Guillermo "Guille" Som

Relación entre delegados y eventos

El avisador que te avise, buen avisador será

En el número anterior vimos con detalle casi todo lo concerniente a los delegados, y aunque solo lo viésemos de pasada, comprobamos la relación entre los delegados y los eventos. En este número nos centraremos en los eventos, pero antes comprobaremos que hay ciertas características de los delegados que los hacen imprescindibles para usarlos con los eventos.

Delegados

Cuando definimos un delegado tanto en C# como en Visual Basic usamos la instrucción `delegate`. Esa instrucción en realidad lo que hace es definir un tipo de datos especial. Y la primera impresión es que en realidad esa instrucción utiliza el tipo definido en la propia librería de clases de .NET que tiene el mismo nombre: `System.Delegate`. Pero no es así: en realidad el tipo de datos que obtenemos con una declaración por medio de la palabra clave `delegate` es `System.MulticastDelegate`, que está basada en la clase `Delegate`.

En esta primera parte del artículo nos centraremos en la clase `MulticastDelegate` y en cómo se pueden agrupar varios métodos delegados en un mismo objeto delegado, además de ver cómo podemos invocar a todos esos métodos con una sola llamada.

> **NOTA**
>
> A lo largo de este artículo, usaré `delegate` con la primera letra en minúscula para referirme a la instrucción que tanto C# como Visual Basic utilizan para definir delegados. Pero cuando quiera hacer referencia a la clase `System.Delegate`, usaré la primera letra en mayúscula; así los programadores de C# no se confundirán con los cambios de mayúsculas y minúsculas que tanto les afecta, porque en ese lenguaje, el tamaño (de las letras) sí que importa.

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 Iberoamericana* y autor del libro *Manual Imprescindible de Visual Basic .NET*.
http://www.elguille.info

Delegados multidifusión: unión de varios delegados

Como ya comenté en el artículo anterior, los delegados multidifusión son los que en realidad nos permiten que los eventos tengan el comportamiento que tienen y nos permitan utilizarlos de forma muy flexible. En un momento veremos por qué.

Al usar la instrucción `delegate`, creamos en realidad un objeto de tipo `MulticastDelegate`. Veamos que nos dice la documentación de Visual Studio 2005 sobre esta clase:

Representa un delegado multidifusión; es decir, un delegado que puede tener más de un elemento en su lista de invocación.

Está claro, en este caso no hay dudas sobre el significado. Y debido a que éste es el tipo de delegado que siempre usaremos (o al menos el que usaremos en un gran porcentaje de las ocasiones), en adelante al indicar que usamos un delegado será para referirnos a este tipo de delegado que permite tener más de un elemento asociado, ya que, como acabamos de comprobar, los delegados que podemos crear por medio de la instrucción `delegate` son de este tipo de delegados "compuestos".

Ahora que sabemos qué es un delegado multidifusión, veamos cómo podemos unir varios delegados en uno solo.

Agregar delegados a la lista de un delegado multidifusión

En C# los operadores `+=` y `-=` están sobrecargados para poder utilizarlos con este tipo de delegados, de forma que si queremos añadir más de un dele-

gado a la lista de delegados lo haremos usando precisamente el operador de asignación e incremento (+=). Veamos un ejemplo de cómo agrupar varios delegados y cómo invocarlos, y cuando veamos lo que ocurre, lo tendremos todo más claro.

En el fuente 1 tenemos una definición de un delegado, el cual es una plantilla para un método de tipo void (no devuelve nada) y que recibe como parámetro una cadena. En ese mismo código fuente, tenemos tres métodos que podemos usar con una variable definida mediante ese delegado; cada uno de los métodos muestra la cadena pasada como argumento en la consola, pero de forma diferente, con la idea de que podamos ver que es lo que en realidad está ocurriendo.

```
// La definición del delegado
delegate void MostrarCadenaCallback(string str);

// Varias definiciones del método con la firma adecuada
// para el delegado MostrarCadenaCallback

// Muestra la cadena sin alterar
static void mostrarStr(string str){
    Console.WriteLine(str);
}
// Muestra la cadena en mayúsculas
static void mostrarUpper(string str){
    Console.WriteLine(str.ToUpper());
}
// Muestra la cadena en minúsculas
static void mostrarLower(string str){
    Console.WriteLine(str.ToLower());
}
```

