Guillermo "Guille" Som

Delegados y eventos en Visual Basic 2005

Llega Custom Event... ¡y yo con estos pelos!

En esta ocasión vamos a centrarnos en los delegados y eventos desde el punto de vista del programador de Visual Basic, y particularmente en la nueva instrucción Custom Event, desde la que tendremos control total sobre cómo se crean, destruyen y utilizan los eventos desde Visual Basic 2005. También veremos cómo definir e interceptar los eventos de las distintas formas que nos permite ese lenguaje.

» Eventos en Visual Basic 2005

Desde sus inicios, Visual Basic ha sido un lenguaje enfocado a aliviar al programador de las tareas de bajo nivel, consiguiendo de esa forma que el programador se centre en lo realmente importante y se olvide un poco de todo lo que ocurre de fondo. Los eventos no son una excepción, y por eso los programadores de Visual Basic a la hora de definir un evento solo tienen que preocuparse de una cosa: definirlo. Aunque esto estaba bien en las versiones anteriores a .NET, ya que .NET Framework necesita más, y no le basta con una simple definición de un evento. Porque, tal como vimos en los dos artículos anteriores (**dotNetManía** nº 30 y 31), el motor de tiempo de ejecución de .NET (CLR) necesita que cada evento que hayamos definido tenga un delegado asociado, y por suerte el compilador de Visual Basic se encarga de esa exigencia y crea ese delegado por nosotros.

Cuando definimos un evento en Visual Basic, solo nos tendremos que preocupar del nombre que tendrá dicho evento y de los parámetros (si tiene) que usará. Esos parámetros –la cantidad y tipos– son importantes, ya que el método usado para interceptar el evento debe tener el mismo número y tipos de parámetros que hayamos usado a la hora de definir el evento. Por ejemplo, si tenemos la definición de evento que se muestra en el código del fuente 1, la definición del método que lo interceptará (el que recibirá la notificación cuando dicho evento se produzca) también ha de tener esos dos parámetros de tipo `String`, tal como vemos en el código del fuente 2.

```
Event DatosCambiados( ByVal nuevo As String, _
                      ByVal anterior As String)
```

Fuente 1. Definición de un evento con dos parámetros

```
Sub cli_DatosCambiados( ByVal nuevo As String, _
                        ByVal anterior As String)
```

Fuente 2. Método que intercepta el evento definido en el fuente 1

Cuando se lanza un evento, en realidad se está llamando a los métodos que lo interceptan, y lo habitual es que esos métodos estén en clases diferentes a la que define el evento. Para .NET los métodos "receptores" de los eventos son funciones que no devuelven ningún valor, y cuando el evento se produce, el CLR llama (o ejecuta) dichos métodos. Pero el CLR tiene que asegurarse de que el método receptor del evento cumple con la condición de que tenga la misma firma que el evento, es decir, tenga el mismo número y tipo de parámetros. Y la forma de asegurarlo es por medio de los delegados, que como sabemos, en el fondo son "punteros administrados" a funciones. Pero los programadores de Visual Basic no tienen porqué saber nada de punteros, ya sean administrados o no. Por eso es el propio compilador de Visual Basic el que se encarga de esos "detalles". Y sin que nosotros lo sepamos, cuando definimos un evento, en realidad el compilador define también un delegado, el cual asocia con dicho evento.

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 de los libros "Manual Imprescindible de Visual Basic .NET" y "Visual Basic 2005".
http://www.elguille.info

Cuando definimos un evento en Visual Basic, solo tenemos que preocuparnos del nombre que tendrá dicho evento y de los parámetros (si tiene) que usará

Los programadores de Visual Basic no tienen por qué saber nada de esa relación evento-delegado, o al menos así era hasta la llegada de Visual Basic 2005, y en realidad ni tan siquiera en esa versión tienen que saber nada de dicha relación, salvo porque ahora el lenguaje incluye una nueva instrucción que permite controlar al dedillo todo lo relacionado con los eventos: `Custom Event`. Con esa instrucción podemos controlar todo lo que atañe al uso de los eventos: cuándo se asocia un evento con un método, cuándo se quita esa relación y cuándo se produce el evento. Pero antes de entrar en detalles, veamos qué es lo que tenemos que hacer para definir, lanzar e interceptar eventos desde Visual Basic. Empezaremos viendo cómo hacer todo eso de la forma tradicional y sencilla, para después ver cómo definir eventos al estilo .NET, es decir, relacionando los eventos con delegados.

