Estilos en XAML

Luis Miguel Blanco Ancos

¿Qué es un estilo?

Dentro del contexto del diseño de interfaces de usuario, un estilo consiste en aquella característica disponible en una herramienta de desarrollo, léase IDE, lenguaje, diseñador visual, etc., que permite definir unas reglas comunes de aspecto, aplicables a un conjunto de elementos de nuestro interfaz.

XAML como lenguaje para definir interfaces

XAML (Extensible Application Markup Language), consiste en un esquema XML desarrollado para la creación de interfaces gráficos en WPF, utilizando como su propio nombre indica, una sintaxis de marcado para definir los elementos visuales. Dado que este artículo versa sobre uno de los aspectos del interfaz de usuario, y por lo tanto, será necesario crear ventanas de ejemplo con algunos controles básicos, se asume por parte del lector un mínimo conocimiento de XAML; en caso contrario, le remito al número 19 de dotNetManía, en cuyo artículo “Primeros pasos con Windows Presentation Foundation”, encontrará una magnífica introducción a este nuevo subsistema gráfico.

Creando el proyecto de ejemplo

Los ejemplos presentados a lo largo de este artículo se han desarrollado utilizando la versión CTP de WinFX correspondiente a Enero de 2006, aunque si el lector dispone de una versión posterior, también deberían de funcionarle correctamente. Igualmente, como entorno de desarrollo se ha utilizado Visual Studio 2005 (VS 2005 en adelante), ya que una vez instalados ambos productos, en VS 2005 disponemos de plantillas de proyecto específicas para los diferentes tipos de aplicaciones que podemos desarrollar con WinFX. La figura 1 muestra el IDE de VS 2005, con la creación del proyecto EstilosXAML, utilizado para algunos de los ejemplos de este artículo.

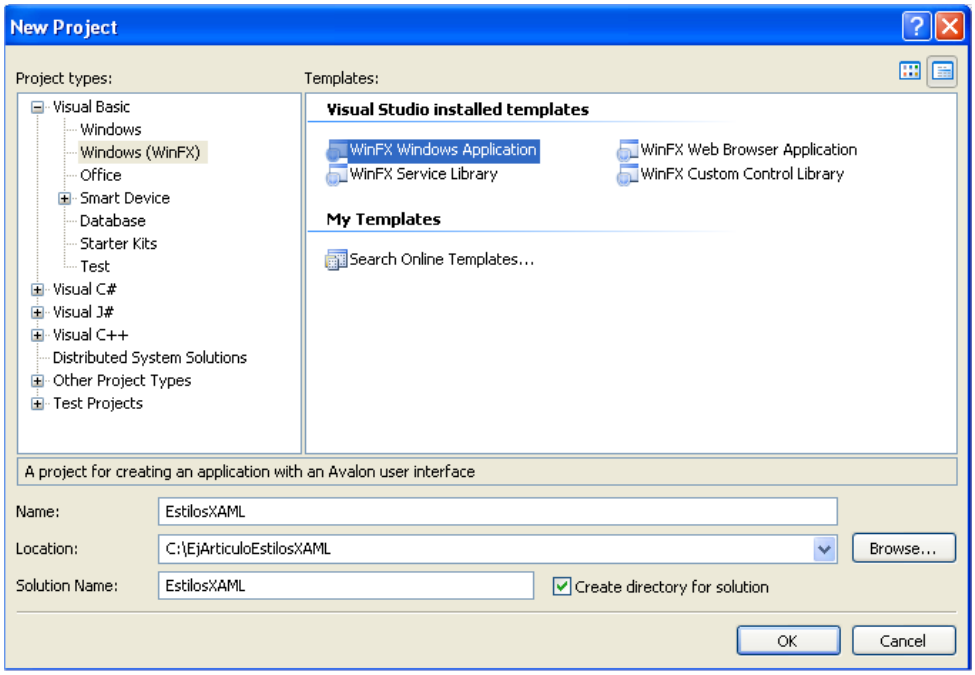

Figura 1. Proyecto WinFX Windows Application.

El lector puede encontrar todos los ejemplos expuestos a lo largo de este artículo en la dirección www.dotnetmania.com.

La utilidad de los estilos

Supongamos que comenzamos a diseñar una ventana con varios controles TextBox, los cuales deben tener la misma apariencia. Para ello escribimos el código que vemos en el fuente 1.

```
<Window x:Class="Window1"
 xmlns="http://schemas.microsoft.com/winfx/avalon/2005"
 xmlns:x="http://schemas.microsoft.com/winfx/xaml/2005"
 Title="EstilosXAML" Height="350" Width="400">

 <Canvas>
  <TextBox Name="txtNombre"
   Canvas.Left="10" Canvas.Top="5"
   Background="PaleGreen" FontWeight="Bold"/>

  <TextBox Name="txtApellidos"
   Canvas.Left="10" Canvas.Top="40"
   Background="PaleGreen" FontWeight="Bold"/>

  <TextBox Name="txtCiudad"
   Canvas.Left="10" Canvas.Top="80"
   Background="PaleGreen" FontWeight="Bold"/>
```

```
 </Canvas>
</Window>
```

Fuente 1.

El aspecto de la ventana quedaría como vemos en la figura 2.

Figura 2. Ventana con controles sin estilo.

Revisando el código del fuente 1, comprobamos que para que todos los TextBox tengan el mismo aspecto en cuanto a color y letra, debemos repetir el mismo código uno por uno, lo cual supone un mayor esfuerzo de codificación.

Podemos aliviar parte de esta tarea repetitiva, y al mismo tiempo organizar mejor nuestro código, si creamos un estilo para a este tipo de control y lo aplicamos en nuestra ventana, como muestra el fuente 2.

```
<Window x:Class="Window1"
 xmlns="http://schemas.microsoft.com/winfx/avalon/2005"
 xmlns:x="http://schemas.microsoft.com/winfx/xaml/2005"
 Title="EstilosXAML" Height="350" Width="400">

 <Window.Resources>
  <Style TargetType="{x:Type TextBox}">
   <Setter Property="Background" Value="PaleGreen"></Setter>
   <Setter Property="FontWeight" Value="Bold"/>
  </Style>
 </Window.Resources>

 <Canvas>
  <TextBox Name="txtNombre"
   Canvas.Left="10" Canvas.Top="5" />

  <TextBox Name="txtApellidos"
   Canvas.Left="10" Canvas.Top="40" />
```

```
  <TextBox Name="txtCiudad"
    Canvas.Left="10" Canvas.Top="80" />
 </Canvas>

</Window>
```

Fuente 2.