Fuente 1. Definición de un delegado y tres métodos que tienen la misma firma

Para usar ese delegado y asociarlo a cualquiera de los métodos lo haríamos tal como vemos en el fuente 2. En este caso solo usamos uno de los tres métodos que hemos definido, en particular el método mostrarUpper.

```
MostrarCadenaCallback delegado1;
//delegado1 = new MostrarCadenaCallback(mostrarUpper);
delegado1 = mostrarUpper;

delegado1("Hola delegado");
```

Fuente 2. La forma habitual de usar un delegado e invocarlo

Al ejecutar el código del fuente 2, comprobaremos que el mensaje indicado se muestra completamente en mayúsculas, que en realidad es lo que esperamos que ocurra, y tal como vimos en el artículo anterior tenemos dos formas de asignar a la variable delegado1 el método que vamos a usar; la asignación que está comentada es la que tendríamos que usar en las versiones anteriores a Visual C# 2005.

Pero lo que aquí tenemos que "demostrar" es cómo agregar más de un método a una misma variable, y tal como vimos al principio de esta sección, debemos hacerlo mediante el operador +=, por tanto, si usamos el código mostrado en el fuente 3, conseguiremos lo que estamos buscando.

```
MostrarCadenaCallback delegado2;
delegado2 = new MostrarCadenaCallback(mostrarStr);
delegado2 += new MostrarCadenaCallback(mostrarLower);
delegado2 += mostrarUpper;

delegado2("¡Hola Delegado!");
```

Fuente 3. Asociar varios métodos con una misma variable de tipo delegado

Como vemos en dicho código fuente, la forma de asociar varios delegados es la misma que cuando asignamos solo uno, con la única diferencia del operador usado para dicha asignación. Y con el operador de asignación y suma también podemos usar las dos formas de asignar el método al que se llamará desde el delegado.

Lo más interesante de todo esto está en la última línea. Cuando invocamos al delegado pasándole el parámetro que espera (en este caso una cadena), éste se encargará de llamar secuencialmente a todos y cada uno de los métodos delegados que hayamos añadido a la variable que hace referencia al delegado multidifusión.

El orden de llamada a cada uno de los métodos que tenemos en la lista del delegado es el mismo en el que hemos añadido esos métodos; por tanto, el código del fuente 3 producirá la siguiente salida:

```
¡Hola Delegado!
¡hola delegado!
¡HOLA DELEGADO!
```

De la misma forma que podemos añadir nuevos métodos a un delegado, podemos quitar métodos que previamente hayamos añadido. Para ello usaremos el operador -=, el cual se usa exactamente como el que acabamos de ver, pero su función es eliminar delegados de la lista. Si queremos quitar un delegado que no está previamente añadido no se produce ninguna excepción; simplemente la petición se ignora.

Si al código del fuente 3 le agregamos la siguiente línea antes de llamar al delegado, lo que conseguiremos es que solo haya dos métodos en la lista del delegado:

```
delegado2 -= mostrarLower;
```

Nuevamente comprobamos que también podemos usar la forma "abreviada" para quitar métodos de la lista de delegados.

NOTA

En Visual Basic no hay forma de añadir y quitar métodos de la lista de un delegado multidifusión, al menos de la forma como se hace con C#. La única forma de hacerlo es por medio de los métodos compartidos `Combine` y `Remove` de la clase `Delegate` o del delegado que definamos, aunque el valor devuelto siempre es del tipo `Delegate`. Para llamar al delegado podemos usar el método `Invoke` de la clase `Delegate` o bien usar la llamada directa, en cuyo caso el objeto debe ser del tipo del delegado que estamos combinando. En el código de ejemplo que acompaña al artículo hay varios métodos con distintas formas de usar el método `Combine` y `Remove`.

Todo esto es así de complicado si trabajamos directamente con delegados que no están relacionados con eventos, ya que si trabajamos con eventos, podemos usar las instrucciones **AddHandler** y **RemoveHandler** para conseguir la misma funcionalidad de los operadores `+=` y `-=` respectivamente.

