

**Apartado IV:  
Windows  
Presentation  
Foundation**  
por Luis Alfonso Rey



# Prólogo

## Apartado IV

*Para mi mujer y mis hijos, que son mi inspiración para hacer todos los días,  
lo que hago.*

Dado el carácter introductorio y no enciclopédico del libro, las siguientes páginas pretenden ser una presentación más o menos detallada de lo que es WPF pero no exhaustiva. Esto sitúa este libro en el ámbito de los manuales prácticos y no en el de los libros de referencia.

La segunda advertencia que hay que realizar es que todos los ejemplos del libro han sido realizados con la versión de Visual Studio 2010 Ultimate y esto puede presentar alguna discrepancia si se trata de seguir el libro con versiones diferentes, evidentemente si estas lo son en cuanto a la versión, en ocasiones estas discrepancias pueden ser más evidentes.

Dicho esto y antes de entrar en materia, vamos a explicar qué es WPF. Muchos de los lectores ya lo sabrán en mayor o menor medida, razón por la cual están leyendo este libro, pero para aquellos que no lo tengan claro empezaremos diciendo que *Windows Presentation Foundation* es una tecnología para crear

interfaces de aplicación, que trata de aunar los diferentes interfaces posibles (formularios, 3D y Web) mediante una sola API de programación.

La primera versión de WPF se lanza dentro del *Framework* 3.0 de .NET, pero en las sucesivas versiones 3.5, 3.5 SP 1 y 4.0 se van incluyendo novedades, no solo a nivel tecnológico sino también de producto, básicamente en Visual Studio y Blend, que nos permiten paulatinamente mejorar la experiencia de desarrollo, que inicialmente era muy pobre, hasta en convertirla en algo bastante sencillo, agradable y potente.

# El Modelo De Aplicación

Al empezar lo mejor es hacerlo por el principio, y este en Windows es hablar de qué tipo de aplicaciones se pueden crear con WPF.

Estas pueden ser básicamente dos, aplicaciones Windows y aplicaciones de Navegación.

## Aplicaciones de Windows

Las primeras son más convencionales y en cuanto al modelo de aplicación se puede decir que muy similares a sus primas de *WinForms*.

Las aplicaciones Windows está encapsuladas en una clase de tipo *Application*, esta clase es la responsable de crear y gestionar la vida de la aplicación, devolver los códigos de salida, procesar los parámetros de línea y en ocasiones gestionar la navegación.

La aplicación se ejecuta mediante el método *Run* que lanza el evento *Startup*. Durante la vida de la aplicación el usuario puede intercambiar la aplicación activa, la cual posee el foco, y esto puede provocar el lanzamiento de los eventos *Deactivated*, cuando se pierde el foco y *Activated*, cuando se vuelve a obtener.

Otro evento sumamente útil relacionado con la aplicación es *DispatcherUnhandledException*, dicho evento es lanzado antes de que una excepción no manejada sea transmitida al usuario causando con ello la inestabilidad de la aplicación.

El proceso de cierre es quizá el más complejo, ya que involucra el método *Shutdown* que cierra la aplicación, así como varios eventos como *Exit* que es lanzado cuando la aplicación trata de cerrarse, dándonos la oportunidad a través de la estructura *ExitEventArgs* y la propiedad *ApplicationExitCode* de devolver un código de salida. Además las aplicaciones se pueden cerrar por sucesos indirectos, como por ejemplo que la sesión de la máquina se cierre. Para este caso la aplicación nos provee de un evento *SessionEnding* que mediante la estructura *SessionEndingCancelEventArgs* nos permite cancelar el cierre de sesión programáticamente.

Para ilustrar lo anterior lo único que tenemos que hacer es crear un nuevo proyecto de tipo XAML application, visualizar la ventana de propiedades en Visual Studio, buscar y abrir el archivo App.xaml o Application.xaml y depositar el cursor sobre la tag de Xml <Application. Una vez hecho esto en la ventana de propiedades podemos observar las propiedades y eventos de la aplicación, precisamente en el evento *Exit* situaremos el siguiente código:

```
Visual Basic.NET
Private Sub Application_Exit(ByVal sender As
System.Object, ByVal e As
System.Windows.ExitEventArgs) Handles MyBase.Exit
    MessageBox.Show("Adi os")
End Sub

C#
private void Application_Exit(object sender,
ExitEventArgs e)
{
    MessageBox.Show("Adi os");
}
```

Tras la ejecución la aplicación mostrará un cartel cada vez que la cerremos.

## Aplicaciones de navegación

Las aplicaciones de navegación son un nuevo modelo de aplicaciones propuestas por WPF que pueden resultar muy interesantes. Se caracterizan fundamentalmente por el estilo que tienen de transitar entre su ventanas, o también llamadas páginas. El tránsito se produce de una manera muy similar a como se produce en las aplicaciones Web, mediante enlaces y métodos que nos remiten a una URI. Además estas aplicaciones también tienen un servicio de navegación que recuerda los “sitios” visitados y nos permite retornar a ellos, del mismo modo que en una aplicación Web.

Habitualmente estas aplicaciones se ejecutan dentro del navegador, usando en algunos casos los propios controles para manejar resortes de su mecanismo de navegación. Esto merece una explicación más detallada, que la aplicación se ejecute dentro del navegador no significa que sea una aplicación Web. De hecho no lo es en absoluto, es una aplicación Windows que se ejecuta alojada en el navegador y que necesita, como las aplicaciones Windows, que el *Framework* se encuentre instalado en la máquina. En esto difieren de las aplicaciones Silverlight, que se ejecutan dentro del ámbito del navegador pero no necesitan el *Framework* sino un *Plug-in* que hace las veces de ejecutor.

Además también se consideran aplicaciones de navegación aquellas aplicaciones Windows de *WPF* pero que hacen uso de la clase *PageFunction<t>* o *PageFunction(Of t)*.

Para ejemplificar estas aplicaciones podemos crear una nueva WPF Application y en ella agregar una nueva *Page Function*. Dentro de la etiqueta `<Grid>` podemos situar el siguiente código:

XAML

```
<TextBlock>Prueba de navegación</TextBlock>
```

Posteriormente en la ventana principal `MainWindow.xaml` depositar un nuevo botón desde la *ToolBox* y tras hacer doble clic sobre él, insertar en el cuerpo del gestor del evento asociado:

Visual Basic.NET

```
Private Sub Button1_Click(ByVal sender As
System.Object, ByVal e As
System.Windows.RoutedEventArgs) Handles Button1.Click
    Dim wizard As New NavigationWindow()
    wizard.Source = New Uri("PageFunction1.xaml",
UriKind.Relative)
    wizard.ShowDialog()
End Sub
```

```
C#
private void button1_Click(object sender,
RoutedEventArgs e)
{
    NavigationWindow wizard = new
NavigationWindow();
    wizard.Source = new Uri ("PageFunction1.xaml ",
UriKind.Relative);
    wizard.ShowDialog();
}
```

En el caso de las aplicaciones de navegación la clase *Application* se sustituye por su derivada *NavigationApplication* que nos provee de los servicios y miembros necesarios para articular una aplicación de navegación.

El primer miembro del que hablaremos será una propiedad, *StartupURI* que nos permite especificar la página de inicio de la aplicación. Para dar soporte a la posterior navegación la aplicación contiene un *NavigationService*. Este servicio no controla la página que actualmente se está mostrando sino el camino que se ha seguido para alcanzarla. Acceder a él es algo oscuro, se hace a través del método *GetNavigationService* que exige un parámetro de tipo *DependantObject*, que ahora es precipitado explicar pero, que para entenderlo rápidamente es un objeto de WPF. En ese sentido el mecanismo más fácil de satisfacer este parámetro es utilizando un formulario o un control dentro de este. Una vez obtenido este servicio él nos provee de métodos como *GoBack* o *GoForward* para movernos por el historial o *Navigate* que nos permite dar un salto incondicional a una página cualquiera de la aplicación.

