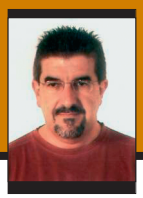

Guillermo "Guille" Som

El enrutador que los enrute... Eventos en XAML

Con la llegada de XAML y las aplicaciones para Windows Presentation Foundation (WPF) han cambiado ciertos conceptos en la programación para el entorno .NET, y los eventos no son una excepción a este cambio. De hecho, ahora debemos cambiar un poco el chip para dejar paso a los eventos enrutados (routed events), y en este artículo veremos qué son, cómo funcionan y lo más importante: cómo utilizarlos.

Hospital DotNet

Guillermo Som ha cedido los derechos de autor de este artículo al proyecto Hospital DotNet. Este proyecto es una iniciativa del portal DOTNETSOLIDARIO que pretende recaudar fondos para la creación de un hospital en una de las zonas más desfavorecidas del tercer mundo.

Usted puede colaborar con este proyecto. Visite la Web en *www.dotnetsolidario.com* para conocer los detalles.

» *Empecemos viendo* la descripción de eventos enrutados (*routed events*). Según la documentación del SDK de Windows, un evento enrutado es una instancia de un evento que se propaga por un árbol de elementos relacionados en lugar de apuntar a un solo elemento. Según esa misma documentación, hay tres tipos de enrutamiento: *bubbling* (de burbuja), *tunneling* (de túnel) y *directo*. Con el enrutamiento *bubbling*, el evento se propaga ("sube") desde el elemento que lo produce hasta la parte superior del árbol de elementos. Por otra parte, en el caso del enrutamiento *tunneling* esa propagación "baja" desde la parte superior del árbol hasta el elemento que produce el evento. Por último, el enrutamiento directo no propaga el evento en ninguna dirección y se comporta como los eventos "normales" a los que estamos acostumbrados.

Para comprender mejor estas nuevas definiciones, debemos tener en cuenta cómo funcionan o se comportan los elementos de una aplicación WPF, que como sabemos, en este tipo de aplicaciones están definidos en XAML. Por regla general, los elementos de una aplicación XAML suelen estar contenidos en otros elementos; de hecho, al igual que ocurre con las aplicaciones normales de Windows, siempre hay algún contenedor que "contiene" los controles, aunque sea el propio formulario, en el caso de las aplicaciones de Windows, o cualquiera de los contenedores de las aplicaciones de WPF.

La diferencia de las aplicaciones basadas en Windows Forms con las aplicaciones para WPF, es que, por ejemplo, un botón no solo es un botón, ya que tal como se crean las aplicaciones con XAML ese botón puede contener otros elementos, por ejemplo un bloque de texto o una imagen, y a diferencia de las aplicaciones "normales", esos elementos que contiene se pueden tratar de forma independiente. Pero en el caso de ese botón compuesto, no tendría mucho sentido tratar cada uno de los eventos de los elementos que lo

Guillermo "Guille" Som
Es Microsoft MVP de Visual Basic desde 1997. Es redactor de dotNetManía, mentor de Solid Quality Iberoamericana, tutor de campusMVP, miembro de Ineta Speakers Bureau Latin America, y autor de los libros "Manual Imprescindible de Visual Basic .NET" y "Visual Basic 2005".
http://www.elguille.info

componen de forma independiente, sino que nos resultaría más práctico tratarlos como un todo. En este caso los eventos enrutados nos pueden ser de mucha utilidad, en particular los de tipo *tunneling*, los que se producen primero en la parte superior del árbol de elementos, en nuestro caso, el propio botón. Veamos otros conceptos relacionados con estos eventos enrutados y cómo detectarlos antes de que lleguen al elemento que realmente lo produce.