Nomenclatura en las definiciones de los delegados

Para finalizar el tema de los delegados, una nota sobre la nomenclatura recomendada.

En la definición de los delegados se recomienda el sufijo `EventHandler` para los nombres de delegados que se utilizan en eventos y el sufijo `Callback` para los nombres de delegados que no sean manejadores de eventos.

En caso de que definamos una clase para usar como parámetro de un delegado relacionado con un evento, debemos añadirle el sufijo `EventArgs` si dicha clase se deriva de `System.EventArgs`.

Una vez hechas estas aclaraciones, pasemos a ver qué son y cómo definir y usar los eventos.

¿Qué es un evento?

Un evento es un mensaje (o notificación) que lanza un objeto de una clase determinada cuando algo ha ocurrido. El ejemplo más clásico y fácil de entender es cuando el usuario pulsa con el ratón en un botón de un formulario. Esa acción produce, entre otros, el evento `Click` del botón en el que se ha pulsado. De esa forma, el botón notifica que esa acción ha ocurrido, y ya es cuestión nuestra que hagamos algo cuando eso ocurra. Si estamos interesados en interceptar ese evento tendremos que comunicárselo a la clase que define el botón; si no lo estamos, simplemente no es necesario que indiquemos nada.

Como es de suponer, los eventos se pueden definir en cualquier clase, y no solo en clases que tengan algún tipo de interacción con el usuario, aunque lo más habitual es que precisamente se usen con clases que forman parte de la interfaz gráfica que se le presenta al usuario de una aplicación. De esa forma podremos saber que algo está ocurriendo y en qué control, de forma que sepamos en todo momento lo que el usuario quiere hacer o lo que está haciendo.

Definir eventos

Como ya vimos en el artículo anterior, la definición de un evento está estrechamente ligada con los delegados, particularmente en el caso de C#, ya que en Visual Basic podemos definir eventos sin que tengamos que asociarlos con delegados, al menos desde el punto de vista del programador; el compilador de Visual Basic se encarga de la relación entre la definición del evento y el delegado que debe tener asociado el evento que definamos.

El hecho de que cada evento esté relacionado con un delegado es porque la forma de lanzar ese evento (o lo que es lo mismo, la forma de notificar que ese evento ha ocurrido) es llamando al delegado con los parámetros que hayamos definido.

Por ejemplo, si queremos tener un evento en una clase para que nos notifique cada vez que se modifica el contenido de una propiedad, tendremos que definir un delegado que indique la "firma" que debe tener el método que vaya a recibir dicha notificación. Además, debemos definir el evento propiamente dicho, el cual será una variable especial del tipo del delegado. Lo de "variable especial" es porque en realidad se define como cualquier otra variable (solo que con un tipo delegado como tipo de datos), pero añadiéndole la instrucción `event`. En el fuente 4 vemos la definición del delegado y el evento.

```
public delegate void NombreCambiadoEventHandler(
                        string nuevo, string anterior);
public event NombreCambiadoEventHandler NombreCambiado;
```

Fuente 4. Definición de un evento y el delegado asociado

En este ejemplo, definimos un delegado que recibe dos parámetros; el primero es el nuevo valor que hemos asignado a la propiedad y el segundo el que tenía antes de asignar ese nuevo valor.

A continuación definimos el evento, que es del tipo del delegado. Como vemos, en realidad la definición del evento no se diferencia mucho de cómo definiríamos una variable que quisiéramos tener relacionada con un evento, salvo porque utilizamos la instrucción event para definirla. Esa instrucción hace que exista "automáticamente" una relación entre el delegado y el evento, de forma que no tengamos que instanciar expresamente el delegado para poder usarlo.

Usar los eventos de una clase

Una vez que tenemos definido el evento en una clase, podemos recibir notificaciones cada vez que ese evento se produzca. Por supuesto, la clase que define el evento es la que se encargará de producir el evento para informar a los objetos que deseen recibir dicha notificación.