De vuelta a la ***NavigationApplication*** no debemos olvidar que esta nos provee de ciertos eventos para controlar la navegación como *Navigating*, *Navigated*, *NavigationProgress*, *NavigationFailed*, *NavigationStopped* y *LoadCompleted*, todos ellos destinados a controlar los diferentes estados por los que pasa la aplicación tanto durante la navegación como en función del resultado de esta.

Para crear una de estas aplicaciones hemos de seleccionar una plantilla diferente, en concreto WPF Browser Application. Un código de ejemplo para la invocación de un método de navegación podría ser este:

```
Visual Basic .NET
NavigationService.GetNavigationService(Me).Navigate(New
Uri("Page1.xaml", UriKind.Relative))
```

```
C#
NavigationService.GetNavigationService(this).Navigate(new
Uri("Page1.xaml", UriKind.Relative));
```



## Conclusión

WPF nos ofrece diferentes modelos de aplicación, que nos permiten adaptarnos mejor a situaciones en las cuales un interfaz es mejor entendido, o presenta ventajas que nos permiten mejorar procedimientos como el despliegue o el control de la seguridad.

## A continuación

En el capítulo siguiente trataremos la especificación XAML, que nos servirá para que de manera declarativa podamos crear interfaces. Además trataremos algunos aspectos básicos de XML con el fin de facilitar la comprensión de aquel lector que esté menos familiarizado con ellos.



# XAML

En el epígrafe anterior hemos comprobado como una determinada característica diferencia sustancialmente WPF del resto de aplicaciones .NET, el XAML. Digo diferencia aunque existen otros tipos de aplicaciones que utilizan formulaciones similares. Sin embargo es innegable que a simple vista el XAML se aplica de manera única a este tipo de proyectos.

Pero, ¿qué es exactamente XAML?, pues un acrónimo de eXtensible Application Markup Language esto es, un XML para definir aplicaciones. Las define a modo de formato de serialización de objetos, lo que quiere decir que nosotros escribimos un XML que posteriormente será leído e interpretado de forma que cada elemento dentro de él se convertirá en un objeto dentro de la aplicación, y generalmente dentro de una ventana o página.

Este proceso puede sonar muy similar al que se produce con el HTML porque de hecho lo es. Específicamente se han buscado las ventajas de cada uno de los entornos de desarrollo para conformar WPF y en este caso la ventaja principal es la separación efectiva que nos permite este modelo, de forma que desarrollador y diseñador puedan trabajar sobre la misma ventana de manera separada y coincidente o no, en el tiempo.



Más en detalle un fichero XAML es un fichero XML y aunque casi todo el mundo comprende bien cómo funciona un fichero XML aquí vamos a invertir unas líneas para que posteriormente todo se entienda correctamente.

## XML

Fuera de explicaciones históricas, podemos decir que un fichero XML es un fichero de texto formado por marcas y que al menos ha de estar “bien formado”.

Por bien formado se entiende:

1. Que los documentos han de mantener una estructura estrictamente jerárquica.
2. Que los documentos han de tener un solo elemento raíz.
3. Que los atributos siempre deben marcar su contenido entre comillas.
4. Que es sensible a mayúsculas y minúsculas.

to

## Representación

En XAML tienen representación casi todas las clases relacionadas con WPF, y se usan por tanto para establecer que objetos van a componer una ventana, por ejemplo.

Dentro de un archivo XAML los elementos representan objetos mientras que los atributos propiedades. Esto sería por tanto un botón con una anchura de 30 y de color Rojo:

```
XAML
<Button Width="30" Background="Red" />
```

Sin embargo esta sintaxis también sería válida:

```
XAML
<Button>
    <Button.Background>
        Red
    </Button.Background>
</Button>
```

En este caso la propiedad está representada como un elemento anidado y no como un atributo. Esto es así para poder insertar además de contenidos simples en las propiedades, valores mucho más complejos:

```
XAML
<Button>
    <Button.Background>
        <LinearGradientBrush>
            <GradientStopCollection>
                <GradientStop Offset="0"
Color="White" />
                <GradientStop Offset="1"
Color="Red" />
            </GradientStopCollection>
        </LinearGradientBrush>
    </Button.Background>
</Button>
```

Además como se puede ver en ocasiones el nombre de la propiedad está precedido por el de la clase `Button.Background` y en otras no. `GradientStopCollection` generalmente quiere decir que la propiedad cuando va precedida es que es una propiedad de la clase u objeto, mientras que en el segundo es tan solo un contenido compatible con el exigido por el tipo de la propiedad.

Existen también otro tipo de propiedades bastante curiosas, echemos un vistazo al ejemplo:

```
XAML
<Grid>
    <Button x:Name="Button1" Grid.Column="0"/>
</Grid>
```

La propiedad *x:Name* en realidad es la propiedad *Name*, como cabría esperar, pero originalmente la especificación XAML establecía unas cuantas propiedades que se encontraban precedidas por *x:* (actualmente ya no es necesario escribir *x:*), estas propiedades generalmente estaban referidas con la meta-programación.

Cosa diferente es la propiedad *Grid.Column*, que para una persona que comience a usar esta tecnología puede resultar de lo más extraña. Se conocen por el nombre de *attached properties* o propiedades adjuntas, que básicamente permite incluir propiedades definidas en una clase, en este caso *Grid*, en otras, el botón. Con ello conseguimos que el botón reaccione contextualmente en este caso al situarse en una columna numerada con 0 y definida en el *Grid* que lo contiene.

## Sistema de propiedades y eventos

Conseguir el efecto de las *attached properties* así como otras muchas ventajas como el *binding*, los estilos, etc. Se debe principalmente a las conocidas como *Dependency Properties*. Las propiedades dependientes son propiedades que están respaldadas por lo que se conoce como *WPF Property System* y podrían definirse como, propiedades que se computan teniendo en cuenta elementos externos, como estilos, otras propiedades, la situación del control, etc. Así si un botón se encuentra dentro de un panel que modifica su posición o tamaño al aplicarle alguna lógica de distribución, las propiedades que tienen que ver con estas características del control se verán afectadas por el panel y su lógica de distribución.

Una propiedad dependiente tiene un aspecto como este:

```
Visual Basic .NET
Public Shared ReadOnly NombreProperty As
    DependencyProperty =
        DependencyProperty.Register("Nombre",
                                    GetType(Tipo))

Public Property Nombre() As Tipo
    Get
        Return CType(GetValue(NombreProperty), Tipo)
```

```

        End Get
        Set(ByVal value As Tipo)
            SetValue(NombreProperty, value)
        End Set
    End Property

```

```

C#
public static readonly DependencyProperty
NombreProperty =
    DependencyProperty.Register(
        "Nombre", typeof(Tipo),
        );
public bool Nombre
{
    get { return (Tipo)GetValue(NombreProperty); }
    set { SetValue(NombreProperty, value); }
}

```

Como se puede observar la propiedad se declara de manera similar, la diferencia estriba en que el campo de respaldo es muy extraño. De hecho siempre es *DependencyProperty*, este tipo es el tipo que permite que el valor de la propiedad esté respaldado en el sistema de propiedades. Luego en la propia definición hay que especificar el tipo y como se ve, establecer un nombre al cual asociarlo, de esta manera podemos tener más de un campo del mismo tipo.

Además el nombre del campo suele ser idéntico al nombre de la propiedad que lo usa, esto es una convención, más la fórmula *Property*.

Acceder al valor de este campo tampoco es sencillo, leerlo se logra mediante el método *GetValue*, que devuelve un objeto que luego deberemos de convertir al tipo deseado. Escribir por tanto se hace mediante otro método llamado *SetValue*. Pero, ¿de dónde salen estos métodos? Bien salen de la clase ***DependencyObject***, clase de la cual es obligatorio que descienda cualquier clase que pretenda tener propiedades dependientes.