Los tipos de enrutamiento de los eventos XAML

En WPF cada evento se puede detectar de dos formas diferentes, según lo queramos interceptar antes de llegar al elemento que realmente lo produce (*tunneling*), en cuyo caso el nombre del evento va precedido de `Preview`, lo que nos da una idea de cuál es la intención de dicho evento: tener la posibilidad de interceptarlo antes de que realmente se produzca. Por otro lado tenemos los eventos *bubbling*, a los que no se añade ningún prefijo. Como regla general, todos los eventos de los elementos de WPF van en pareja, y por cada evento suele haber uno previo y uno normal. Por ejemplo, la pulsación de las teclas suele detectarse con el evento `KeyDown`, evento de tipo *bubbling* (lo detectaríamos en el control que lo produce) y el correspondiente de tipo *tunneling* es `PreviewKeyDown` (para detectarlo antes de que llegue al control). Aclarar que este tipo de eventos emparejados también están disponibles en la versión 2.0 de .NET Framework, aunque evidentemente sin la misma potencia "enrutadora" que en .NET Framework 3.0.

Hay ciertos eventos que no van en parejas, aunque sí que suelen estar relacionados con otros eventos; por ejemplo, el evento `Click` de un botón sería del tipo *bubbling*, aunque no tiene emparejado el equivalente al enrutamiento *tunneling* (no existe un evento `PreviewClick`). En principio, podría parecer que ese evento es de tipo directo, ya que solo se produce en el control que lo define y, aparentemente, no tiene equivalente previo. Aunque en el caso de los eventos relacionados con el ratón, siempre hay formas de buscar los equivalentes previos; por ejemplo, el evento `Click` va precedido de varios eventos, entre ellos los que detectan la pulsación del botón izquierdo del ratón: `MouseLeftButtonDown` y `PreviewMouseLeftButtonDown`, aunque en este caso en particular del evento `Click`, el propio control que detecta la pulsación marca el evento como manipulado (`Handled`), impidiendo que se propague por el árbol de contenedores. Pero tal como XAML nos permite definir los eventos enrutados, también podemos indicar que se intercepte el evento `Click` de un botón en el contenedor (o padre) de ese botón; además, de forma independiente al evento interceptado por el propio botón.

Enrutamiento bubbling

La mejor forma de entender estos conceptos es viéndolos con un ejemplo.

Para mantener las cosas simples, el código XAML de ejemplo está muy simplificado (pero 100% operativo) y consiste en una ventana (`Window`) que contiene un `StackPanel` que a su vez contiene dos botones y dos etiquetas; en el `StackPanel` definimos un evento que interceptará la pulsación en cualquiera de los botones que contenga, además, de forma independiente, cada botón define su propio evento `Click`. Para saber cuál es el orden en el que se producen los eventos tengo definida una función que simplemente incrementa una variable y devuelve el valor de la misma. En los fuentes 1 y 2 tenemos tanto la definición del código XAML como el correspondiente al uso desde C# para interceptar esos eventos.

El evento `Click` es de tipo *bubbling*, por tanto, primero se produce en el elemento que provoca el evento y después se propaga al resto de elementos que están en el mismo árbol, es decir, al contenedor del control y al contenedor del contenedor, etc.; en nuestro caso al `StackPanel` y, si así lo hubiéramos previsto, al objeto `Window`. Por tanto, tal como está el código del fuente 2, primero se mostrará el mensaje en la etiqueta `labelInfo2` y después en `labelInfo`.

En el elemento `StackPanel` indicamos que queremos interceptar el evento `Click` de los botones que contenga este control y en el código del método que intercepta ese evento mostramos el nombre del control que lo produce. En ese código usamos la propiedad `OriginalSource` del objeto recibido en el segundo parámetro, aunque en este caso particular también nos hubiera valido usar el valor devuelto por la propiedad

```
<Window x:Class="dnm.eventosXAML01_cs.Window1" ... >
    <StackPanel Name="stackPanel1" Button.Click="stackPanel_ButtonClick">
      <Button Name="btnUno" Content="Uno" Click="btnUno_Click" />
      <Button Name="btnDos" Content="Dos" Click="btnDos_Click" />
      <Label Name="labelInfo" Content="Info" />
      <Label Name="labelInfo2" Content="Info2" Background="LightBlue" />
    </StackPanel>
</Window>
```