Definir eventos en Visual Basic

La definición de un evento en Visual Basic la haremos por medio de la instrucción `Event`. Los eventos, al igual que el resto de miembros de una clase o tipo, deben tener un nombre y opcionalmente parámetros. Usando el código del fuente 1 tendremos definido un evento que nos puede servir para indicar que el valor de una propiedad ha cambiado; ese evento informará al método que lo intercepte de cuál es el nuevo valor que se ha asignado a la propiedad, así como cuál era el valor que tenía antes de esa asignación.

Lanzar (o producir) un evento

Para producir un evento, lo haremos por medio de la instrucción `RaiseEvent`. A dicha instrucción le tenemos que indicar el nombre del evento que queremos producir y los valores a usar en los parámetros que tenga el evento. Cuando trabajamos con el entorno de desarrollo de Visual Basic, al escribir esa instrucción, éste nos mostrará los eventos que tenemos definidos, y al seleccionarlo veremos qué parámetros tenemos que indicar, tal como muestra la figura 1.

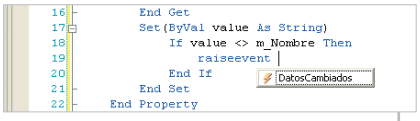

Figura 1. *IntelliSense* nos muestra los eventos que tenemos definidos en la clase

En el código del fuente 3 se muestra cómo definir y lanzar un evento. Tal como vemos, ese evento sirve para notificar el cambio en la propiedad `Nombre` de la clase `Cliente`.

```
Public Class Cliente
  Public Event DatosCambiados(ByVal nuevo As String, _
                              ByVal anterior As String)

  Private m_Nombre As String
  Public Property Nombre() As String
    Get
        Return m_Nombre
    End Get
    Set(ByVal value As String)
        If value <> m_Nombre Then
            RaiseEvent DatosCambiados(value, m_Nombre)
        End If
        m_Nombre = value
    End Set
  End Property
End Class
```

Fuente 3. Ejemplo de una clase que define y lanza un evento

Interceptar eventos

Para recibir la notificación de que un evento se ha producido, debemos definir un método que tenga la misma firma que el evento, es decir, que tenga el mismo número y tipo de parámetros. En nuestro ejemplo, podría ser como el mostrado en el código del fuente 2.

Cuando el evento se produce en la clase que lo define (por medio de `RaiseEvent`), cada método que esté asociado con ese evento recibirá la notificación de que el evento se ha producido, pero para poder

recibir ese aviso, antes debemos asociar dicho método con el evento que queremos interceptar.

Asociar un método con un evento

Para que un evento pueda avisar a todos los métodos que esperan el aviso, hay que asociar ese evento con cada método que recibirá la notificación. En Visual Basic podemos hacerlo de dos formas:

Asociar un método con evento automáticamente

La forma más simple es declarando la variable de la clase que define el evento con `WithEvents`. De esa forma, podemos definir el método de la misma forma que lo hacemos con los controles de los formularios, es decir, usando la instrucción `Handles`, después de la definición del método, seguida de la variable y el nombre del evento a interceptar. En el código del fuente 4 podemos ver cómo definir una variable de tipo `Cliente` y el método que recibirá la notificación.

```
Private WithEvents cli As Cliente

Sub Main()
    cli = New Cliente
    cli.Nombre = "Pepe"
End Sub

Private Sub cli_DatosCambiados( ByVal nuevo As String, ByVal anterior As String) _
                              Handles cli.DatosCambiados

   Console.WriteLine("Han cambiado los datos:{2}" & "Anterior: {0}{2}Nuevo: {1}", _
                     anterior, nuevo, vbCrLf)
End Sub
```

Fuente 4. Las variables definidas con `WithEvents` permiten asociar un método a un evento por medio de `Handles`

La ventaja de declarar la variable con `WithEvents` es doble: por un lado nos permite usar la instrucción `Handles` para asociar fácilmente un método con el evento, y por otro nos permite agregar dicho método de forma automática, al menos si trabajamos con el editor de Visual Basic que incorporan todas las versiones de Visual Studio 2005 (incluida la versión Express). Para añadir el método solo tendremos que mostrar la ventana de código y seleccionar la variable definida con `WithEvents` de la lista despegable de la izquierda y el evento a interceptar de la lista despegable de la derecha (ver figura 2). De esta forma, el editor de Visual Basic se encargará de crear el método con los parámetros adecuados y asociarlo al evento por medio de la instrucción `Handles`.