En el fuente 2, dentro de la etiqueta de recursos de la ventana, Window.Resources, definimos un estilo mediante la etiqueta Style.

El atributo TargetType, perteneciente a la etiqueta Style, nos permite especificar el tipo de control sobre el que aplicamos el estilo. Seguidamente, para establecer las características que mostrará el control al que se aplica el estilo, usaremos un asignador de propiedad, representado por la etiqueta Setter, en la que mediante los atributos Property y Value, asignamos un valor a una propiedad del tipo de control destinatario del estilo; como es natural, podemos utilizar tantos asignadores, como propiedades del control queramos incluir en el estilo.

Como consecuencia de la creación de nuestro estilo, por cada control TextBox existente en el código XAML de la ventana, automáticamente se le aplicarán las propiedades definidas en el estilo, ahorrándonos el trabajo de establecer dichas propiedades independientemente por cada control, suponiendo ello una mejor organización y mantenimiento de las mismas, al estar situadas en un punto centralizado de nuestro código.

Estilos de aplicación explícita, o estilos con nombre

En el ejemplo de estilo comentado en el apartado anterior, podemos ver que no es necesario indicar a cada uno de los controles TextBox el estilo a utilizar, dado que al ser declarado con el atributo TargetType, ya se aplica automáticamente a los controles. A este modo de utilización del estilo se le denomina también estilo de aplicación implícita.

Pero supongamos que no queremos que un estilo se aplique a todos los controles de un mismo tipo uniformemente, o bien, queremos tener estilos que puedan ser asignados a tipos distintos de controles; es el momento de recurrir a los estilos de aplicación explícita, también denominados estilos con nombre.

Los estilos con nombre se caracterizan porque al ser declarados se emplea el atributo x:Key, con el cual asignamos un nombre al estilo. Posteriormente, desde el código XAML de un control, invocaremos el estilo mediante su atributo Style y la sintaxis {StaticResource NombreEstilo}. Esta técnica nos permite aplicar selectivamente estilos a los controles, lo que en ciertos escenarios puede ser más conveniente que la aplicación global de un estilo a todos los controles de un determinado tipo.

En el fuente 3 ampliamos el código de nuestra ventana de ejemplo, añadiendo un estilo con nombre, que aplicaremos a ciertos controles Label, y a un TextBox.

```
<Window x:Class="Window1" .... >
 <Window.Resources>
  ....
  <Style x:Key="styVarios">
   <Setter Property="Control.BorderBrush" Value="Orange"></Setter>
   <Setter Property="Control.BorderThickness" Value="3"></Setter>
   <Setter Property="Control.Background" Value="Aqua"></Setter>
  </Style>
 </Window.Resources>

 <Canvas>
  <Label
   Canvas.Left="10" Canvas.Top="5"
   Content="Nombre:"
   Style="{StaticResource styVarios}" />

  <TextBox Name="txtNombre"
   Canvas.Left="75" Canvas.Top="5" />

  <Label
   Canvas.Left="10" Canvas.Top="40"
   Content="Apellidos:"
   Style="{StaticResource styVarios}" />

  <TextBox Name="txtApellidos"
   Canvas.Left="75" Canvas.Top="40" />

  <Label
   Canvas.Left="10" Canvas.Top="80"
   Content="Ciudad:" />

  <TextBox Name="txtCiudad"
   Canvas.Left="75" Canvas.Top="80"
   Style="{StaticResource styVarios}" />
 </Canvas>
</Window>
```

Fuente 3.

La figura 3 muestra el resultado de ejecución de la ventana.

Figura 3. Ventana utilizando estilo con nombre.

Nótese que en el caso de los estilos con nombre, al definir el asignador, el atributo Property debe especificarse con el formato Clase.Propiedad, siendo Clase el tipo de la plataforma correspondiente al control sobre el que se aplica, o un tipo más genérico en la jerarquía de controles; es por esto que en el ejemplo anterior también podríamos haber declarado las etiquetas Setter como muestra el fuente 4.

```
....
<Setter Property="Label.BorderBrush" Value="Orange"></Setter>
....
```

Fuente 4.

El lector puede pensar en este momento que si utilizamos Label.Propiedad en el asignador, el estilo sólo sería válido para ser aplicado a controles Label, sin embargo, aunque esto parece lo más lógico no es así, ya que el control TextBox al que se aplica el estilo también lo aceptaría.

El motivo de este comportamiento reside en que al ser aplicado el asignador, este es aceptado tanto por controles que se encuentran en el mismo árbol de jerarquía, como en aquellos que comparten la implementación de la propiedad, caso este último que es el que nos ocupa actualmente.

El control Label se encuentra en el espacio de nombres System.Windows.Controls.ContentControl, mientras que TextBox se halla ubicado en System.Windows.Controls.Primitives.TextBoxBase, sin embargo, la implementación de las propiedades que usamos en el estilo styVarios del ejemplo es compartida por ambas clases, y esto hace que podamos declarar en el atributo Property del asignador cualquiera de las dos clases, con la seguridad de que el estilo será aplicado indistintamente a controles de ambos tipos, como vemos en el fuente 5.

```
....
// podemos declarar el asignador así
<Setter Property="Label.BorderBrush" Value="Orange"></Setter>
....
// y también así
<Setter Property="TextBox.BorderBrush" Value="Orange"></Setter>
```

....

Fuente 5.

También puede darse el caso contrario: controles con propiedades específicas, las cuales no existen en otros controles de nuestra ventana. En esta situación igualmente podemos crear un único estilo y asignadores para este tipo de propiedades.