Fuente 1. Código XAML del primer ejemplo

```
private void stackPanel_ButtonClick(object sender, RoutedEventArgs e)
{
    Button btn = (Button)e.OriginalSource;
    labelInfo.Content = "Has pulsado en " + btn.Name + " " + total();
}
private void btnUno_Click(object sender, RoutedEventArgs e)
{
    labelInfo2.Content = "Botón Uno " + total();
}
private void btnDos_Click(object sender, RoutedEventArgs e)
{
    labelInfo2.Content = "Botón Dos " + total();
}
```

Fuente 2. Código de C# del primer ejemplo

```
<Window x:Class="dnm.eventosXAML01b_cs.Window1" ...
    Button.Click="Window1_ButtonClick" >
```

Fuente 3. Modificando el fuente 1 podemos interceptar en el objeto `Window` el evento `Click` de los botones

```
private void Window1_ButtonClick(object sender, RoutedEventArgs e)
{
    Button btn = (Button)e.OriginalSource;
    labelInfo.Content += "\n(Window)Has pulsado en " + btn.Name + " " + total();
    labelInfo.Content += '\n' + ((FrameworkElement)e.Source).ToString();
}
private void stackPanel_ButtonClick(object sender, RoutedEventArgs e)
{
    Button btn = (Button)e.OriginalSource;
    labelInfo.Content = "(StackPanel)Has pulsado en " + btn.Name + " " + total();
    labelInfo.Content += '\n' + ((FrameworkElement)e.Source).ToString();
}
// Al indicar e.Handled = true el evento no se propagará.
private void btnUno_Click(object sender, RoutedEventArgs e)
{
    labelInfo2.Content = "Botón Uno " + total();
    labelInfo.Content = "";
    e.Handled = true;
}
```

Fuente 4. Cambios al fuente 2 para usarlo con el fuente 3

```
<Window x:Class="dnm.eventosXAML02_cs.Window1" ... >
  <StackPanel Name="stackPanel1" PreviewMouseMove="stackPanel_MouseMove">
    <Button Name="btnUno" Content="Uno" MouseMove="btnUno_MouseMove" />
    <Button Name="btnDos" Content="Dos" MouseMove="btnDos_MouseMove" />
    <Label Name="labelInfo" Content="Info" />
    <Label Name="labelInfo2" Content="Info2" Background="LightBlue" />
  </StackPanel>
</Window>
```

Fuente 5. Código XAML del ejemplo de *tunneling*

`Source`. Como hemos comentado, el evento interceptado por ese método se producirá después de que se haya producido el evento en cada uno de los botones. En cualquier caso, si dentro de los métodos que interceptan los eventos particulares de cada botón quisiéramos dejar de "reproducir" el evento, es decir, evitar que se propague a los contenedores del botón, podemos asignar un valor verdadero a la propiedad `Handled` del segundo parámetro.

En el código de los fuentes 3 y 4 tenemos los cambios a realizar en el ejemplo anterior para propagar el evento `Click` de los botones al objeto `Window`, y para marcar como manejado dicho evento, pudiendo comprobar todo lo aquí comentado.

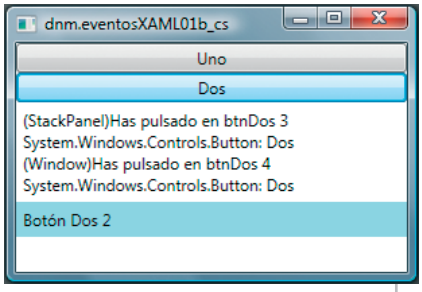

Figura 1. El segundo ejemplo en ejecución después de pulsar en el botón "Dos"

En la figura 1 vemos la aplicación en ejecución después de haber pulsado en el segundo botón, en la que podemos apreciar el orden en el que se ejecuta el código.