Para recibir el mensaje del evento, debemos crear un método que tenga la misma firma que el delegado que hemos asociado a dicho evento. Usando el evento definido en el fuente 4, la asociación deberíamos hacerla tal como vemos en el código del fuente 5.

```
Cliente cli = new Cliente();
cli.NombreCambiado += new
      Cliente.NombreCambiadoEventHandler(
            cli_NombreCambiado);
```

Fuente 5. Asociación de un evento con un método

El método indicado en el constructor del delegado debemos definirlo con la firma determinada por el propio delegado, quedando la definición tal como vemos en el fuente 6.

```
void cli_NombreCambiado(string nuevo, string anterior)
{
    label2.Text = "Se ha cambiado el nombre:\n" +
                  "Nuevo valor:    " + nuevo + "\n" +
                  "Valor anterior: " + anterior;
}
```

Fuente 6. Definición del método que recibirá la notificación cuando se produzca el evento.

Como podemos apreciar, la forma de asociar el evento con el método que recibirá la notificación cada vez que éste se produzca es usando el operador **+=**, que como vimos es la forma que tienen los delegados multidifusión de añadir los métodos que recibirán el aviso cada vez que dicho delegado sea llamado; en el caso de los eventos, ese aviso se iniciará cuando lancemos el evento desde la clase que lo define (en un momento veremos cómo).

Una pega que podemos encontrarnos es con la definición del método que recibe el evento. En realidad no es un problema, ya que si sabemos que definición tiene el delegado sabremos cuál debe ser la definición de dicho gestor de evento. Pero para que nos resulte más cómodo, el editor de Visual C# 2005 nos facilita dicha creación; incluso nos permite saber cuál es el delegado asociado con el evento que queremos interceptar.

En las figuras 1 y 2 vemos cómo el IDE de Visual Studio 2005 nos permite tanto usar el delegado correcto (figura 1) como la creación del método con la firma adecuada (figura 2).

De esta forma, no tendremos que buscar la definición del delegado para crear de forma fácil el método que debemos usar.

Figura 1. Creación automática del delegado adecuado desde el editor de Visual C# 2005

Figura 2. Creación automática de un método con la firma del delegado asociado con el evento

> **NOTA**
>
> Debido a que los eventos, de forma predeterminada, son miembros de instancia (están asociados con cada instancia de la clase que los define), si creamos un nuevo objeto de esa clase debemos volver a asociar el método que recibirá la notificación del evento.
>
> Esto no es necesario si esa clase solo la instanciamos una vez y usamos siempre la misma instancia.

Producir un evento

Una vez que tenemos la definición del delegado y el evento en nuestra clase, tenemos la posibilidad de lanzar o producir dicho evento. Cuándo y dónde lanzar el evento depende de nuestro código. En el ejemplo de la clase **Cliente** que produce un evento cada vez que se modifica la propiedad **Nombre**, lo haremos de la forma que mostramos en el fuente 7.

```
private string m_Nombre;
public string Nombre
{
    get { return m_Nombre; }
    set{
        if( NombreCambiado != null )
        {
            NombreCambiado(value, m_Nombre);
        }
        m_Nombre = value;
    }
}
```

Fuente 7. La forma de producir un evento en C#

En el bloque **set** es donde lanzamos el evento. En este ejemplo lo lanzamos se modifique o no el contenido de la propiedad, pero ese detalle no es lo

importante; en lo que de verdad debemos fijarnos es en la comprobación de si el evento no es nulo, ya que debemos hacer esa comprobación, con idea de lanzar el evento solo si alguien lo está interceptando; en caso contrario no debemos llamar al delegado multidifusión, que es en realidad el que se encarga de "propagar" el mensaje a todos los métodos que hayamos definido para recibir el mensaje.

Si alguien está esperando la notificación del evento, usamos el mismo nombre del evento (que en realidad es un `MulticastDelegate`) para lanzar el evento, indicando los parámetros adecuados (en nuestro ejemplo, el nuevo valor de la propiedad y el valor que tenía antes). Y una vez que hemos producido el evento es que asignamos el nuevo valor al campo que mantiene el contenido de esa propiedad.

> **NOTA**
>
> Veremos cómo definir, interceptar y lanzar los eventos en Visual Basic en un próximo artículo dedicado exclusivamente a los eventos desde el punto de vista de ese lenguaje, en el que también veremos cómo usar la nueva instrucción `Custom Event`, exclusiva de Visual Basic 2005.