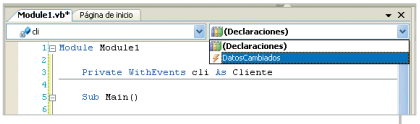

Figura 2. El editor de Visual Basic permite crear automáticamente los métodos asociados a los eventos

Debemos aclarar que el uso de la instrucción `WithEvents` solo se permite a nivel de módulo; es decir, no podemos usar esa instrucción para declarar una variable dentro de un método o propiedad (no puede ser local), y tampoco podemos usarla para definir arrays. Aunque sí podemos usar cualquier modificador de ámbito como `Public`, etc.

Asociar un método con evento de forma manual

La otra forma de asociar un evento con un método es por medio de la instrucción `AddHandler`. En este caso, la declaración de la variable la podemos hacer en cualquier parte del código, incluso de forma local (dentro de un método), y usando cualquiera de los modificadores permitidos, además de que esa variable puede ser un elemento de un array. Una vez que tenemos definida e instanciada la variable, usaremos la instrucción `AddHandler` tal como se muestra en el siguiente código:

```
AddHandler cli.DatosCambiados, _
      AddressOf cli_DatosCambiados
```

En el primero de los dos argumentos indicamos la variable y el evento que queremos interceptar; en el segundo tenemos que indicar la dirección de memoria del método que interceptará ese evento. Tal como vemos en el código anterior, la dirección de memoria de un método se indica en Visual Basic por medio del operador `AddressOf`.

Una de las desventajas de usar `AddHandler` es que la definición del método que interceptará el evento lo tenemos que definir de forma manual. Aquí tengo que romper una lanza a favor del editor de Visual C# 2005, ya que, como vimos en el artículo pasado (**dotNetManía** nº 31), dicho editor permite la creación automática del método asociado a un evento. Aunque (consuelo que nos queda), debido a la precompilación que hace el editor de Visual Basic, si la definición del método no coincide con la del evento, nos avisará de ese hecho; en la figura 3 podemos ver ese aviso.

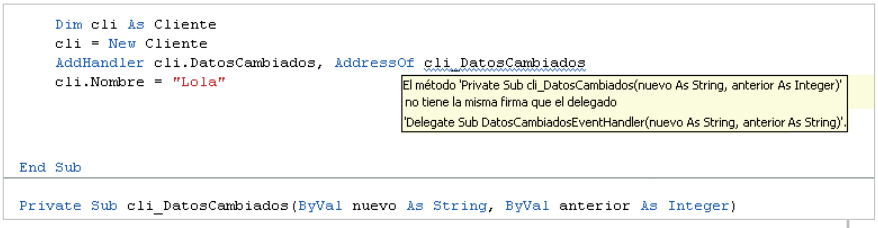

Figura 3. Aviso de que el método no coincide con el evento

La otra desventaja (con respecto a `WithEvents`) es que si la variable que hemos usado con `AddHandler` la volvemos a instanciar, obligatoriamente tendremos que volver a asociar el evento con el método, algo que es lógico, ya que si eliminamos el objeto de la memoria no existe el método al que queremos acceder; esto es algo que no ocurre si la variable está declarada con `WithEvents`, ya que siempre que se vuelve a crear el objeto se asocian los métodos que hayan usado la instrucción `Handles`. En cualquier caso, al destruir el objeto (o al asignarle una nueva instancia) se desasocian los métodos con los eventos.

Quitar la asociación de un método con un evento

Si necesitamos quitar una asociación de un método con un evento (independientemente de que esa asociación se haga automáticamente con `WithEvents`/`Handles`), podemos usar la instrucción `RemoveHandler`, a la que le indicaremos los mismos parámetros que a `AddHandler`, es decir, el evento y el método:

```
RemoveHandler cli.DatosCambiados,
    AddressOf cli_DatosCambiados
```

Si ese método no estaba asociado con ese evento no se produce ningún error, ya que simplemente se ignora. Cuando veamos el código de la instrucción `Custom Event` comprenderemos mejor cómo funciona todo esto, y porqué no se produce un error.