Como ejemplo ilustrativo tenemos el control TextBox con su propiedad Text, y por otra parte el control CheckBox con su propiedad IsChecked. El fuente 6 demuestra cómo es posible crear un estilo con un asignador para cada una de estas propiedades, y utilizarlo sin que los asignadores entren en conflicto. Al ser aplicado el estilo, lo que sucederá es que si un asignador no encuentra la propiedad en el control de destino, simplemente no será tenido en cuenta.

```
<Window.Resources>
        <Style x:Key="styNuevo">
                <Setter Property="TextBox.Text" Value="Introduzca su nombre" />
                <Setter Property="CheckBox.IsChecked" Value="True" />
        </Style>
</Window.Resources>

<Canvas>
        <TextBox Canvas.Left="5" Canvas.Top="5" Style="{StaticResource styNuevo}" />

        <CheckBox Canvas.Left="150" Canvas.Top="5"
                Content="Comprobar" Style="{StaticResource styNuevo}" />
</Canvas>
```

Fuente 6.

Para el código del fuente 6 se ha creado en el proyecto una nueva ventana con el nombre Window2; en el caso de necesitar que la ventana inicial sea esta, debemos editar el archivo MyApp.xaml, y en el atributo StartupUri asignar el nombre de la ventana inicial del proyecto, como vemos en el fuente 7.

```
<Application x:Class="MyApp"
   xmlns="http://schemas.microsoft.com/winfx/avalon/2005"
   xmlns:x="http://schemas.microsoft.com/winfx/xaml/2005"
   StartupUri="Window3.xaml">
   <Application.Resources>

   </Application.Resources>
</Application>
```

Fuente 7.

Herencia de estilos

Gracias al atributo BasedOn de la etiqueta Style, podemos crear un estilo hijo a partir de un estilo definido anteriormente, de manera que el nuevo estilo tendrá todos los asignadores del estilo padre más los propios que él defina, como vemos en el ejemplo del fuente 8.

```
<Window.Resources>
        <Style x:Key="styPadre">
                <Setter Property="Button.Background" Value="SkyBlue"/>
                <Setter Property="Button.Foreground" Value="Red"/>
        </Style>

        <Style x:Key="styHijo" BasedOn="{StaticResource styPadre}">
                <Setter Property="Button.FontSize" Value="18"/>
                <Setter Property="Button.Opacity" Value="0.5"/>
        </Style>
</Window.Resources>

<Canvas>
        <Button
                Canvas.Left="5" Canvas.Top="5"
                Style="{StaticResource styPadre}">
                ACEPTAR
        </Button>

        <Button
                Canvas.Left="100" Canvas.Top="5"
                Style="{StaticResource styHijo}">
                CANCELAR
        </Button>
</Canvas>
```

Fuente 8.

En la figura 4 podemos apreciar la diferencia en la aplicación de ambos estilos.

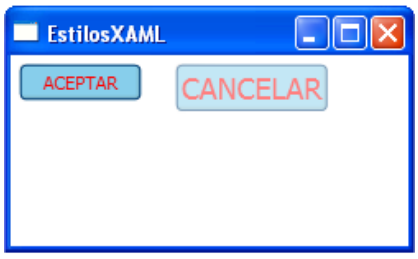

Figura 4. Controles con herencia de estilos.

Ámbito de estilos

El radio de acción que abarca un estilo depende básicamente del lugar dentro del código en donde el estilo es declarado. En los ejemplos vistos hasta el momento hemos declarado los estilos en la etiqueta de recursos de la ventana, de forma que todos los controles contenidos en la misma recibían el estilo si ello era menester.

No obstante podemos reducir o ampliar esta zona de influencia del estilo; por ejemplo, si lo declaramos en el archivo MyApp.xaml del proyecto, dentro de la etiqueta Application.Resources, el estilo se aplicará globalmente a todos los controles de la aplicación definidos por el estilo. Como contrapartida, si creamos un estilo en un elemento con un rango de influencia menor, como puede ser el Canvas de una ventana, sólo afectará a los controles situados dentro de dicho elemento.

Para ilustrar las mencionadas situaciones, vamos a crear un proyecto con el nombre AmbitoEstilos, que contenga dos ventanas. A nivel de aplicación crearemos un estilo que aplicaremos a los controles Button; en los recursos de una de las ventanas también añadiremos un estilo para los TextBox; y dentro del Canvas, el estilo será para los CheckBox, como vemos en el fuente 9.

```
<!--MyApp.xaml-->
<Application x:Class="MyApp"
 xmlns="http://schemas.microsoft.com/winfx/avalon/2005"
 xmlns:x="http://schemas.microsoft.com/winfx/xaml/2005"
 StartupUri="Window1.xaml">

 <Application.Resources>
  <Style TargetType="{x:Type Button}">
   <Setter Property="Background" Value="SkyBlue"/>
   <Setter Property="Width" Value="100"/>
   <Setter Property="Height" Value="40"/>
  </Style>
 </Application.Resources>
</Application>

<!--Window1.xaml-->
<Window x:Class="Window1"
 xmlns="http://schemas.microsoft.com/winfx/avalon/2005"
 xmlns:x="http://schemas.microsoft.com/winfx/xaml/2005"
 Title="AmbitoEstilos" Width="200" Height="200">
 <Window.Resources>
  <Style TargetType="{x:Type TextBox}">
   <Setter Property="FontSize" Value="18"/>
   <Setter Property="FontStyle" Value="Italic"/>
   <Setter Property="FontWeight" Value="UltraBold"/>
  </Style>
 </Window.Resources>
```

```
 <Canvas>
  <Canvas.Resources>
   <Style TargetType="{x:Type CheckBox}">
    <Setter Property="IsChecked" Value="True"/>
    <Setter Property="BorderThickness" Value="5"/>
   </Style>
  </Canvas.Resources>

  <TextBox Canvas.Left="5" Canvas.Top="5" Width="150" />
  <Button Name="btnAbrir" Canvas.Left="5" Canvas.Top="50" Content="Abrir
ventana" />
  <CheckBox Canvas.Left="5" Canvas.Top="100" Content="Comprobado"/>
 </Canvas>
</Window>

<!—codebehind de Window1.xaml-->
Private Sub btnAbrir_Click(ByVal sender As Object, ByVal e As
System.Windows.RoutedEventArgs) Handles btnAbrir.Click
   Dim oWin2 As Window2 = New Window2
   oWin2.Show()
End Sub

<!--Window2.xaml-->
<Window x:Class="Window2"
 xmlns="http://schemas.microsoft.com/winfx/avalon/2005"
 xmlns:x="http://schemas.microsoft.com/winfx/xaml/2005"
 Title="AmbitoEstilos" Width="200" Height="200">
 <Canvas>
  <Button Name="btnAceptar" Canvas.Left="5" Canvas.Top="5" Content="Aceptar" />
 </Canvas>
</Window>
```

Fuente 9.