Enrutamiento tunneling

La otra forma de "enrutar" los eventos en XAML es la técnica conocida como *tunneling*, que consiste en la propagación de los eventos desde el contenedor de nivel más superior hasta el elemento que en realidad produce el evento. Los eventos clasificados en el tipo *tunneling* contienen en el nombre el prefijo `Preview`. Por ejemplo, si queremos detectar el movimiento del ratón en una serie de controles de una aplicación de tipo WPF, podríamos escribir un código como el mostrado en el fuente 5.

En el control `StackPanel` definimos el evento `PreviewMouseMove`, además en cada uno de los dos botones que contiene ese panel hay definido también un evento `MouseMove`. El primero se producirá antes que cualquiera de los otros dos, al menos cuando el ratón se mueva por cualquiera de esos dos botones. Pero al estar definido a nivel del contenedor, ese evento será capaz de detectar el movimiento del ratón en cualquiera de los controles que contiene además de en sí mismo.

Con el código del fuente 6 podemos saber en qué control está el ratón en cada momento y gracias al valor devuelto por la función `total()` también sabremos en qué orden se ejecutan los eventos, al menos si ese movimiento de ratón lo hacemos sobre cualquiera de los dos botones, ya que si movemos el ratón encima de cualquiera de las dos etiquetas, también se producirá (e interceptará) el evento a nivel del panel.

Como ya sabemos, cada control de las aplicaciones XAML en realidad está formado por otros controles. Por ejemplo, las etiquetas (o los botones) para mostrar el texto en realidad utilizan un control del tipo `TextBlock`, y si vemos el resultado mostrado en la figura 2, nos daremos cuenta que el control que se recibe en la propiedad `OriginalSource` del segundo parámetro del método `stackPanel_MouseMove` en realidad es un control `TextBlock`, mientras que el control indicado por la propie-

```
private void stackPanel_MouseMove(object sender, RoutedEventArgs e)
{
    FrameworkElement elem = (FrameworkElement)e.Source;
    labelInfo.Content = "El ratón está en " + elem.Name + " " + total();
    // OriginalSource y Source no tienen porqué ser lo mismo
    labelInfo.Content += "\nSource= " + e.Source.ToString();
    labelInfo.Content += "\nOriginalSource= " +
            e.OriginalSource.ToString();
}
private void btnUno_MouseMove(object sender, MouseEventArgs e)
{
    labelInfo2.Content = "El ratón está en el botón Uno " + total();
}
private void btnDos_MouseMove(object sender, MouseEventArgs e)
{
    labelInfo2.Content = "El ratón está en el botón Dos " + total();
}
```

Fuente 6. Código C# del ejemplo de *tunneling*

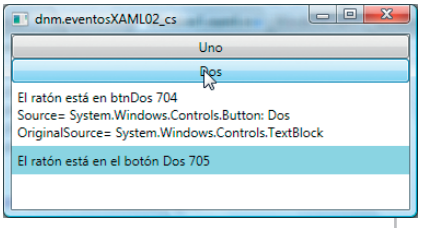

Figura 2. Resultado de ejecutar el código del fuente 6

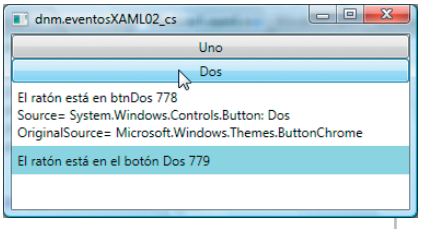

Figura 3. El origen del evento puede ser cualquier elemento que forme el control

dad `Source` hace referencia al propio botón en el que se está efectuando el movimiento del ratón.

Y si desplazamos el cursor del ratón sobre una parte que no tiene texto, tal como vemos en la figura 3, nos indicará que el origen "real" del evento es "algo" llamado `ButtonChrome`, que como podemos suponer (al menos viendo el espacio de nombres al que pertenece), es un elemento usado para "pintar" el color dependiendo del tema actual que tengamos en nuestro equipo.