Otra de las “rarezas” de WPF son los *Routed Events*. Si una propiedad dependiente nos permite propiedades que tienen en cuenta el entorno a la hora de calcularse, un evento enrutado es muy similar en concepto pero referido a los eventos. Sin embargo la razón es si cabe más fácil de entender.

```

XAML
<Grid>
    <Button x:Name="Button1" Grid.Column="0"/>
</Grid>

```

Pensemos por un momento una situación que es muy común en WPF y en otros sistemas de programación, un botón dentro de un panel. Comúnmente en cualquier sistema de programación al hacer clic sobre el botón no tenemos duda de que el evento a gestionar es el Click del botón. Esto es así porque estamos ante lo que se llama un evento directo, sólo salta en el receptor directo del “estímulo”.

Sin embargo esto no es así en los eventos enrutados. En estos el estímulo es propagado a lo largo de todos los controles de la jerarquía, o sea, en el ejemplo tanto en el botón como en el panel. Pero ¿para qué sirve esto? Esto permite articular una característica de WPF que es la capacidad de crear controles complejos anidando otros más simples, cosa que exploraremos más adelante.

De hecho y para complicarlo más, la estrategia de propagación no es para todos los eventos igual. En algunos casos ante el clic al botón el estímulo sería enviado primero al contenedor más general, el panel, denominando esta estrategia *Tunnel* y en ocasiones al contenedor más concreto, denominando en este caso *Bubble*.



Comúnmente los eventos de tipo *Tunnel* se nombran empezando por la palabra *Preview*. Los eventos enrutados también suelen ir en parejas, así al *PreviewMouseLeftButtonDown* le corresponde un *MouseLeftButtonDown*.

Cuando sobre un control se produce un estímulo, por ejemplo un clic, este es propagado primero siguiendo la estrategia *Tunnel*, para luego continuar con la *Bubble*. Todas ellas comparten la misma instancia de evento que se propaga a lo largo de todos los eventos.



Esta cadena se puede finalizar, dando el evento por gestionado, en algún punto tan solo estableciendo la propiedad *Handled* a True, propiedad localizada en la estructura EventArgs correspondiente del evento.

Visual Basic .NET

```
Public Shared ReadOnly EventoEvent As RoutedEvent =
   EventManager.RegisterRoutedEvent("Evento",
    RoutingStrategy.Bubble, GetType(RoutedEventHandler),
    GetType(TipoClase))

Public Custom Event Evento As RoutedEventHandler
    AddHandler(ByVal value As RoutedEventHandler)
        Me.AddHandler(EventoEvent, value)
    End AddHandler

    RemoveHandler(ByVal value As RoutedEventHandler)
        Me.RemoveHandler(EventoEvent, value)
    End RemoveHandler

    RaiseEvent(ByVal sender As Object, ByVal e As
    RoutedEventArgs)
        Me.RaiseEvent(e)
    End RaiseEvent
End Event
```

C#

```
public static readonly RoutedEvent EventoEvent =
    EventManager.RegisterRoutedEvent(
        "Evento", RoutingStrategy.Bubble,
        typeof(RoutedEventHandler), typeof(TipoClase));

public event RoutedEventHandler Evento
{
    add { AddHandler(EventoEvent, value); }
    remove { RemoveHandler(EventoEvent, value); }
}
```

Como se puede comprobar la analogía entre la declaración del evento y de la propiedad son muy similares, si acaso destacar que el tipo en este caso es *RoutedEvent*, que se incluye como último parámetro de la cláusula register, el tipo de la clase que contiene el evento y que los eventos para acceder al campo son *AddHandler* y *RemoveHandler*, como por otro lado cabría esperar.

## Controles y sus propiedades más comunes

Como la mayoría de los sistemas de programación, el número de controles ha crecido tanto que es prácticamente imposible ni tan siquiera mencionarlos, sin embargo sigue siendo necesario conocer algunos para poder comenzar. Aún a riesgo de convertir esto en una enumeración, vamos a mencionar algunos aquí y posteriormente dedicaremos algunas líneas a explicar propiedades y eventos comunes y situar al lector con un conocimiento inicial suficiente para comprender los ejemplos siguientes. Tómese si se quiere esta sección como una sección de posterior consulta.

<i>Layout</i>	<i>Los controles Layout se utilizan para controlar el tamaño, la posición y la disposición de los elementos hijos.</i>
<i>Border</i>	<i>Dibuja un borde, un fondo o ambos alrededor de otro elemento.</i>
<i>Canvas</i>	<i>Define un área donde posicionar elementos por coordenadas relativas al control canvas.</i>
<i>DockPanel</i>	<i>Define un área donde situar elementos horizontal o verticalmente, relativos a otros.</i>
<i>Grid</i>	<i>Define una rejilla donde situar los controles en función de filas y columnas.</i>
<i>GroupBox</i>	<i>Es un control que contiene otros controles definiendo para ello un borde y una cabecera.</i>
<i>ScrollViewer</i>	<i>Es un área capaz de hacer scroll con los elementos que contiene.</i>
<i>StackPanel</i>	<i>Distribuye los elementos apilándolos vertical y horizontalmente.</i>
<i>Viewbox</i>	<i>Define un área que puede escalar un elemento para encajar en un determinado espacio.</i>
<i>Window</i>	<i>Permite la creación, configuración y gestión de la vida de las ventanas y los diálogos.</i>
<i>WrapPanel</i>	<i>Sitúa los elementos en una sola línea de forma que cuando se alcanza el final de esta se produce una ruptura para ocupar la siguiente.</i>

<b>Botones</b>	<i>Los botones son uno de los elementos más básicos en el diseño de UI y nos permiten realizar acciones pulsando sobre ellos.</i>
<i>Button</i>	<i>Representa un botón que reacciona al clic del ratón.</i>
<i>RepeatButton</i>	<i>Representa un botón que emite periódicamente un clic desde que se presiona hasta el momento en que se libera.</i>

<b>Datos</b>	<i>Los controles de datos se usan para mostrar información desde un origen de datos.</i>
<i>DataGrid</i>	<i>Es un control que nos permite mostrar conjuntos de datos de manera tabular.</i>
<i>ListView</i>	<i>Es un control que nos permite mostrar elementos como una lista de datos.</i>
<i>TreeView</i>	<i>Muestra datos en una estructura jerárquica en donde los elementos se pueden desplegar o colapsar.</i>

<b>Fecha</b>	<i>Controles de datos para mostrar y seleccionar fechas.</i>
<i>Calendar</i>	<i>Es un control que te permite seleccionar una fecha mostrando un calendario.</i>
<i>DatePicker</i>	<i>Es un control que te permite seleccionar una fecha mostrando un combo que al ser desplegado muestra un calendario.</i>

<b>Menús</b>	<i>Los menús se usan para agrupar acciones o proveer de ayuda visual.</i>
<i>ContextMenu</i>	<i>Representa un menú que provee funcionalidad contextual a un determinado control.</i>
<i>Menu</i>	<i>Representa un menú que agrupa las acciones globales para toda una aplicación.</i>
<i>ToolBar</i>	<i>Provee de un contenedor para un grupo de comandos o controles.</i>

<b>Selección</b>	<i>Estos controles posibilitan que el usuario seleccione una o más opciones.</i>
<i>CheckBox</i>	<i>Representa un control que se puede seleccionar o no.</i>

<i>ComboBox</i>	<i>Es un control que permite seleccionar de una lista desplegable.</i>
<i>ListBox</i>	<i>Es un control que nos permite seleccionar uno o más elementos de una lista.</i>
<i>RadioButton</i>	<i>Representa un control que se puede seleccionar o no y que puede ser asociado con otro RadioButton para hacer selección exclusiva.</i>
<i>Slider</i>	<i>Representa un control que nos permite seleccionar un valor dentro de un rango utilizando una barra de desplazamiento.</i>