En la figura 5 vemos el aspecto de los formularios de este ejemplo.

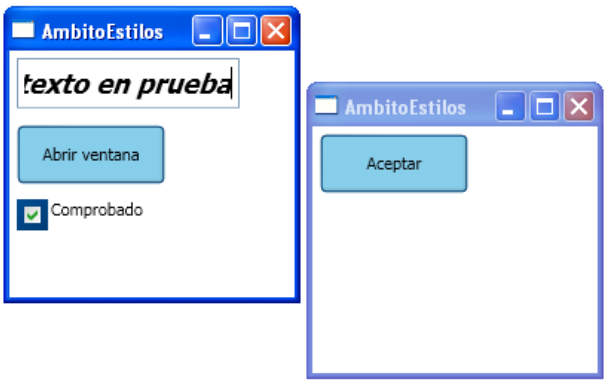

Figura 5. Estilos con diferente ámbito.

Estilos por código

Aunque en la mayoría de las ocasiones manipularemos nuestros estilos declarativamente desde XAML, también es posible realizar su creación y asignación desde el código de nuestra aplicación.

Para acceder a un estilo desde el codebehind de nuestra aplicación, debemos llamar al método FindResource() de la clase Window, pasándole como parámetro una cadena con el estilo a recuperar. Esta operación nos devolverá un objeto Style que podemos asignar a los controles de la ventana.

En el caso de que necesitemos crear los estilos dinámicamente en tiempo de ejecución, instanciaremos un objeto de la clase Style, a cuya colección Setters iremos añadiendo, también dinámicamente, tantos objetos asignadores como queramos que tenga el estilo.

Como ejemplo de creación dinámica de estilos, crearemos un nuevo proyecto con el nombre EstilosPorCodigo, con la siguiente definición de ventana en XAML, que vemos en el fuente 10.

```
<Window x:Class="Window1"
 xmlns="http://schemas.microsoft.com/winfx/avalon/2005"
 xmlns:x="http://schemas.microsoft.com/winfx/xaml/2005"
 Title="EstilosPorCodigo" Height="262" Width="336">

 <Window.Resources>
  <Style x:Key="styControles">
   <Setter Property="Control.FontWeight" Value="Heavy"/>

   <Setter Property="Control.Background">
    <Setter.Value>
     <LinearGradientBrush StartPoint="0,0.5" EndPoint="1,0.5">
      <LinearGradientBrush.GradientStops>
       <GradientStop Color="GreenYellow" Offset="0"/>
       <GradientStop Color="Cyan" Offset="0.30"/>
       <GradientStop Color="Pink" Offset="0.70"/>
      </LinearGradientBrush.GradientStops>
     </LinearGradientBrush>
    </Setter.Value>
   </Setter>
  </Style>
 </Window.Resources>

 <Canvas>
  <Button Name="btnAceptar"
   Canvas.Left="10" Canvas.Top="10"
```

```
    Width="100" Height="50" Content="Aceptar"/>

  <ListBox Name="lstCiudades"
    Canvas.Left="10" Canvas.Top="100"
    Width="100" Height="100">
    <ListBoxItem>Bruselas</ListBoxItem>
    <ListBoxItem>París</ListBoxItem>
    <ListBoxItem>Londres</ListBoxItem>
  </ListBox>

  <CheckBox
    Name="chkAplicarEstilo"
    Canvas.Left="125" Canvas.Top="10">
    Aplicar estilo
  </CheckBox>

  <Button
    Name="btnCrearEstilo"
    Canvas.Left="125" Canvas.Top="50"
    Width="100" Height="25">
    Crear estilo
  </Button>

  <TextBox Name="txtNombre"
    Canvas.Left="125" Canvas.Top="90"
    Width="150" Height="50">
    Prueba
  </TextBox>
 </Canvas>
</Window>
```

Fuente 10.

A destacar en este fuente el asignador de estilo que establece el valor para la propiedad Background. Como puede comprobar el lector, en esta ocasión no asignamos un nombre simple de color para la propiedad, sino que utilizamos un objeto LinearGradientBrush, que crea el color a partir de un degradado basado en la mezcla de varios colores.

Puesto que se trata de un valor complejo para asignar a la propiedad Background del estilo, utilizaremos la etiqueta Setter.Value, incluyendo en su interior el valor para la propiedad.

Para aplicar el estilo creado en el XAML, utilizaremos un control CheckBox, mientras que para crear un nuevo estilo por código emplearemos un Button. El fuente 11 muestra el codebehind en el que se desarrolla esta tarea.

```
Partial Public Class Window1
   Inherits Window

   Private Sub chkAplicarEstilo_Click(ByVal sender As Object, ByVal e As
System.Windows.RoutedEventArgs) Handles chkAplicarEstilo.Click
      ' aplicar/quitar desde código un estilo definido en xaml
      If Me.chkAplicarEstilo.IsChecked Then
         Me.btnAceptar.Style = CType(Me.FindResource("styControles"), Style)
         Me.lstCiudades.Style = CType(Me.FindResource("styControles"), Style)
      Else
         Me.btnAceptar.Style = Nothing
         Me.lstCiudades.Style = Nothing
      End If
   End Sub

   Private Sub btnCrearEstilo_Click(ByVal sender As Object, ByVal e As
System.Windows.RoutedEventArgs) Handles btnCrearEstilo.Click
      ' crear estilo por código y aplicarlo a un control
      Dim styCaja As Style = New Style(GetType(TextBox))
      styCaja.Setters.Add(New Setter(BackgroundProperty, Brushes.Aquamarine))
      styCaja.Setters.Add(New Setter(FontFamilyProperty, New FontFamily("Arial
Black")))
      styCaja.Setters.Add(New Setter(BorderThicknessProperty, New Thickness(7)))

      Resources.Add("styCajaTexto", styCaja)
      Me.txtNombre.Style = styCaja
   End Sub
End Class
```

Fuente 11.