RoutedEventArgs: tipo base de los eventos enrutados

A diferencia de las aplicaciones "normales" (las basadas en Windows Forms), los tipos de datos usados como segundo parámetro de los métodos que interceptan los eventos en las aplicaciones XAML (o que usan los controles de WPF) se derivan (directa o indirectamente) del tipo `RoutedEventArgs`. Como acabamos de ver, ese tipo define ciertas propiedades que nos permiten saber qué elemento (o control) es el que realmente ha producido el evento.

Si nos fijamos bien en el código del fuente 6, particularmente en la definición del método que intercepta el evento `PreviewMouseMove`, veremos que el segundo parámetro lo hemos definido como `RoutedEventArgs`, cuando en realidad el delegado que define ese evento indica que ese segundo parámetro debería ser del tipo `MouseEventArgs` (que es el que hemos usado en los otros dos métodos). Sin embargo, el código ha compilado correctamente; esto es así, al menos en C#, porque este lenguaje permite usar cualquier clase base que tenga la clase esperada como parámetro, algo conocido como *contravarianza* (ver **dotNetManía** número 30). Sin embargo, si nuestro lenguaje de programación es Visual Basic, siempre tendremos que indicar el tipo de parámetro correcto.

Eventos XAML definidos como estilos

No son solo los eventos "normales", es decir, los definidos directa y explícitamente, los que podemos enrutar en XAML, ya que gracias a los *triggers* (desencadenadores) podemos "jugar" un poco con los eventos en las aplicaciones XAML. No voy a entrar en demasiados detalles sobre esta forma de definir eventos, ya que en el número 27 de esta misma revista **Luis Miguel Blanco** habló con detalle de los estilos XAML en general, así como de los desencadenadores relacionados con los eventos. Pero para aquellos lectores que no tienen ese número de **dotNetManía** a mano, y para cerrar este artículo con algo puramente XAML, veamos cómo podemos definir las acciones que queremos que se realicen cuando un evento en particular ocurra, pero sin necesidad de escribir ni una sola línea de código, al menos de código en C# o Visual Basic, ya que ese código sí que habrá que escribirlo en XAML.

En el fuente 7 podemos ver cómo "ligar" ciertos eventos con los controles de tipo `Button` que definamos en nuestra aplicación, particularmente para que se ejecuten ciertas acciones sobre determinadas propiedades, en este caso usando un poco de "animación" por medio de elementos de tipo `Storyboard`. En ese código hemos definido un estilo para todos los controles de tipo `Button`, y en particular interceptamos dos eventos: `MouseEnter` y `MouseLeave`. Esos eventos los indicamos por medio del atributo (o propiedad) `RoutedEvent` del elemento `EventTrigger`. En cada `Storyboard` (el guión de la película que nos queremos montar con ese evento) le decimos qué es lo que queremos que haga cuando se produzca ese evento. En este caso, en el evento `MouseEnter` indicamos que queremos animar la propiedad `Width` hasta que alcance el valor indicado. En el evento `MouseLeave` decimos que también queremos animar esa misma propiedad del control, pero al no indicar el valor, se usará el que tuviera inicialmente. Aclarar que si pretendemos "animar" una propiedad que no está previamente definida con un valor inicial, recibiremos una excepción.

```
<Window x:Class="dnm.eventosXAML03_cs.Window1" ... >
  <Window.Resources>
    <Style TargetType="{x:Type Button}">
      <Style.Triggers>
        <EventTrigger RoutedEvent="Mouse.MouseEnter">
          <EventTrigger.Actions>
            <BeginStoryboard>
              <Storyboard>
                <DoubleAnimation Duration="0:0:.5"
                  Storyboard.TargetProperty="Width" To="270" />
              </Storyboard>
            </BeginStoryboard>
          </EventTrigger.Actions>
        </EventTrigger>
        <EventTrigger RoutedEvent="Mouse.MouseLeave">
          <EventTrigger.Actions>
            <BeginStoryboard>
              <Storyboard>
                <DoubleAnimation Duration="0:0:1"
                  Storyboard.TargetProperty="Width" />
              </Storyboard>
            </BeginStoryboard>
          </EventTrigger.Actions>
        </EventTrigger>
      </Style.Triggers>
    </Style>
  </Window.Resources>
  <Canvas>
    <Button Name="btnUno" Content="Uno"
            Canvas.Left="10" Canvas.Top="10" Width="100" Height="26" />
    <Button Name="btnDos" Content="Dos"
            Canvas.Left="10" Canvas.Top="40" Width="100" Height="26" />
    <Label Content="Mueve el ratón por los botones"
           Background="Indigo" Foreground="White"
           Canvas.Left="10" Canvas.Top="80" />
  </Canvas>
</Window>
```