Interceptar el mismo evento más de una vez

Como podemos comprobar en el código mostrado en las secciones anteriores, la asociación entre un evento y el método que recibirá la notificación de que dicho evento produce lo hacemos mediante el operador `+=`. Si esa asociación la hacemos con varios métodos, todos y cada uno de esos métodos recibirán la notificación de que el evento se ha producido. Esto es posible porque en el fondo la instrucción `event` en realidad está definiendo un objeto del tipo `MulticastDelegate`, y como vimos al principio de este artículo por medio de ese tipo de delegado podemos "asociar" varios métodos con un mismo delegado.

Un evento es un mensaje (o notificación) que lanza un objeto de una clase determinada cuando algo ha ocurrido

Por tanto, todos los métodos que agreguemos por medio del operador `+=` estarán listos para recibir el mismo mensaje que lancemos (ver fuente 7). Y tal como vimos en el código del fuente 3, el orden en el que se notificarán será el mismo en el que hayamos añadido esos métodos a la lista de métodos que quieren recibir el evento.

Esa asociación la podemos realizar tanto usando un nuevo método como usando un método ya existente. En este último caso, ese método se ejecutará tantas veces como veces esté en la lista de delegados que recibirán la notificación, algo que seguramente no será de mucha utilidad, pero que es posible. Y debido a que podemos hacerlo, debemos tenerlo en cuenta, para que no ocurra esa múltiple notificación. Por ejemplo, si la clase que produce eventos solo la instanciamos una vez, y utilizamos el siguiente código para "ligar" el método con el evento:

```
cli.NombreCambiado += new
  Cliente.NombreCambiadoEventHandler(
                    cli_NombreCambiado);
```

Cada vez que ejecutemos esa línea se añadirá el método `cli_NombreCambiado` a la lista de métodos que recibirán la notificación, y cuando el evento se produzca, se llamará esa misma cantidad de veces, consiguiendo algo que seguramente no era lo que queríamos.

Un mismo método que recibe eventos de clases diferentes

En el mismo ejemplo que estamos usando, podemos asociar un mismo método para que reciba notificaciones de objetos diferentes, ya que si la firma del método coincide con la del delegado del evento, ese método lo podremos usar sin ningún tipo de problema. Esto es aplicable no solo al objeto "cli" que hemos definido en el ejemplo, sino a cualquier otra instancia que creemos del tipo `Cliente` (o cualquier otra clase que queramos usar, siempre que el método tenga la misma firma que el delegado correspondiente); de esa forma podremos ahorrarnos algo de código.

Debido a que el ejemplo de la clase Cliente puede que no nos aclare mucho, veamos esto último usando controles y eventos definidos en los controles de `Windows.Forms`. Por ejemplo, si tenemos varios controles de tipo TextBox en un formulario y queremos seleccionar todo el texto que tenga ese control al recibir el foco (cuando el control es el control activo), podemos hacer una múltiple asociación con un mismo método, tal como vemos en el fuente 8.

Ni qué decir tiene que todos los eventos (como es este caso), deben tener la misma firma, lo cual no implica que deban ser del mismo tipo, ya que lo único que en realidad importa es que el delegado asociado con el evento tenga la misma definición que el método que recibirá la notificación del evento. En el caso de los contro-

```
this.textBox1.Enter += new System.EventHandler(this.textBox_Enter);
this.textBox2.Enter += new System.EventHandler(this.textBox_Enter);
this.textBox3.Enter += new System.EventHandler(this.textBox_Enter);
```

Fuente 8. Asociación de eventos de clases diferentes con un mismo método

les de `Windows.Forms`, ese mismo método lo podríamos asociar a un botón:

```
this.button1.Enter +=new
      EventHandler(this.textBox_Enter);
```

Lo que sí debemos tener en cuenta es que el código que usemos en ese método esté preparado para que los controles puedan ser de tipos diferentes, tal como vemos en el código del fuente 9.

```
private void textBox_Enter(object sender, EventArgs e)
{
   labelInfo.Text = "El foco lo tiene "+
                    ((Control)sender).Name;
   if( sender is TextBox )
   {
       ((TextBox)sender).SelectAll();
   }
}
```

Fuente 9. Si un mismo método se producirá desde clases diferentes debemos tenerlo en cuenta en el código.