La figura 6 muestra la ventana en ejecución después de haberle aplicado estos estilos por código.

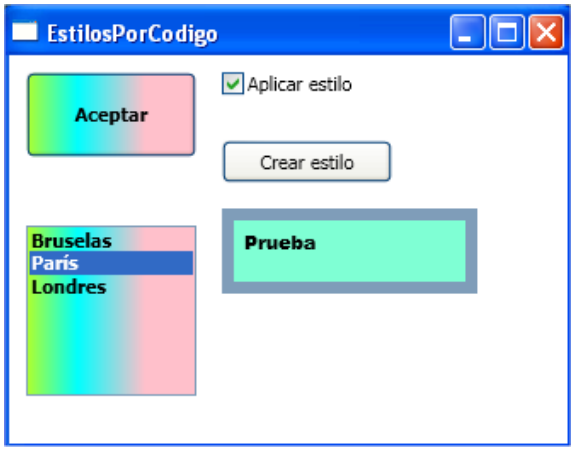

Figura 6. Estilos aplicados por código.

Aplicación directa de estilo en la declaración del control

Si bien esta técnica en rara vez la usaremos, ya que implica un trabajo adicional de codificación y un nulo reaprovechamiento del estilo, debemos saber que es posible crear estilos en la propia declaración del control, como vemos en el fuente 12.

```
<Button Name="btnHola"
        Canvas.Top="150" Canvas.Left="20"
        Height="30">
        <Button.Style>
                <Style>
                        <Setter Property="Button.Background"
Value="Yellow"></Setter>
                        <Setter Property="Button.FontSize" Value="16"></Setter>
                        <Setter Property="Button.Width" Value="200"></Setter>
                </Style>
        </Button.Style>
        Hola
</Button>
```

Fuente 12.

Estilos y plantillas de datos

Supongamos que creamos un proyecto con el nombre EstilosPlantillasDatos, y escribimos una clase cuyos objetos queremos visualizar a través de un control situado en la ventana del proyecto. El código de la clase podemos verlo en el fuente 13.

```
Public Class Empleado
   Private nCodigo As Integer
   Private sNombre As String

   Public Property Codigo() As Integer
      Get
         Return nCodigo
      End Get
      Set(ByVal value As Integer)
         nCodigo = value
      End Set
   End Property

   Public Property Nombre() As String
      Get
         Return sNombre
      End Get
      Set(ByVal value As String)
```

```
            sNombre = value
        End Set
    End Property

    Public Sub New(ByVal nCod As Integer, ByVal sNom As String)
        nCodigo = nCod
        sNombre = sNom
    End Sub
End Class
```

Fuente 13.

A continuación escribimos el código de definición de la ventana que tenemos en el fuente 14.

```
<Window x:Class="Window1"
 xmlns="http://schemas.microsoft.com/winfx/avalon/2005"
 xmlns:x="http://schemas.microsoft.com/winfx/xaml/2005"
 Title="EstilosPlantillasDatos" Height="328" Width="352">

 <Canvas>
  <Button Name="btnAsignar"
   Canvas.Left="10" Canvas.Top="10" Content="Asignar objeto"/>

  <Label Name="lblPrueba"
   Canvas.Left="10" Canvas.Top="50"
   Width="200" Height="80"
   Background="LightBlue">
   Etiqueta de prueba
  </Label>
 </Canvas>
</Window>
```

Fuente 14.

Y por ultimo, en el evento Click del botón, instanciamos un objeto de la clase Empleado, y lo asignamos al control Label. Ver fuente 15.

```
Partial Public Class Window1
    Inherits Window
    '....
    Private Sub btnAsignar_Click(ByVal sender As Object, ByVal e As
System.Windows.RoutedEventArgs) Handles btnAsignar.Click
        Dim oEmpleado As Empleado = New Empleado(12350, "Alberto")
        Me.lblPrueba.Content = oEmpleado
    End Sub
```

```
End Class
```

Fuente 15.

Si esperamos que al pulsar btnAsignar, el control Label al que hemos asignado el objeto muestre las propiedades del mismo, vamos por el camino equivocado, ya que el control carece de la lógica necesaria para saber representar un tipo complejo de la plataforma, como vemos en la figura 7.

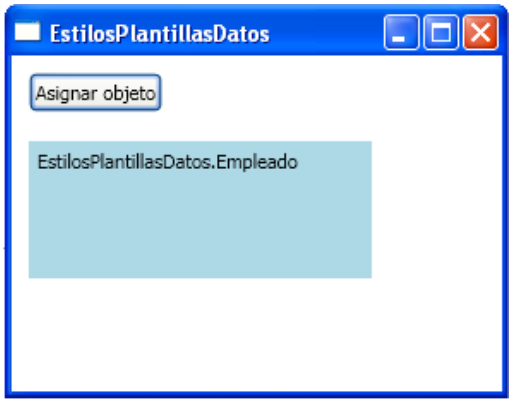

Figura 7. Objeto Empleado asignado a control Label.

Lo que está sucediendo en la figura 7 es que el control Label detecta que tiene un objeto Empleado en su propiedad Content, y sabe que de alguna manera debe interpretarlo y mostrarlo, pero al no haberle indicado la forma de visualizar el objeto, opta por llamar al método genérico ToString() que devuelve una cadena con el espacio de nombres y nombre de la clase Empleado.

Para solucionar este problema, debemos dotar al control Label de un comportamiento inteligente en este aspecto utilizando una plantilla de datos, representada en WPF por un objeto DataTemplate.

Un objeto DataTemplate permite, desde el código XAML de la ventana, especificar cómo van a ser representados visualmente los miembros de un tipo de .NET Framework a través de un control situado en dicha ventana; para ello, el elemento primario que debe tener un DataTemplate es un objeto contenedor (cualquier derivado de la clase Panel), dentro del cual especifiquemos los controles encargados de visualizar el objeto a representar.

Dado que necesitamos acceder desde el código XAML a un objeto de la plataforma gestionado por el CLR (la clase Empleado en este ejemplo), es preciso declarar este hecho, es decir, mapear el espacio de nombres en el que está la clase, para que sea accesible desde XAML; esto lo conseguimos mediante la etiqueta Mapping al comienzo del código de definición de la ventana, como vemos en el fuente 16.

```
<?Mapping XmlNamespace="mio" ClrNamespace="EstilosPlantillasDatos" ?>
```

Fuente 16.