Fuente 7. Eventos y acciones definidas directamente en XAML por medio de estilos

NOTA

Aclarar que solo podremos acceder desde el código a los controles creados en el fichero XAML si esos controles tienen la propiedad **Name** (o el atributo **x:Name**) asignada (podemos usar indistintamente cualquiera de las dos "propiedades", pero solo una de ellas en cada control o elemento XAML). Además, debemos compilar el proyecto para que se creen las definiciones de esos controles en el código y así podamos acceder a ellos. Sin estos dos "requisitos" los controles no serán accesibles desde el código de nuestro lenguaje favorito.

Si queremos definir esos eventos para cualquier tipo de control, tendremos que cambiar el valor indicado en **TargetType**, pero como hemos comprobado, ese estilo (y sus correspondientes eventos) solo se aplicarán a los controles que coincidan con ese "tipo destino". Si quisiéramos definir ese estilo para diferentes tipos de controles, podríamos definirlo como un estilo con nombre; de esa forma, podríamos aplicar dicho estilo de forma independiente a ciertos controles.

En el código que acompaña al artículo he definido el mismo estilo del código fuente 7 (con pequeños cambios), pero al definirlo como un estilo con nombre, solo será aplicable a los controles que "quieran" usar ese estilo en concreto. En nuestro ejemplo, lo aplicamos a uno de los dos botones y a la etiqueta. En ese código, no solo animamos el ancho del control, sino también el tamaño de la fuente. Vuelvo a recordar que esto solo funcionará si esos controles han definido de forma explícita los valores iniciales de las propiedades a animar.

Conclusiones

En este artículo hemos dado un pequeño repaso a los distintos tipos de eventos que podemos utilizar en las aplicaciones para WPF, centrándonos en los eventos enrutados (*routed events*), de los cuales como hemos visto existen principalmente dos tipos, según la forma que tenga el evento de propagarse. También hemos visto cómo podemos interceptarlos de forma generalizada o de forma concreta, todo dependiendo de dónde definamos el evento. Por último, hemos comprobado que por medio de los desencadenadores (*triggers*) podemos definir e interceptar eventos usando solo código XAML.

Quisiera aclarar que todo el código aquí mostrado se puede usar sin necesidad de utilizar ninguna versión beta, ya que lo único que necesitamos es tener instalado Visual Studio 2005 y la versión 3.0 de .NET Framework, versión que ya está incluida en Windows Vista, pero que también podemos instalar en cualquier equipo con Windows XP SP2 o Windows 2003 SP1. Por supuesto, la forma más fácil de crearlos es usando las extensiones "Orcas" para Visual Studio 2005, pero solo si queremos ver el diseño de los "formularios" mientras escribimos el código XAML.

En la Web de la revista está disponible todo el código de ejemplo, tanto para Visual Basic como para C#. En el ZIP con el código he añadido dos proyectos vacíos que puede usar como plantillas para crear sus proyectos para WPF, ya que en una instalación normal de Visual Studio 2005 no existen esos tipos de proyectos ni tampoco los ficheros de tipo **.xaml** usados para incluir el código XAML.