<i>Navegación</i>	<i>Mejoran o extienden la navegación de una aplicación.</i>
<i>Frame</i>	<i>Control que soporta navegación.</i>
<i>Hyperlink</i>	<i>Elemento que provee capacidades de almacenamiento de vínculos entre elementos navegables.</i>
<i>Page</i>	<i>Encapsula el contenido de una página que puede ser navegada por IE.</i>
<i>NavigationWindow</i>	<i>Representa una ventana que soporta navegación.</i>
<i>TabControl</i>	<i>Representa un control que contiene múltiples elementos que ocupan el mismo espacio en la pantalla.</i>

<i>Diálogo</i>	<i>Los diálogos encapsulan funcionalidad de configuración o selección como en el caso de la impresión.</i>
<i>OpenFileDialog</i>	<i>Diálogo para la selección de archivos para su posterior apertura.</i>
<i>PrintDialog</i>	<i>Diálogo para la configuración de la impresora.</i>
<i>SaveFileDialog</i>	<i>Diálogo para la selección de archivos para su posterior guardado.</i>

<i>Información de usuario</i>	<i>Proveen información o claridad a una aplicación. Normalmente no se puede interactuar con ellos.</i>
<i>AccessText</i>	<i>Permite especificar un carácter como tecla de acceso.</i>
<i>Label</i>	<i>Control que permite mostrar un texto y aporta una tecla de acceso.</i>

<i>Popup</i>	<i>Es una ventana de tipo “pop-up” (emergente).</i>
<i>ProgressBar</i>	<i>Indica el progreso de una operación.</i>
<i>StatusBar</i>	<i>Muestra información en una barra horizontal en una ventana.</i>
<i>TextBlock</i>	<i>Control muy ligero para mostrar texto.</i>
<i>ToolTip</i>	<i>Crea una ventana emergente que muestra información sobre un elemento .</i>

<b><i>Documentos</i></b>	<b><i>Controles especializados para ver documentos, optimizando procesos de lectura-</i></b>
<i>DocumentViewer</i>	<i>Sirve para mostrar documentos fijos por páginas.</i>
<i>FlowDocumentPageViewer</i>	<i>Sirve para mostrar documentos de flujo mostrando una página por vez.</i>
<i>FlowDocumentReader</i>	<i>Sirve para mostrar documentos de flujo en un control con varios mecanismos de muestra incorporados.</i>
<i>FlowDocumentScrollViewer</i>	<i>Sirve para mostrar documentos de flujo en modo de scroll continuo.</i>
<i>StickyNoteControl</i>	<i>Control que permite a los usuarios adjuntar texto o anotaciones manuscritas a los documentos.</i>

<b><i>Input</i></b>	<b><i>Controles para la inserción de texto u otro tipo de datos.</i></b>
<i>TextBox</i>	<i>Control para la inserción de texto plano, sin formato.</i>
<i>RichTextBox</i>	<i>Control para la inserción de documentos de texto enriquecido.</i>
<i>PasswordBox</i>	<i>Control para la inserción de contraseñas, con los caracteres enmascarados.</i>

<b><i>Medios</i></b>	<b><i>Controles con Soporte para video, audio y los formatos de imagen más populares.</i></b>
<i>Image</i>	<i>Control para mostrar imágenes.</i>

<i>MediaElement</i>	<i>Elemento destinado a reproducir video y audio, local o con soporte para streaming.</i>
<i>SoundPlayerAction</i>	<i>Es una TriggerAction que puede reproducir ficheros wav.</i>

Algunos de estos controles serán tratados posteriormente con más detalle como los encargados de gestionar el Layout o los documentos, otros serán reiteradamente revisitados como los botones o los elementos de Input. Casi todos ellos tienen una serie de propiedades características de las cuales depende su funcionamiento y otras más generales con el tamaño, márgenes, etc. Como por ejemplo estas:

<i>Opacity</i>	<i>Indica el nivel de transparencia de un control.</i>
<i>Cursor</i>	<i>Indica, de todos lo disponibles, el icono de cursor que se verá cuando el cursor del ratón esté sobre el control.</i>
<i>Visibility</i>	<i>Indica si un control es visible, invisible o está colapsado.</i>
<i>Width, Height</i>	<i>Dimensiones de control.</i>
<i>MinWidth, MaxWidth, MinHeight, MaxHeight</i>	<i>Dimensiones máximas y mínimas que ocupa un control. Habitualmente usadas para salvaguardar el Layout de la aplicación.</i>
<i>Margin</i>	<i>Espacio externo de separación del control con los controles circundantes.</i>
<i>Padding</i>	<i>Espacio interno de separación del control con su contenido.</i>
<i>UseLayoutRounding</i>	<i>Establece cuando un control y su contenido, aplicará lógica de redondeo en el dibujo de los vértices.</i>

Por otro lado además de propiedades existen también eventos comunes a la mayoría de los controles, como por ejemplo:

<i>GotMouseCapture</i>	<i>Bubble</i>	<i>Se produce cuando se captura el ratón.</i>
<i>LostMouseCapture</i>	<i>Bubble</i>	<i>Se produce cuando se pierde el ratón.</i>
<i>MouseLeftButtonUp</i>	<i>Bubble</i>	<i>Se produce cuando se suelta el botón izquierdo del ratón.</i>

<i>MouseLeftButtonDown</i>	<i>Bubble</i>	<i>Se produce cuando se aprieta el botón izquierdo del ratón.</i>
<i>MouseRightButtonUp</i>	<i>Bubble</i>	<i>Se produce cuando se suelta el botón derecho del ratón.</i>
<i>MouseRightButtonDown</i>	<i>Bubble</i>	<i>Se produce cuando se aprieta el botón derecho del ratón.</i>
<i>MouseMove</i>	<i>Bubble</i>	<i>Se produce cuando se mueve el ratón.</i>
<i>MouseWheel</i>	<i>Bubble</i>	<i>Se produce cuando se mueve la rueda del ratón.</i>
<i>PreviewMouseLeftButtonUp</i>	<i>Tunnel</i>	<i>Se produce cuando se suelta el botón izquierdo del ratón.</i>
<i>PreviewMouseLeftButtonDown</i>	<i>Tunnel</i>	<i>Se produce cuando se aprieta el botón izquierdo del ratón.</i>
<i>PreviewMouseRightButtonUp</i>	<i>Tunnel</i>	<i>Se produce cuando se suelta el botón derecho del ratón.</i>
<i>PreviewMouseRightButtonDown</i>	<i>Tunnel</i>	<i>Se produce cuando se aprieta el botón derecho del ratón.</i>
<i>PreviewMouseMove</i>	<i>Tunnel</i>	<i>Se produce cuando se mueve el ratón.</i>
<i>PreviewMouseWheel</i>	<i>Tunnel</i>	<i>Se produce cuando se mueve la rueda del ratón.</i>
<i>GotFocus</i>	<i>Bubble</i>	<i>Se produce cuando se captura el cursor.</i>
<i>LostFocus</i>	<i>Bubble</i>	<i>Se produce cuando se pierde el cursor.</i>
<i>KeyUp</i>	<i>Bubble</i>	<i>Se produce cuando se suelta una tecla del teclado.</i>
<i>KeyDown</i>	<i>Bubble</i>	<i>Se produce cuando se aprieta una tecla del teclado.</i>
<i>TextInput</i>	<i>Bubble</i>	<i>Se produce cuando se obtiene texto.</i>
<i>PreviewKeyUp</i>	<i>Tunnel</i>	<i>Se produce cuando se suelta una tecla del teclado.</i>
<i>PreviewKeyDown</i>	<i>Tunnel</i>	<i>Se produce cuando se aprieta una tecla del teclado.</i>
<i>PreviewTextInput</i>	<i>Tunnel</i>	<i>Se produce cuando se obtiene texto.</i>

## Conclusión

WPF logra una separación efectiva entre desarrolladores y diseñadores lo que permite que ambos trabajen al mismo tiempo sin colisionar. Además nos provee de propiedades dependientes o eventos enrutados. Todas estas características conforman una API de desarrollo consistente y que además de ofrecernos numerosas aportaciones tecnológicas nos añade un mundo nuevo en cuanto al desarrollo en equipo.