Seguidamente en el código XAML también debemos especificar, mediante el atributo xmlns:EspacioNombresXML, el objeto que vaya a hacer uso del espacio de nombres que acabamos de mapear, en este caso la ventana, como muestra el fuente 17.

```
<Window x:Class="Window1"
 xmlns="http://schemas.microsoft.com/winfx/avalon/2005"
 xmlns:x="http://schemas.microsoft.com/winfx/xaml/2005"
 xmlns:mio="mio"
 Title="WindowsApplication1"
 Height="362" Width="426">
. . . .
. . . .
</Window>
```

Fuente 17.

El próximo paso consiste en crear el DataTemplate dentro de la zona de recursos de la ventana, especificando el tipo de objeto cuyos miembros va a representar en su interior, utilizando para ello el atributo DataType, y la sintaxis {x:Type EspacioNombres:Clase}. También deben declararse los controles a utilizar para representar el objeto, y dentro de cada uno de ellos, establecer el enlace o DataBinding con los miembros del objeto, usando la sintaxis {Binding Path=PropiedadClase}. Ver el fuente 18.

```
<Window.Resources>
  <DataTemplate DataType="{x:Type mio:Empleado}">
    <StackPanel Orientation="Vertical" Background="LawnGreen">
      <Label Content="{Binding Path=Codigo}" HorizontalAlignment="Left" />
      <TextBox Text="{Binding Path=Nombre}" />
    </StackPanel>
  </DataTemplate>
</Window.Resources>
```

Fuente 18.

Tras realizar todas estas operaciones, al ejecutar de nuevo nuestra ventana, el control Label lblPrueba ya sabrá como interpretar visualmente el objeto Empleado, mostrándose como vemos en la figura 8.

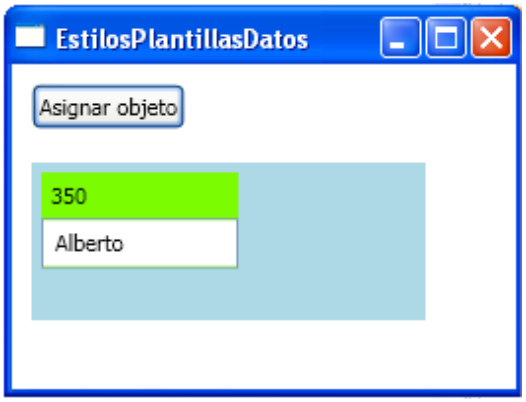

Figura 8. Objeto visualizado mediante un DataTemplate.

Pero estamos en un artículo sobre estilos, y el DataTemplate muestra el objeto “en bruto”, es decir, sin formateo alguno; es por ello que ahora vamos a crear estilos en los recursos de la ventana, que utilizaremos desde la plantilla de datos, para que el objeto Empleado tenga una apariencia adaptada a nuestros gustos. Veamos para ello el fuente 19.

```
 <Window.Resources>
  <Style x:Key="styEtiqueta">
   <Setter Property="Control.FontWeight" Value="Bold"/>
   <Setter Property="Control.Background" Value="Pink"/>
   <Setter Property="Control.Width" Value="30"/>
  </Style>

  <Style x:Key="styCaja">
   <Setter Property="Control.Background" Value="Gold"/>
  </Style>

  <DataTemplate DataType="{x:Type mio:Empleado}">
   <StackPanel Orientation="Vertical" Background="LawnGreen">
    <Label Content="{Binding Path=Codigo}" Style="{StaticResource styEtiqueta}"
HorizontalAlignment="Left" />
    <TextBox Text="{Binding Path=Nombre}" Style="{StaticResource styCaja}" />
   </StackPanel>
  </DataTemplate>
 </Window.Resources>
```

Fuente 19.

La información mostrada por la plantilla de datos al ejecutar de nuevo la ventana será la misma, pero su apariencia será muy diferente gracias a la aplicación del estilo, como vemos en la figura 9.

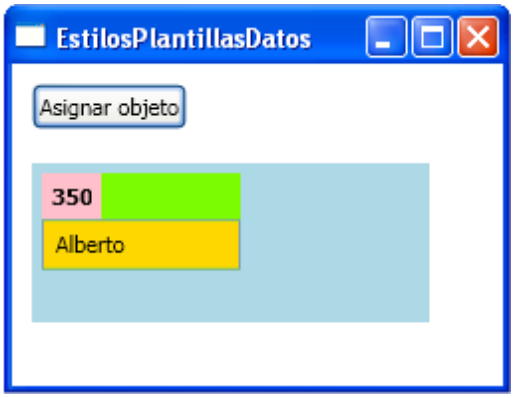

Figura 9. DataTemplate con estilos.

Desencadenadores de estilos

Un desencadenador o trigger, es aquel elemento que ejecuta una acción cuando ocurre una determinada circunstancia en nuestro código.

En el contexto de los estilos en XAML un desencadenador consiste en una etiqueta con el nombre Trigger, definida dentro de un estilo, que aplicará los asignadores que contiene siempre y cuando se cumpla la condición definida en el desencadenador.

Los desencadenadores de estilos en WPF se clasifican en varias categorías, en función del elemento sobre el que actúan; así tenemos desencadenadores para propiedades, para objetos del CLR, y para eventos. En los siguientes apartados trataremos cada tipo por separado.

Desencadenador para propiedad

Se trata del tipo más básico, y su funcionamiento se basa en el siguiente mecanismo: cuando la propiedad definida en el atributo Property tenga el valor indicado en el atributo Value, se aplicarán los asignadores definidos en el interior de la etiqueta Trigger. Como ejemplo ilustrativo, crearemos un proyecto con el nombre DesencadenadorPropiedad, y en el XAML de su ventana escribiremos el código del fuente 20.

```
<Window x:Class="Window1" …. >
 <Window.Resources>
  <Style x:Key="styCaja">
   <Style.Triggers>
    <Trigger Property="IsFocused" Value="True">
     <Setter Property=" TextBox.Background" Value=" LightCoral "/>
     <Setter Property="TextBox.FontWeight" Value="Bold"></Setter>
    </Trigger>
   </Style.Triggers>
  </Style>
 </Window.Resources>
```

```
 <Canvas>
  <TextBox Width="150"
    Canvas.Left="10"
    Canvas.Top="10"
    Text="prueba"/>

  <TextBox Width="200"
    Canvas.Left="10"
    Canvas.Top="40"
    Text="otro texto"
    Style="{StaticResource styCaja}"/>
 </Canvas>
</Window>
```

Fuente 20.