> **NOTA**
>
> Como ya comenté en el primer artículo (**dotNetManía** nº 30), las instrucciones de Visual Basic `AddHandler` y `RemoveHandler` son equivalentes a las sobrecargas `+=` y `-=` (respectivamente) de los eventos de C#.

Definir eventos usando delegados

Como hemos dicho antes, aunque .NET siempre necesita que haya un delegado relacionado con cada evento que definamos, en Visual Basic no es obligatorio, al menos para nosotros, ya que el compilador de Visual Basic siempre define un delegado y lo asocia al evento que definamos. Pero si nosotros queremos quitarle ese trabajo al compilador, podemos definir los delegados por nuestra cuenta y asociarlos al evento.

¿Qué necesidad tenemos de hacer algo que el compilador de Visual Basic hará automáticamente por nosotros?

¡Buena pregunta! Tenemos un par de razones para hacerlo: la primera es porque si vamos a definir varios eventos que tengan la misma firma, es decir, que definan los mismos parámetros, el compilador creará un delegado por cada uno de esos eventos, independientemente de que ya exista un delegado que tenga esa misma definición de parámetros o que haya otros eventos con parámetros idénticos. Para comprobarlo, puedes añadir a la clase `Cliente` otro evento, en esta ocasión para que notifique que los apellidos han cambiado (no hace falta que añadas más código):

```
Public Event ApellidosCambiados(
                    ByVal nuevo As String,
                    ByVal anterior As String)
```

Compila el proyecto, y si examinas el ejecutable con la utilidad MSIL Disassembler, verás que hay una definición de un delegado para cada uno de los eventos, y que la nomenclatura usada es la recomendada: `<nombre del evento>EventHandler`.

Si esa es nuestra intención, tener varios eventos que tengan la misma cantidad y tipos de parámetros, podemos definir un delegado y asociar ese delegado con cada uno de esos eventos. El delegado lo podemos definir tal y como puede ver en el fuente 5.

Y los eventos simplemente los definimos como del tipo de ese delegado (que es como ya explicamos que se hace con C#) (fuente 6).

```
Public Delegate Sub DatosCambiadosEventHandler(
 ByVal nuevo As String, ByVal anterior As String)
```

Fuente 5. Definición del delegado que usaremos en los eventos

```
Public Event NombreCambiado As DatosCambiadosEventHandler
Public Event ApellidosCambiados As DatosCambiadosEventHandler
```

Fuente 6. Declaración de dos eventos que usan el delegado del fuente 5

A la hora de producir cualquiera de esos eventos, lo haremos como hemos visto antes, es decir, usando `RaiseEvent`. Incluso IntelliSense nos mostrará los parámetros que tenemos que usar, que serán los que hayamos definido en el delegado, tal como podemos ver en la figura 4.

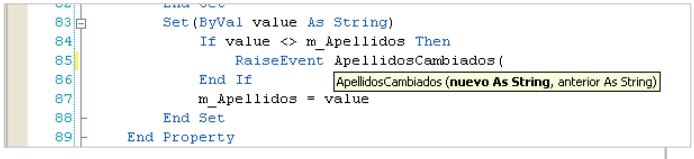

Figura 4. `RaiseEvent` se usa igual, independientemente de cómo hayamos definido el evento

Como vemos, la forma de definir el evento no influye en la forma de producirlos e incluso de interceptarlos, ya que en realidad así es como se definen siempre.

La segunda razón para definir eventos por medio de delegados es para poder usar la instrucción `Custom Event`, ya que, como veremos en un momento, la única forma de usar esa instrucción es por medio de delegados.

Incluso puede haber una tercera razón, que sería para "compatibilizar" la definición de los eventos con la única forma que tienen a su disposición los programadores de C#. De esa forma, si en nuestro proyecto trabajan programadores de los dos lenguajes, les resultará más fácil saber qué es lo que estamos haciendo.