## A continuación

Los diferentes dispositivos a los que se puede aplicar WPF producen que la adaptación a las diferentes pantallas sea crítica. Es por esa razón entre otras, que necesitamos un potente sistema de gestión y racionalización de los espacios. En el siguiente capítulo veremos en que consiste.



# Paneles Y *Layout*

En WPF el diseño y la distribución de los elementos son, si cabe, más importantes que otros sistemas de programación. El principal motivo es que tanto el diseño como la distribución de los elementos son uno de los principales motivos por los que realizamos proyectos en WPF.

En estas líneas no vamos a adentrarnos en estos conceptos como tal, diseño y distribución de los controles, ya que quedan fuera del ámbito de este libro pero sí que intentaremos sentar unas bases de conocimiento para que las experiencias anteriores con otros sistemas de programación se puedan reutilizar con WPF.

El primer concepto que tenemos que tener claro es que cada elemento en WPF ocupa un rectángulo de espacio al ser mostrado. El tamaño de este rectángulo se computa a través de una suma de factores, propiedades de diseño, espacio disponible, panel contenedor, etc.

El tamaño de cualquier objeto se puede recuperar a través de clase `LayoutInformation` y su método `GetLayoutSlot`.

WPF entiende el Layout como un proceso recursivo en el que todos los elementos se posicionan, dimensionan y dibujan. Este sistema se encarga de realizar este a lo largo de todos los elementos y sus contenidos, completando así la presentación.

La funcionalidad antes descrita es de especial complejidad en los paneles, ya que estos a diferencia del resto de componentes suelen contener multitud de hijos, elementos contenidos en su interior, que distribuir.

Para eso los elementos y en especial los paneles proveen de dos métodos *MeasureOverride* y *ArrangeOverride* que nos permiten realizar los cálculos oportunos. El proceso vendría a ser así:

1. Un elemento comienza midiendo sus propiedades básicas.
2. Después se evalúan las propiedades heredadas de *FrameworkElement*, como *Width*, *Height* y *Margin*.
3. Se aplica la lógica concreta de *Panel*.
4. El contenido se organiza después de medir todos los elementos secundarios.
5. Se dibujan los elementos contenidos.

## Paneles

Los elementos más importantes dentro del desarrollo de un diseño en WPF son los paneles. Estos nos proporcionan diferentes lógicas de distribución que correctamente aplicadas, pueden simplificar mucho la confección de interfaces. Es por esta razón que un estudio de las diferentes opciones es sumamente útil.

El primero de los paneles a estudiar es el *Canvas*. Este panel se caracteriza por permitirnos distribuir libremente a lo largo y ancho del espacio que ocupa los diferentes elementos contenidos, haciendo uso de las propiedades asociadas *Canvas.Top*, *Canvas.Left*, *Canvas.Right* y *Canvas.Bottom*.



XAML

```
<Canvas>
    <Button x: Name="Button1" Canvas.Top="10"
Canvas.Left="10" />
</Canvas>
```

El *DockPanel* es un panel en donde los elementos se pueden distribuir bien asociados a lo largo de cualquiera de los lados del panel, bien ocupando todo el espacio libre restante.

Cuenta con una propiedad asociada *DockPanel.Dock*, que le indica a cada elemento como se ha de distribuir y una propiedad dependiente *LastFillChild*, que indica que se debe hacer con el último elemento insertado en el panel, si ocupar con él todo el espacio disponible o no.



XAML

```
<DockPanel >
    <Button x: Name="Button1"
DockPanel.Dock="Top" />
</DockPanel >
```

El *Grid* es un panel con la capacidad de dividir su espacio en filas y columnas, situando en las celdas resultantes los elementos.

Para definir la filas y las columnas tenemos las propiedades *RowDefinitions* y *ColumnDefinitions*, formuladas como sendas colecciones de *RowDefinition* y *ColumnDefinition* respectivamente. En una fila el dato más importante a definir es la altura así *RowDefinition* cuenta con una propiedad *Height*, mientras que las *ColumnDefinition* tienen la propiedad *Width*. En ambos casos la propiedad puede ser especificada o no. En este último caso se considerará que la dimensión especificada es *Auto*, esto es autodimensionada, dependiente del tamaño del contenido. También, y además de introducir un entero correspondiente a la medida deseada, podemos especificar \* (asterisco). En este caso la dimensión de la fila o columna se computará como una parte proporcional del espacio disponible, de esta forma si tenemos dos columnas de anchura \* en un *Grid*, el asterisco equivaldrá a la mitad y si son tres a un tercio.



La cosa se puede complicar con los multiplicadores, que son números enteros que pueden aparecer antes de los asteriscos. Cuando lo hacen no sólo multiplican el tamaño sino que aumentan el número de partes en que dividir el espacio. De esa manera un *Grid* con dos columnas, la primera con dos

asteriscos y la segunda con un asterisco computarán, la primera como dos tercios y la segunda como un tercio.

A la hora de situar por tanto un elemento hemos de decirle que fila y que columna, o si no asumirá el valor por defecto en ellas, o sea 0. Para esto utilizaremos las propiedades asociadas *Grid.Column* y *Grid.Row*. Además podemos decirle a un elemento que se expanda por más de una columna o fila utilizando las propiedades *Grid.ColumnSpan* y *Grid.RowSpan*.

XAML

```
<Grid>
  <RowDefinitions>
    <RowDefinition />
    <RowDefinition />
  </RowDefinitions>
  <ColumnDefinitions>
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="*" />
  </ColumnDefinitions>
  <Button x:Name="Button1" Grid.Column="0"
    Grid.ColumnSpan="2" Grid.Row="0" />
</Grid>
```

El *StackPanel* es un panel que permite apilar los elementos contenidos con una orientación. Para establecer esta simplemente hay que especificar su propiedad *Orientation* a horizontal o vertical.



XAML

```
<StackPanel Orientation="Vertical">
  <Button x:Name="Button1" />
  <Button x:Name="Button2" />
</StackPanel>
```

Por último el *WrapPanel* es un panel que dispone los elementos como si de escritura se tratara, de izquierda a derecha. Cuando se alcanza el límite por la derecha el panel parte la línea de elementos e inaugura una nueva línea. También se puede cambiar la orientación con la propiedad *Orientation*.



XAML

```
<WrapPanel Orientation="Horizontal">
  <Button x:Name="Button1" />
  <Button x:Name="Button2" />
  <Button x:Name="Button3" />
</WrapPanel>
```

## Conclusión

La distribución de los elementos dentro de una aplicación es una de las tareas que tiene más importancia, ya que de ella pueden depender desde la usabilidad hasta la comprensión que de ella hagan los usuarios. Con WPF contamos con un conjunto de paneles que nos permiten crear "Layouts" regulares sin necesidad de crear complejos diseños, tan sólo utilizando estos paneles y la capacidad anidatoria de WPF.

## A continuación

Los datos son lo más importante de una aplicación, y la comunicación de estos al usuario la misión principal de un interfaz. WPF como cualquier otra API de interfaz posee un mecanismo para hacerlo automáticamente. A continuación veremos este mecanismo.



# Databinding y Recursos

El *Databinding* es un proceso por el cual dos objetos sincronizan una de sus propiedades entre si. Por ejemplo si en una aplicación tenemos un objeto de clase Persona la propiedad Nombre puede estar sincronizada con la propiedad Text de un Textbox de manera automática.

Esto es muy conveniente porque cualquier cambio que se produzca tanto en el nombre como a través del Textbox será, inmediatamente comunicado evitándonos así el enojoso proceso de detectarlo nosotros, así como la correspondiente instrucción de asignación.

Y esto no es una tontería, ya que en un formulario medio el número de estas asignaciones puede llegar a ser bastante grande, complicando el código y evidentemente su mantenimiento.