Al ejecutar el fuente 20, al segundo TextBox le será aplicado el estilo cuando obtenga el foco, ya que esta es la condición establecida en el desencadenador. La figura 10 muestra esta situación.

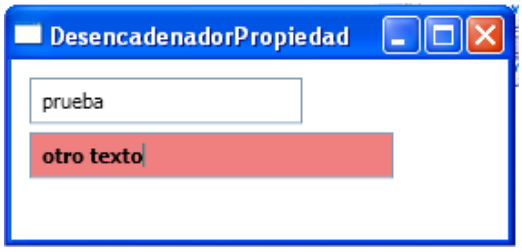

Figura 10.

Pero el uso de desencadenadores no está limitado a un único desencadenador por cada estilo, sino que podemos definir tantos como necesitemos. En el fuente 21 realizamos una pequeña variación del anterior ejemplo, de modo que ahora tendremos múltiples desencadenadores para un mismo estilo.

```
  <Style x:Key="styCaja">
   <Style.Triggers>
    <Trigger Property="IsFocused" Value="True">
     <Setter Property=" TextBox.Background" Value="LightCoral"/>
     <Setter Property="TextBox.FontWeight" Value="Bold"></Setter>
    </Trigger>

    <Trigger Property="TextBox.Text" Value="Camino">
     <Setter Property="TextBox.FontSize" Value="18"></Setter>
    </Trigger>
   </Style.Triggers>
  </Style>
```

Fuente 21.

Además de comprobar si el control tiene el foco, también se comprobará si el contenido del TextBox corresponde a una determinada cadena.

Desencadenador para datos

Este tipo de desencadenador se utiliza en combinación con una plantilla de datos, para aplicar uno o varios estilos a los componentes de dicha plantilla utilizados para visualizar los miembros de un objeto, pero en este caso cuando se cumpla una determinada condición.

Basándonos en el ejemplo de un apartado anterior sobre estilos aplicados a un DataTemplate, vamos a crear un proyecto con el nombre DesencadenadorDatos, en el que deberemos incluir el código de la clase Empleado, realizando sobre el XAML de la ventana algunos retoques, consistentes en crear dos estilos que incorporen un desencadenador por cada propiedad del objeto Empleado, como vemos en el fuente 22.

```
<?Mapping XmlNamespace="mio" ClrNamespace="DesencadenadorDatos" ?>

<Window x:Class="Window1" ....
 xmlns:mio="mio"  . . . . >

 <Window.Resources>
  <DataTemplate DataType="{x:Type mio:Empleado}">
   <StackPanel Orientation="Vertical" Background="LawnGreen">
    <Label Content="{Binding Path=Codigo}" Style="{StaticResource styEtiquetaPlant}"
HorizontalAlignment="Left" />
    <TextBox Text="{Binding Path=Nombre}" Style="{StaticResource styCajaPlant}" />
   </StackPanel>
  </DataTemplate>

  <Style x:Key="styEtiquetaPlant">
   <Style.Triggers>
    <DataTrigger Binding="{Binding Path=Codigo}" Value="275">
     <Setter Property="Control.Background" Value="Cyan" />
    </DataTrigger>
   </Style.Triggers>
  </Style>

  <Style x:Key="styCajaPlant">
   <Style.Triggers>
    <DataTrigger Binding="{Binding Path=Nombre}" Value="Elena">
     <Setter Property="Control.Background" Value="Crimson" />
    </DataTrigger>
```

```
    </Style.Triggers>
  </Style>
 </Window.Resources>

 <Canvas>
  <Label
    Name="lblEtiqueta1"
    BorderThickness="3"
    BorderBrush="Black"
    Background="Silver"
    Canvas.Left="10" Canvas.Top="10"
    Width="200" Height="100"/>

  <Label
    Name="lblEtiqueta2"
    BorderThickness="3"
    BorderBrush="Black"
    Background="Silver"
    Canvas.Left="10" Canvas.Top="125"
    Width="200" Height="100"/>

  <Button
    Name="btnAsignaEtiqueta1"
    Canvas.Left="225" Canvas.Top="10">
    Asignar etiqueta 1
  </Button>

  <Button
    Name="btnAsignaEtiqueta2"
    Canvas.Left="225" Canvas.Top="150">
    Asignar etiqueta 2
  </Button>
 </Canvas>
</Window>
```

Fuente 22.

Como podemos observar, en esta ocasión el estilo sólo será aplicado a los elementos del DataTemplate que cumplan con las condiciones establecidas en los DataTrigger. El fuente 23 muestra el código de los botones que asignan una instancia de la clase Empleado a los controles Label.

```
   Private Sub btnAsignaEtiqueta1_Click(ByVal sender As Object, ByVal e As
System.Windows.RoutedEventArgs) Handles btnAceptar.Click
      Dim oEmpleado1 As Empleado = New Empleado(275, "Elena")
      Me.lblEtiqueta.Content = oEmpleado1
   End Sub
```

```
    Private Sub btnAsignaEtiqueta2_Click(ByVal sender As Object, ByVal e As
System.Windows.RoutedEventArgs) Handles btnAceptar.Click
        Dim oEmpleado2 As Empleado = New Empleado(275, "Alberto")
        Me.lblEtiqueta.Content = oEmpleado2
    End Sub
```

Fuente 23.

Al ejecutar la aplicación, el objeto asignado al control lblEtiqueta1 cumple con las condiciones de ambos estilos, por lo tanto se aplica el estilo al completo en su DataTemplate. Para el control lblEtiqueta2 sólo se aplica el estilo correspondiente a la propiedad Codigo del objeto Empleado asignado al control; todo ello lo vemos en la figura 11.