Añadir "manejadores" de evento

Cuando estamos trabajando con formularios y controles, la asociación entre un evento y el método que usaremos para interceptarlo lo podemos hacer de varias formas. Una de ellas es de forma totalmente manual, que es la que hemos visto anteriormente; pero para ser sinceros, esa no será la forma habitual de asociar un evento con un método, ya que el diseñador de formularios de Visual Studio 2005 nos ofrece una forma más sencilla.

Para "ligar" un evento con un método (y crearlo si no existe), debemos seleccionar el control que define el evento que queremos usar y en la ventana de propiedades seleccionar "Eventos" (el botón con la imagen de un rayo amarillo). De esa forma tendremos una lista de los eventos definidos por ese control (o al menos de los eventos que el creador del control haya decidido que se muestren en esa ventana de propiedades). En la figura 3 podemos ver algunos de los eventos de un formulario.

Como vemos en la figura 3, si el evento está asociado a un método, se muestra el nombre del mismo; si no hay ninguna asociación, el campo se muestra en blanco. Para crear un nuevo método y asociarlo al evento, simplemente tendremos que hacer una doble pulsación en la caja de textos en blanco. Pero si lo que queremos es usar uno de los métodos existentes, podemos seleccionar el método de la lista desplegable; en esa lista solo se mostrarán los métodos que tengan la misma firma del delegado asociado con ese evento, tal como podemos ver en la figura 4, donde el delegado del evento `Click` tiene la misma firma que los mostrados en dicha lista, lo que nos permite usar cualquiera de los existentes o bien crear uno nuevo.

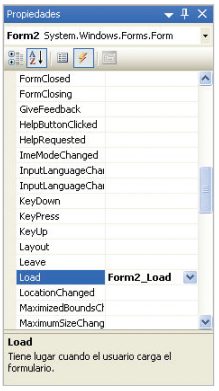

Figura 3. Desde la ventana de propiedades podemos seleccionar el evento que queremos interceptar

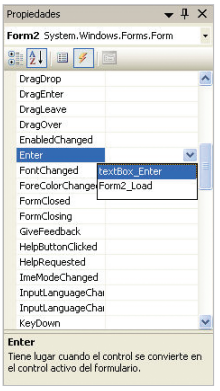

Figura 4. Podemos asociar un evento a un método existente, el cual seleccionamos de una lista desplegable

Lo único que no nos permite el diseñador de formularios es asociar varios métodos de eventos a un mismo evento. En ese caso, tendremos que hacer manualmente esa asociación. La pregunta puede ser ¿dónde hacer esa asociación? La respuesta es: donde queramos, pero siempre que ese código no se repita más de una vez, con idea de no agregar más veces de las necesarias el método que intercepta el evento. Por tanto, lo más recomendable es que escribamos ese código en el constructor del formulario, pero justo después de la llamada al método `InitializeComponent`. Esto también es válido para cuando decidamos asociar manualmente los eventos con los métodos que recibirán la notificación.

Conclusiones

En este artículo hemos visto cómo trabajar con los eventos en C#: desde cómo definirlos hasta cómo interceptarlos y cómo están relacionados los eventos con los delegados, principalmente con los delegados multidifusión, gracias a los cuales podemos asociar un mismo evento con varios métodos. Si a lo comentado en este artículo le añadimos todo lo dicho en el artículo anterior, dedicado casi exclusivamente a los delegados, tenemos todo lo necesario para entender cómo trabajan los eventos y los delegados. Pero no todo está dicho: aún hay ciertas cosas que necesitamos saber para sacarle todo el rendimiento a los eventos y delegados, principalmente a los que definamos en nuestras propias clases. Pero eso lo dejaremos para otro artículo en el que también trataremos estos conceptos desde el punto de vista del programador de Visual Basic.

Como siempre, el código de los ejemplos usados en el artículo está disponible (tanto para C# como para Visual Basic) desde el sitio Web de **dotNetManía**.