## Los interfaces

El cómo de este proceso involucra varios mecanismos así como diferentes tipos de objetos que pueden verse involucrados desde el punto de vista de WPF.

Esto último es importante y cabe resaltarlo de manera especial, lo que se expondrá a continuación es solo válido para WPF, otros mecanismos tanto en .NET como fuera de la plataforma funcionan de manera distinta.

Para una rápida comprensión seguiremos con el ejemplo planteado en la introducción.

En este ejemplo podemos detectar dos procesos:

1. Si el Textbox es modificado ha de comunicar al objeto el cambio.
2. Si el Nombre es modificado ha de comunicar al Textbox el cambio.

El primer proceso se lleva a cabo de manera automática por ser la propiedad `Text` una propiedad dependiente. Esto es muy importante ya que aquellas propiedades que no sean dependientes, aunque se encuentren en una clase de tipo *DependencyObject* caerán dentro del segundo caso. Por tanto si queremos crear un control WPF que soporte *Databinding* en sus propiedades basta con que estas sean dependientes.

El segundo caso está diseñado para que clases comunes soporten enlaces con controles WPF. El mecanismo de sincronización es simple. Dotar a estas clases de un evento que salte cada vez que se produzca una modificación en las propiedades. Para que esto funcione hemos de cumplir algunas normas:

1. Que el evento sea conocido por los controles, por lo que la manera más eficaz es enmarcarlo dentro de un interfaz. Este es *INotifyPropertyChanged*, interfaz que posee un solo miembro *PropertyChanged*, evento diseñado para saltar cuando se produce una modificación en los datos de una propiedad.
2. Que este interfaz sea implementado por la clase de datos, en el ejemplo *persona*. Esta tarea es sumamente fácil porque Visual Studio tiene procesos automatizados para hacerlo automáticamente y al sólo contener un evento no es necesario dotar de código a la implementación.
3. Elevar correctamente el evento en la clase de datos. Esto es tan sencillo como pensar ¿dónde se modifica la propiedad *Nombre*? Pues como todas las propiedades, en el método *set* de su implementación, ahí debe ser elevado el evento. Esta es la razón por la que el binding sólo se puede aplicar a propiedades, porque son las que poseen un método *set*.



C#

```

class Persona: INotifyPropertyChanged
{
    string nombre;

    public string Nombre
    {
        get { return nombre; }
        set {
            nombre = value;
            OnPropertyChanged("Nombre");
        }
    }

    public event PropertyChangedEventHandler
    PropertyChanged;

    private void OnPropertyChanged (string
    property)
    {
        if (PropertyChanged != null)
            PropertyChanged(this, new
            PropertyChangedEventArgs(property));
    }
}

```

Visual Basic.NET

```

Class Persona
Implements INotifyPropertyChanged
Private m_nombre As String

Public Property Nombre() As String
Get
    Return m_nombre
End Get
Set
    m_nombre = value
    OnPropertyChanged("Nombre")
End Set
End Property

Public Event PropertyChanged As
PropertyChangedEventHandler

Private Sub OnPropertyChanged([property] As
String)
    RaiseEvent PropertyChanged(Me, New
    PropertyChangedEventArgs([property]))
End Sub

End Class

```

El lector se habrá dado cuenta de que este enlace se establece entre una clase y un control únicos, es lo que se conoce como *Databinding* simple. Pero existe también el *Databinding múltiple*, entre una lista de personas y un Listbox, por ejemplo. En este caso no sólo el contenido de la propiedad ha de ser tenido en cuenta. En control necesita se “avisado” también cuando cambie la lista, por ejemplo cuando se añadan o quiten personas.

Es por esto que existe el interfaz *INotifyCollectionChanged* que funciona de manera análoga pero para las colecciones. Sin embargo no analizaremos en detalle este interfaz, porque se antoja bastante menos útil.

De un lado existen colecciones como *ObservableCollection<t>* u *ObservableCollection(Of t)* que ya lo tienen implementado, por otro DataSets, tablas de entidades y ObjectQuerys también lo implementan. De forma que no es probable que el lector necesite implementarlo, sirvan estas líneas como “prueba de existencia” de este interfaz.

## Las expresiones de binding

Si bien antes se plantearon las condiciones que habían de cumplir los objetos que pueden participar en un *Databinding*, el hecho de poder no implica participar. Es decir que tener una clase persona y un control no significa que hayan de estar relacionados. Esta relación se establece mediante las expresiones de binding.

Antes de avanzar en el conocimiento de estas expresiones hemos de decir que las expresiones de *Databinding* se pueden expresar de manera imperativa, es decir por código.

C#

```
Persona p = new Persona();
Binding binding = new Binding("Nombre");
binding.Source = p;
binding.Mode = BindingMode.OneWay;
myTextBox.SetBinding(TextBox.TextProperty, binding);
```

Visual Basic.NET

```
Dim p As New Persona()
Dim binding As New Binding(Nombre)
binding.Source = p
binding.Mode = BindingMode.OneWay
myTextBox.SetBinding(TextBox.TextProperty, binding)
```

Este código establece la relación entre las dos clases y es bastante fácil de entender, si bien la tercera línea merece una explicación.

Esta línea establece el modo de enlace, que nos permite establecer tres tipos de relaciones:

1. *TwoWay*, el control y el objeto pueden ser el origen de una modificación y se sincronizan mutuamente.
2. *OneWay*, sólo el objeto puede modificarse y sincronizar, el control perderá cualquier cambio que se realice sobre él.
3. *OneTime*, el objeto se sincronizará con el control una sola vez, comúnmente en el momento en que el control es mostrado.
4. *OneWayToSource*, sólo el control puede sincronizarse, sería la inversa de *OneWay*.
5. *Default*, se establece en función del control, si este es bidireccional como el *Textbox* será *TwoWay* y si es una *Label* *OneWay*.

De cualquier manera el código anterior no es muy conveniente, porque establecer los enlaces en código los hace menos accesibles dentro del “Mare Magnum” de código de la aplicación, incrementando la dificultad además.

Es por esto que se utilizan las expresiones. Estas no son más que un pedazo de texto con una sintaxis que se insertan en XAML.

La sintaxis es bien simple, todas las expresiones se enmarcan entre dos llaves, llevan la palabra *Binding* y a continuación y de manera opcional una serie de pares nombre\valor separados por comas, para establecer la información del enlace:

XAML

```
<TextBox Text="{Binding Source=P, Path=Nombre, Mode=OneWay}" />
```

```
<TextBox Text="{Binding Path=Nombre, Mode=OneWay}" />
```

```
<TextBox Text="{Binding Path=Nombre}" />
```

```
<ListBox ItemsSource="{Binding}" />
```

El número de especificaciones que se pueden hacer es numerosa e incluso esta última, sin prácticamente información es posible. En este punto nos vamos a ocupar de las más importantes.

1. *Source*, indica el objeto origen de datos por su nombre, si el objeto es un control se puede sustituir por *ElementName*.
2. *Path*, indica la propiedad, el nombre proviene de que puede ser una propiedad simple o proveniente de navegación, o sea, Nombre o Dirección.Calle.
3. *Mode*, nos permite establecer el modo del binding.

Hemos de reseñar también que la primera de las líneas es incorrecta, y sólo está mencionada a efectos de ejemplo. La razón es que no es posible hacer referencia a una variable definida en código desde el XAML.

El proceso de conectar código con XAML sería bastante incómodo de no ser porque existen dos soluciones, por un lado la incorporación de un espacio de nombres como un espacio de nombres XML, esta solución será tratada más adelante. La segunda es el *DataContext* y será tratada en el siguiente punto.

## DataContext

La propiedad *DataContext* es una propiedad definida en la clase *FrameworkElement*. Esta clase es el antecesor primero de todas las clases de WPF. Esto quiere decir que cualquier elemento puede disponer de ella, pero ¿por qué es tan importante?