> **NOTA**
>
> Cuando definimos un delegado que utiliza la nomenclatura recomendada de añadir `EventHandler` al nombre que demos al evento, y resulta que ya tenemos un evento declarado con la instrucción `Event` que coincide con ese nombre (pero sin el "apéndice" `EventHandler`), recibiremos un error indicando que hay conflicto de duplicidad de nombres, ya que el delegado definido implícitamente por el compilador de Visual Basic tiene el mismo nombre. La única solución en este caso es cambiar el nombre del evento definido directamente o el del delegado.

Definir eventos con Custom Event

La instrucción `Custom Event`, al igual que su hermana menor `Event`, sirve para definir eventos, pero con la particularidad de que nos permite saber con todo lujo de detalles que está ocurriendo con los eventos: cuándo se asocia un método con el evento, cuándo se quita esa asociación y cuándo se utiliza la instrucción `RaiseEvent` para producirlo.

Debido a todos estos detalles o debido a este control que podemos tener sobre los eventos, necesitamos usar delegados, ya que la instrucción `Custom Event` en realidad se define usando tres bloques de instrucciones, y dos de esos bloques reciben precisamente un delegado como parámetro, además de que la propia instrucción debemos definirla como un tipo de delegado. Para comprenderlo mejor, en el código del fuente 7 tenemos la declaración equivalente al evento `NombreCambiado` del fuente 6, que también utiliza el delegado `DatosCambiadosEventHandler` que vimos en el código del fuente 5.

> La instrucción Custom Event sirve para definir eventos, pero con la particularidad de que nos permite saber con todo lujo de detalles qué está ocurriendo con los eventos

Como vemos en el código del fuente 7, los eventos creados con `Custom Event` en realidad están formados por tres bloques de código, uno para cada una de las tres acciones que podemos realizar: asociar un método con un evento (`AddHandler`), quitar esa asociación (`RemoveHandler`) y producir el evento (`RaiseEvent`).

En los dos primeros, el parámetro que reciben es un objeto del tipo de delegado asociado con este evento (que es el indicado después de la instrucción `Custom Event`), mientras que en el tercero, los parámetros son los que recibe el método que interceptará dicho evento, y, como es de esperar, son los mismos que define el delegado.

```
Public Delegate Sub DatosCambiadosEventHandler( ByVal nuevo As String,
                                                ByVal anterior As String)

Private delegadosNombre As New List(Of DatosCambiadosEventHandler)

Public Custom Event NombreCambiado As DatosCambiadosEventHandler
    AddHandler(ByVal value As DatosCambiadosEventHandler)
        ' Añadir el delegado a la colección de delegados
        delegadosNombre.Add(value)
    End AddHandler

    RemoveHandler(ByVal value As DatosCambiadosEventHandler)
        ' Se quita un manejador de eventos
        delegadosNombre.Remove(value)
    End RemoveHandler

    RaiseEvent(ByVal nuevo As String, ByVal anterior As String)
        ' Producir el evento en cada uno de los delegados de la colección
        For Each de As DatosCambiadosEventHandler In delegadosNombre
            de.Invoke(nuevo, anterior)
        Next
    End RaiseEvent
End Event
```

Fuente 7. Declaración de un evento con `Custom Event`

En ese mismo código podemos observar que usamos una colección para almacenar todos los delegados que apuntan a los métodos que quieren ser notificados cuando el evento se produzca. Aquí utilizo una colección genérica de tipo `List(Of DatosCambiadosEventHandler)`, pero podríamos usar cualquier otro tipo de colección o forma de almacenar esos delegados, ya que lo que realmente interesa es saber cuáles son, con idea de poder llamarlos (invocarlos) desde el bloque `RaiseEvent`, desde el cual usamos el delegado de la forma habitual, que es utilizando el nombre de la variable del delegado y pasándole los argumentos por medio de `Invoke`, método que es totalmente opcional.

Lo que hacemos con `Custom Event` es en realidad lo que hace el propio .NET con los eventos cada vez que se encuentra con la instrucción `AddHandler` (o automáticamente en el caso de un método que incluye la instrucción `Handles`): guardar la dirección de memoria de ese método en una colección, y cuando el evento se produce, llamar a cada uno de esos métodos para avisarle de que el evento se ha producido; por supuesto, esas llamadas se hacen por medio de los delegados.