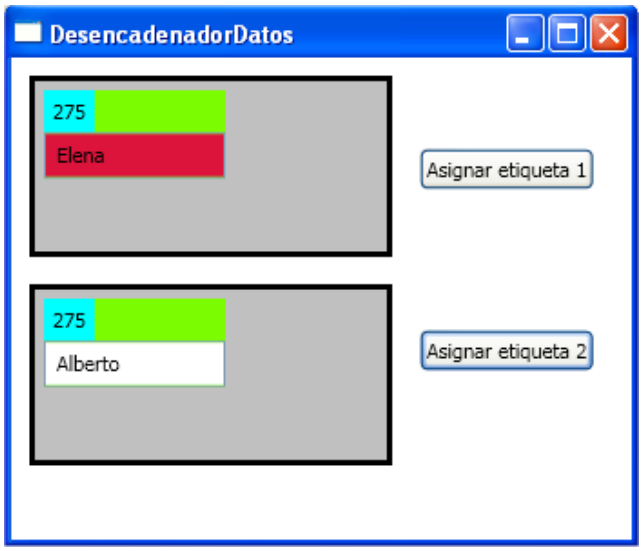

Figura 11.

Desencadenador para eventos

El tercer y último tipo de desencadenador se mantiene a la escucha de un determinado evento, y cuando este ocurre, genera una animación sobre el control al que se ha aplicado el estilo.

Para este ejemplo crearemos un proyecto con el nombre DesencadenadorEvento, en el que para crear la animación que responda al evento del desencadenador, definiremos una acción que contenga un objeto Storyboard.

Respecto a la animación, WPF dispone de una amplia variedad de tipos, en función de la propiedad del control que necesitemos animar. En el siguiente ejemplo, realizamos una animación sobre el ancho y alto de un control Button durante un espacio de tiempo de 5 y 10 segundos respectivamente. Dado que se trata de una propiedad

numérica de tipo Double, emplearemos el elemento DoubleAnimation, como muestra el fuente 24.

```
<Window x:Class="Window1"
 xmlns="http://schemas.microsoft.com/winfx/avalon/2005"
 xmlns:x="http://schemas.microsoft.com/winfx/xaml/2005"
 Title="DesencadenadorEvento">

 <Window.Resources>
  <Style TargetType="{x:Type Button}">
   <Setter Property="Background" Value="Green"></Setter>
   <Setter Property="Foreground" Value="White"></Setter>

   <Style.Triggers>
    <EventTrigger RoutedEvent="Click">
     <EventTrigger.Actions>
      <BeginStoryboard>
       <Storyboard>
        <DoubleAnimation Storyboard.TargetProperty="Width"
         To="250" Duration="0:0:5" />
        <DoubleAnimation Storyboard.TargetProperty="Height"
         To="100" Duration="0:0:10" AutoReverse="True"/>
       </Storyboard>
      </BeginStoryboard>
     </EventTrigger.Actions>
    </EventTrigger>
   </Style.Triggers>

  </Style>
 </Window.Resources>

 <Canvas>
  <Button
   Canvas.Left="10" Canvas.Top="10"
   Width="100" Height="30" Content="Prueba a pulsar" />
 </Canvas>
</Window>
```

Fuente 24.

Nótese en el fuente 24, que la animación para la altura del control volverá a dejarlo en su estado inicial, gracias al atributo AutoReverse, perteneciente al elemento DoubleAnimation.

Desencadenadores de condición múltiple

En algunos escenarios puede ser interesante que un desencadenador se ejecute cuando se cumpla más de una condición. Esto es factible utilizando las etiquetas MultiTrigger y Condition, pertenecientes a Style, las cuales nos permiten definir un desencadenador compuesto por varias condiciones, que sólo será aplicado si todas ellas se cumplen.

En el fuente 25 creamos un estilo de este tipo, en el que el control al que se asigne deberá cumplir tres condiciones para que dicho estilo se active.

```
<Window ….>
 <Window.Resources>
  <Style x:Key="styVariasCondiciones">
   <Style.Triggers>
    <MultiTrigger>
     <MultiTrigger.Conditions>
      <Condition Property="Text" Value="Impresora"></Condition>
      <Condition Property="IsMouseOver" Value="True"/>
      <Condition Property="Background" Value="Lavender"/>
     </MultiTrigger.Conditions>

     <Setter Property="Control.FontSize" Value="18"></Setter>
     <Setter Property="Control.Foreground" Value="Violet"></Setter>

    </MultiTrigger>
   </Style.Triggers>
  </Style>
 </Window.Resources>

 <Canvas>
  <TextBox
   Canvas.Left="10"
   Canvas.Top="10"
   Background="Lavender"
   Height="50"
   Width="150"
   Style="{StaticResource styVariasCondiciones}" />
 </Canvas>
</Window>
```

Fuente 25.

La figura 12 muestra esta ventana en ejecución, con el estilo activo sobre el control.

Figura 12.

Una variante de este desencadenador lo constituye el MultiDataTrigger, que como su nombre indica, se aplica a los desencadenadores de tipo DataTrigger, que muestran los miembros de un objeto, como vemos en el fuente 26.

```
<Style x:Key="styEtiqueta">
 <Style.Triggers>
  <MultiDataTrigger>
   <MultiDataTrigger.Conditions>
    <Condition Binding="{Binding Path=Codigo}" Value="895" />
    <Condition Binding="{Binding Path=Nombre}" Value="Vicente" />
   </MultiDataTrigger.Conditions>

   <Setter Property="Control.Background" Value="Violet" />

  </MultiDataTrigger>
 </Style.Triggers>
</Style>
```

Fuente 26.

En el fuente 26, el estilo será aplicado a un DataTemplate siempre que las propiedades del objeto Empleado cumplan las dos condiciones definidas en el MultiDataTrigger.

</Style> Cerrando estilo

Y llegamos al final de este artículo en el que nos hemos dedicado a repasar los principales aspectos de los estilos en XAML, el nuevo lenguaje de marcado para la creación de interfaces de usuario en WPF. Confiamos en haber despertado en el lector la curiosidad e interés por esta nueva e interesante tecnología, en la que a buen seguro nos veremos completamente inmersos en un plazo no muy largo de tiempo.