembro e indicando que ese único miembro implementa las dos interfaces:

```
Public Sub Mostrar(_
      ByVal saludo As String) _
         Implements IPrueba.Mostrar,_
                    IPrueba2.Mostrar
```

Por supuesto, la forma que elijamos dependerá de cómo queramos implementar estos miembros, es decir, si queremos que compartan el mismo código o que ese código sea independiente para cada implementación.

Las interfaces sólo definen los miembros, realmente la "firma" de los mismos, es decir, de qué tipo son, si reciben parámetros, cuántos y de qué tipo, y en el caso de las propiedades, además del tipo de las mismas, si éstas son de sólo lectura, predeterminadas (indizadores en C#)

Implementaciones privadas, ¿realmente son privadas?

El problema con el que nos podemos encontrar es si esta misma forma de actuar, de tener definiciones diferentes para cada implementación, la queremos implementar en C#. Ya que este lenguaje no permite indicar que cada implementación es independiente una de la otra. Al menos si queremos que tanto una como otra sean públicas y queremos que ambas formen parte de la interfaz pública de la clase.

La solución es definir cada miembro independiente para que sólo puedan ser accedidos usando objetos del tipo de cada interfaz (o que uno sea parte de la clase y de una de las interfaces y el otro sólo lo sea de la otra interfaz).

Esto lo podemos hacer definiendo estos miembros de la siguiente forma. En el fuente 4 se muestra cómo hacerlo para que la implementación de **IPrueba** sea parte de la clase y que la implementación de **IPrueba2** sólo pueda ser accedida mediante un objeto del tipo de esa interfaz.

```
public void Mostrar(string saludo)
{
      // Esta implementación servirá para la interfaz IPrueba
      Console.WriteLine(saludo + ", " + Nombre);
}

void IPrueba2.Mostrar(string saludo)
{
      // Esta implementación solo es para IPrueba2
      Console.WriteLine("2: " + saludo + ", " + Nombre);
}
```

Fuente 4. Implementación privada en C# de los miembros de una interfaz

Igualmente si sólo queremos que ambas implementaciones sean privadas, es decir, que no formen parte de la propia clase que las define, sino que sólo sean accedidas desde un objeto del tipo de la interfaz, las declaraciones las haremos usando la segunda forma mostrada en el fuente 4.

En el caso de Visual Basic, como hemos visto, podemos hacerlo de las dos formas mostradas anteriormente, pero si queremos que tengan la misma funcionalidad "privada" que hemos visto en C#, podemos declarar esos miembros como privados, de forma que no formen parte de la interfaz pública de la clase, sino que sólo podamos usarlas desde objetos del mismo tipo que las interfaces. En el siguiente fuente podemos ver un ejemplo de esto que comentamos:

```
Private Sub Mostrar1(ByVal saludo As String) _
                  Implements IPrueba2.Mostrar
```

.NET es un lenguaje de herencia simple, es decir, una clase de .NET no se puede derivar de más de una clase, pero en el caso de las interfaces, podemos decir que los lenguajes de .NET son de herencia múltiple

Como hemos comentado, aunque estos miembros estén definidos como privados, debemos saber que siempre formarán parte de la interfaz que implementa la clase, por tanto podremos acceder a ellos por medio de un objeto de la interfaz, tal como podemos ver en el código de los fuentes 5 y 6.

```
Dim pr2 As New Prueba2
pr2.Nombre = "Pepe"
pr2.Mostrar("Desde la clase Prueba2: Hola")
Dim ipr1 As IPrueba
Dim ipr2 As IPrueba2
ipr2 = pr2
ipr2.Mostrar("Desde la interfaz IPrueba2: Hola")
ipr1 = pr2
ipr1.Mostrar("Desde la interfaz IPrueba: Hola")
```

Fuente 5. Acceder a los miembros de una clase por medio de un objeto de la interfaz (VB)

```
Prueba2 pr2 = new Prueba2();
pr2.Nombre = "Pepe";
pr2.Mostrar("Desde la clase Prueba2: Hola");
IPrueba ipr1;
IPrueba2 ipr2;
ipr2 = pr2;
ipr2.Mostrar("Desde la interfaz IPrueba2: Hola");
ipr1 = pr2;
ipr1.Mostrar("Desde la interfaz IPrueba: Hola");
```

Fuente 6. Acceder a los miembros de una clase por medio de un objeto de la interfaz (C#)

Conclusión

Como hemos podido comprobar en lo expuesto aquí, las interfaces son una de las formas que tienen los lenguajes para que nuestras clases puedan ofrecer cierta funcionalidad y que dicha funcionalidad pueda ser compartida por clases distintas. Debido a que las interfaces nos obligan a seguir unas normas o formas a la hora de definir ciertos miembros en nuestros tipos, podemos usar esta característica para asegurar que nuestras clases utilizan uno de los pilares de la programación orientada a objetos: el *polimorfismo*, ya que las interfaces le dan a nuestras clases una forma de utilización "medio-anónima", al menos en el sentido de que si utilizamos una interfaz que nuestra clase implementa, podemos acceder a los miembros implementados por esa interfaz, sin necesidad de saber que estamos usando un objeto determinado de una clase en particular.

En otra ocasión veremos de una forma algo más práctica cómo utilizar las interfaces y de paso algunos consejos que siempre vendrán bien para sacar el provecho de este tipo de datos que .NET pone a nuestra disposición.