Si queremos definir más de un evento con la instrucción `Custom Event` debemos crear una colección independiente para cada uno de los eventos, con idea de que cada uno de ellos tenga su propia lista de métodos a los que llamar (por medio de los delegados) desde el bloque `RaiseEvent`. Ni que decir tiene, que si simplemente vamos a hacer lo que vemos en el código del fuente 7 no hace falta que usemos `Custom Event`, salvo que queramos monitorizar esas acciones, y estar seguros que todo va como debe ir. Aunque en esos bloques de código, particularmente en el bloque `AddHandler`, podemos hacer ciertas comprobaciones, de forma que si algunas de ellas no nos satisface, podamos ignorar la petición de interceptar el evento.

Esas comprobaciones pueden ser variadas, y en la mayoría de los casos, la información la obtendremos desde el objeto recibido como parámetro. En el código del fuente 8 vemos cómo acceder a la información sobre dónde se está agregando el manejador de eventos, el nombre del ejecutable, el nombre de la clase y el nombre del método.

```
Public Custom Event NombreCambiado As DatosCambiadosEventHandler
    AddHandler(ByVal value As DatosCambiadosEventHandler)
        ' Se añade el manejador de eventos
        Console.WriteLine("Se añade un manejador de eventos para NombreCambiado")
        '
        ' El nombre del ejecutable
        ' Si se cambia el nombre una vez compilado,
        ' se muestra el nuevo nombre.
        Console.WriteLine("  El ejecutable es: {0}", value.Method.Module.Name)
        ' Target solo tendrá algo si la clase es un objeto de instancia
        If value.Target IsNot Nothing Then
            Console.WriteLine("  La clase es: {0}", value.Target.ToString)
        Else
            Console.WriteLine("  Seguramente se está usando una clase compartida.")
        End If
        ' El nombre del método
        Console.WriteLine("  Nombre del método: {0}", value.Method.Name)

        ' Añadir el delegado a la colección de delegados
        delegadosNombre.Add(value)
    End AddHandler

    RemoveHandler(ByVal value As DatosCambiadosEventHandler)
        ' Se quita un manejador de eventos
        Console.WriteLine("Se quita un manejador de eventos para NombreCambiado")
        delegadosNombre.Remove(value)
    End RemoveHandler

    RaiseEvent(ByVal nuevo As String, ByVal anterior As String)
        ' Producir el evento en cada uno de los delegados de la colección
        For Each de As DatosCambiadosEventHandler In delegadosNombre
            Console.WriteLine("Se produce el evento NombreCambiado")
            Console.WriteLine("    Nuevo: {0}, anterior: {1}", nuevo, anterior)
            'de(nuevo, anterior)
            ' También se puede usar el método Invoke
            de.Invoke(nuevo, anterior)
        Next
    End RaiseEvent
End Event
```

Fuente 8. Desde los bloques de código de Custom Event podemos acceder a cierta información sobre el cliente que está usando el evento

Hay que aclarar que si el método que recibirá la notificación del evento está declarado en un módulo (`Module`) o en una clase estática de C#, no podremos obtener la información del nombre de la clase, ya que la clase debe ser de instancia no compartida; por eso la comprobación de que `Target` no sea un valor nulo.

Conclusiones

Confío que con lo explicado en este artículo el lector tenga una visión más clara sobre cómo usar los eventos en Visual Basic 2005, y si a este artículo le añadimos lo ya explicado en los dos anteriores (aunque el código de Visual Basic tuviera que verlo desde los ejemplos incluidos en los ZIP), seguro que los delegados tampoco serán una incógnita, al menos en lo relativo a la relación que tienen con los eventos.

En un próximo artículo seguiremos viendo más cosas relacionadas con los eventos y los delegados, particularmente en la forma de crear nuestras propias clases basadas en el tipo `EventArgs` y en cómo comunicarnos desde la aplicación que recibe el evento con la clase que lo produce, para, por ejemplo, cancelar una acción o devolver algunos parámetros que dicha clase necesite.

Como de costumbre, el código completo de los ejemplos usados en este artículo está disponible desde la Web de **dotNetManía**, aunque en esta ocasión ese código en su mayoría es para Visual Basic; en C# solo está el ejemplo que usa la clase `Cliente` desde una clase estática y otra de instancia.