Podríamos decir que la propiedad *DataContext* es un origen de datos por defecto, es decir que en caso de que en una expresión de binding no establezcamos un *Source*, el *DataContext* actuará en su lugar.

La ventaja principal de usar el *DataContext* es que este es parte del interfaz principal de la clase al contrario del *Source*. Dicho en cristiano, al ser *DataContext* una propiedad de la clase se puede asignar desde código, mientras que *Source* es una propiedad de la expresión que cuando menos es mucho más complicado de asignar.

En resumen el código anterior quedaría así:

```
C#
    Persona p = new Persona();
    TextBox1.DataContext = p;
```

```
Visual Basic.NET
    Dim p As New Persona()
    TextBox1.DataContext = p
```

```
XAML
<TextBox x:Name="TextBox1" Text="{Binding
Path=Nombre}" />
```

Con esto conseguiríamos sincronizar la propiedad `Text` del `TextBox` con la propiedad `Nombre` de la variable `p`. Una cosa importante se desprende de la instrucción de asignación del `DataContext`, a ella accedemos mediante el nombre del `TextBox` y para que esto sea posible el `TextBox` ha de tener nombre, es por eso que en la instrucción XAML hemos tenido que especificar la propiedad `x:Name`. Si no, no hubiera sido posible acceder a ella.

Otra cosa importante en cuanto al `DataContext` es que establece un ámbito de enlace, esto significa que si establecemos el `DataContext` de un panel que contiene un `TextBox`, este compartirá `DataContext` con el panel por el mero hecho de estar dentro de él. A menos que establezcamos también el `DataContext` del `TextBox`. Según esto estas líneas producirían una salida similar.

```
C#
Persona p = new Persona();
Grid1.DataContext = p;
```

```
Visual Basic.NET
Dim p As New Persona()
Grid1.DataContext = p
```

```
XAML
<Grid x:Name="Grid1">
  <TextBox Text="{Binding Path=Nombre}" />
</Grid>
```

## El binding múltiple

Hasta ahora nos hemos ocupado del binding simple, pero ya es hora de ocuparnos de múltiple.

Este es un binding que se establece entre una colección, lista, tabla, etc. y un control como el *ListBox*, el *ComboBox* o el *DataGrid*.

Estos controles poseen la capacidad de mostrar un conjunto de datos y no sólo uno.

Todos estos controles poseen una propiedad común que les permite funcionar, la propiedad *ItemsSource*. Esta propiedad se encarga de establecer la colección que sirve de origen para los datos.

También es posible establecer la propiedad *ItemsSource* mediante una expresión, en concreto la expresión *{Binding}*. Esta expresión establecida en la propiedad informa que el *DataContext* establecido más próximo se asigna sin restricciones como *ItemsSource*.

Evidentemente el *DataContext* ha de contener una colección, de no ser así el enlace no funcionaría.

Otra propiedad que tienen los controles de binding múltiple es *IsSynchronizedWithCurrentItem*. Esta propiedad booleana es muy importante ya que informa al control de cual es su entorno de binding. Para entender esto debemos entender como funciona el binding de las colecciones.

Cuando enlazamos una colección con una serie de elementos de binding simple, estos crean una clase de intermediación que necesariamente implementa el interfaz *ICollectionView*. Se utiliza esta clase de intermediación a fin de soportar el ordenamiento y el filtrado sin necesidad de modificar la colección origen.

De cualquier modo, todos los controles de binding simple comparten *View*. No es así con los de binding múltiple, cada uno de ellos inaugura un nuevo *View* para su uso exclusivo. La consecuencia principal de esto es que si tenemos un *TextBox* mostrando el Nombre de la persona actual y al mismo tiempo una *ListBox* mostrando la lista completa de las personas, ambos elementos no estarán sincronizados, o sea cambiando el elemento seleccionado en el *ListBox* no cambiará el nombre que se muestra en el *TextBox*.

Sin embargo si establecemos la propiedad *IsSynchronizedWithCurrentItem* a *True* esta sincronización si tendrá lugar. Esto es así porque el *ListBox* se sumará al *View* que utilizan los *TextBoxes*.

Finalmente nos resta revisar algunos de los controles de binding múltiple. No es posible hacerlo con todos porque sería un trabajo muy largo y no aportaría demasiado así que nos centraremos en un par de ellos, el *ListBox* y el *DataGrid*.

El *ListBox* es capaz automáticamente de mostrar una colección de cadenas, sin embargo cuando la colección es de objetos esto no es tan simple.

Para poder mostrar estos elementos caben dos estrategias, la plantilla de datos, que revisaremos en el capítulo dedicado a las plantillas o mediante las propiedades *DisplayMemberPath* que nos permite establecer la propiedad o ruta del dato a mostrar y *SelectedValuePath* que nos permite establecer la

propiedad o ruta del valor a seleccionar. Por ejemplo, `DisplayMemberPath` puede ser el nombre del país y el `SelectedValuePath` código, comúnmente más útil a la hora de procesar.

```
C#
LeftListBox.ItemsSource = LoadListBoxData();
private ArrayList LoadListBoxData()
{
    ArrayList itemList = new ArrayList();
    itemList.Add("Coffee");
    itemList.Add("Tea");
    itemList.Add("Orange Juice");
    itemList.Add("Milk");
    itemList.Add("Mango Shake");
    itemList.Add("Iced Tea");
    itemList.Add("Soda");
    itemList.Add("Water");
    return itemList;
}

Visual Basic.NET
LeftListBox.ItemsSource = LoadListBoxData()
Private Function LoadListBoxData() As ArrayList
    Dim itemList As New ArrayList()
    itemList.Add("Coffee")
    itemList.Add("Tea")
    itemList.Add("Orange Juice")
    itemList.Add("Milk")
    itemList.Add("Mango Shake")
    itemList.Add("Iced Tea")
    itemList.Add("Soda")
    itemList.Add("Water")
    Return itemList
End Function

XAML
<ListBox Margin="11, 13, 355, 11" Name="LeftListBox" />
```

El *DataGrid* es un control que muestra datos en celdas generadas por la intersección de filas y columnas. El número de filas depende directamente de la colección de datos, son las columnas por tanto las que nos permiten especificar que datos mostrar de cada uno de los elementos.

La propiedad *AutoGeneratedColumns* crea las columnas de manera automática. Pero si queremos restringir los datos a mostrar podemos desactivar esta propiedad y agregar estas columnas de manera manual a través de la propiedad de colección *Columns*. Estas columnas poseen una propiedad *Binding* para introducir una expresión de binding. Además existen varios tipos de columnas, de texto, `CheckBox`, ... y de plantillas que también nos permitirán introducir plantillas de datos como los `ListBoxes`.

XAML

```

<DataGrid ItemsSource="{Binding}">
    <DataGrid.Columns>
        <DataGridTextColumn x:Name="IDColumn"
            Binding="{Binding Path=ID}" Header="ID" />
        <DataGridTextColumn
            x:Name="nombreColumn" Binding="{Binding Path=Nombre}"
            Header="Nombre" />
        <DataGridTextColumn
            x:Name="apellidosColumn" Binding="{Binding
            Path=Apellidos}" Header="Apellidos" />
    </DataGrid.Columns>
</DataGrid>

```

## Conversores

Hasta ahora hemos estado jugando con ventaja cuando nos referimos al binding. Básicamente y de manera implícita hemos estado enlazando propiedades compatibles, es decir, que comparten tipo o cuyos tipos son compatibles, pero esto no tiene por que ser así.

En ocasiones cuando dos propiedades se relacionan por binding los tipos de ambas no son idénticos ni compatibles por lo que necesitamos un *conversor*.

Un conversor no es ni más ni menos que una clase que implementa un interfaz, *IValueConverter*, y que es utilizado como intermediador de las operaciones de binding, antes de que el contenido de una propiedad se sincronice con otra se pasa por el conversor a fin de que los datos de ambas propiedades sean compatibles.

C#

```

public enum AgeGroups
{
    Young = 18,
    Mature = 60,
    Old = 150
}

[ValueConversion(typeof(int), typeof(string))]
public class AgeConverter : IValueConverter
{
    public object Convert(object value, System.Type
        targetType, object parameter,
        System.Globalization.CultureInfo culture)
    {

```



```

        dynamic number =
System.Convert.ToInt32(value);
        if (number < AgeGroups.Young) {
            return AgeGroups.Young.ToString();
        } else if (number < AgeGroups.Mature) {
            return AgeGroups.Mature.ToString();
        } else {
            return AgeGroups.Old.ToString();
        }
    }

    public object ConvertBack(object value,
System.Type targetType, object parameter,
System.Globalization.CultureInfo culture)
    {
        return
System.Convert.ToInt32(Enum.Parse(typeof(AgeGroups),
value));
    }
}

```

VisualBasic.NET

```

Public Enum AgeGroups
    Young = 18
    Mature = 60
    Old = 150
End Enum

<ValueConversion(GetType(Integer), GetType(String))>
Public Class AgeConverter
    Implements IValueConverter

    Public Function Convert(ByVal value As Object,
ByVal targetType As System.Type, ByVal parameter As
Object, ByVal culture As
System.Globalization.CultureInfo) As Object Implements
System.Windows.Data.IValueConverter.Convert
        Dim number = System.Convert.ToInt32(value)
        If number < AgeGroups.Young Then
            Return AgeGroups.Young.ToString()
        ElseIf number < AgeGroups.Mature Then
            Return AgeGroups.Mature.ToString()
        Else
            Return AgeGroups.Old.ToString()
        End If
    End Function

    Public Function ConvertBack(ByVal value As Object,
ByVal targetType As System.Type, ByVal parameter As
Object, ByVal culture As
System.Globalization.CultureInfo) As Object Implements
System.Windows.Data.IValueConverter.ConvertBack

```

```

        Return
        System.Convert.ToInt32([Enum].Parse(GetType(AgeGroups)
        , value))
    End Function
End Class

```

Como se puede ver en el ejemplo hemos codificado un conversor entre un tipo enumerado `AgeGroups` y los enteros, de forma que los datos de numéricos sean interpretados elemento distribuidor entre los diferentes valores del enumerado y viceversa. Así una edad de menos de 18 será considerada como `Young`, entre 18 y 60 `Mature` y mas de 60 `Old`.

Para ello simplemente hemos creado una clase calificada por el atributo ***ValueConversion***, que informa de que tipo, el situado como primer parámetro, es el origen y cual, el segundo, es el destino y que implementa el interfaz *IValueConverter*, que define dos métodos *Convert* que recibe como parámetro un value del tipo origen y devuelve el valor equivalente del tipo destino y un método *ConvertBack* que realiza justo la operación inversa.

Pero una vez codificado esto no es más que una clase, ¿qué debemos hacer para usarlo en una expresión concreta? Las expresiones de binding además de *Path* y *Source* también poseen la propiedad *Converter*. En esta se puede especificar cuál es el conversor usado. ¿Pero cómo? Lo primero que nos damos cuenta es que la clase del conversor no es visible desde XAML, esto quiere decir que o bien asignamos el conversor a través de código, o bien hacemos posible su referencia desde el XAML.

Es esta segunda opción la más razonable y para ello lo que tenemos que hacer es ampliar el rango de XAML válido, como lo haríamos si estuviéramos creando un nuevo control. Para ello lo único que tenemos que hacer es definir un nuevo espacio de nombres XML a nivel del contenedor deseado, que nos va a permitir acoger estas nuevas marcas dentro de su ámbito. En el ejemplo:

```

XAML
<Window x:Class="MainWindow"

xmlns="http://schemas.microsoft.com/wfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/wfx/2006/xaml"
    xmlns:app="clr-namespace:WpfApplication2"
    Title="MainWindow" Height="350" Width="525">

```

La sentencia *xmlns:app="clr-namespace:WpfApplication2"* nos indica que el espacio de nombres XML de nombre *app*, es de tipo código e importa las clases definidas en el espacio de nombres de código *clr-namespace*, de nombre

WpfApplication2. Este espacio de nombres ha de estar accesible directamente o como referencia en la aplicación, y todas la clases públicas declaradas en él pasan automáticamente a ser referenciables desde el XAML, lo cual no quiere decir que se puedan incluir en cualquier parte.

Obviamente una clase que no sea un control no podrá ser situada en un lugar destinado a controles como este es el caso. Un conversor es una clase de gestión que WPF no sabe como pintar y si queremos insertarla en marcado deberíamos hacerlo allí donde efectúa la acción para la que ha sido concebida convertir, o sea asignada a la propiedad Converter de la expresión de binding, algo similar a esto {Binding Path=Edad, Converter=app:AgeConverter}. Desgraciadamente esto no devolverá un error debido a que app no es visible dentro de una expresión de binding, luego la solución que tenemos es la siguiente, primero declaramos en código nuestro conversor en el único sitio posible, los recursos, que veremos un poquito más adelante en este mismo capítulo:

XAML

```
<Window.Resources>
    <app:AgeConverter x:Key="AgeConverter1" />
</Window.Resources>
```

Después y utilizando un binding de recursos podemos asignar el conversor por su clave:

XAML

```
<ComboBox Height="23" HorizontalAlignment="Left"
    Margin="12, 155, 0, 0" Name="ComboBox1"
    VerticalAlignment="Top" Width="120" Text="{Binding
    Path=Edad, Converter={StaticResource AgeConverter1}}"
/>
```

En cualquier caso se recomienda al lector revisitar esta sección una vez se haya leído los epígrafes relacionados con recursos.

## Validación

La validación es el proceso por el cual el dato se prueba correcto antes de ser definitivamente asignado por un proceso de binding.

Este proceso de validación se puede llevar a cabo de varias maneras, con el fin de que utilicemos la que más nos convenga.

El primer camino es el conocido como validación mediante propiedades y básicamente consiste en que en las propiedades de la clase de datos elevemos un excepción, si es que no se hace ya, cuando la información no cumpla unos criterios de validez. El quid de la cuestión se encuentra en la expresión de databinding.

Además de los correspondientes Source, Path, etc. debemos, si queremos que el proceso de binding interprete estas excepciones como procesos de validación y no como excepciones comunes, que hacer constar en las expresiones de binding las siguientes especificaciones:

```
XAML
Text="{Binding Edad, Mode=TwoWay,
NotifyOnValidationError=True,
ValidatesOnExceptions=true}"
```

Donde *NotifyOnValidationError* informa a la expresión que debe elevar el evento de validación cuando se produzca una operación de sincronización. *ValidatesOnExceptions* determina que las excepciones durante el proceso de binding deben ser tomadas como errores de validación.

Otra manera es hacer implementar a nuestra clase de datos el interfaz *IdataErrorInfo*. Este interfaz obliga a toda clase que lo implemente a declarar dos propiedades Error y una propiedad indexada, this en C# e Item en Visual Basic.NET.

La propiedad *Error* está destinada a recoger un mensaje de error general, mientras que la propiedad indexada, nos permite introducir un mensaje por propiedad, ya que son los nombres de estas los que se utilizan como indexador.

```
C#
public class Persona: INotifyPropertyChanged,
IDataErrorInfo
{
    private string _nombre;
    public string Nombre
    {
        get { return _nombre; }
        set {
            if (value.Length > 0)
            {
                _nombre = value;
                OnPropertyChanged("Nombre");
            }
            else errors.Add("Nombre", "Nombre no
puede estar vacío");
        }
    }
}
```

```

        private void OnPropertyChanged(string p)
        {
            if (PropertyChanged != null)
            {
                PropertyChanged(this, new
PropertyChangEdEventArgs(p));
            }
        }
        public event PropertyChangedEventHandler
PropertyChanged;
        Dictionary<string, string> errors = new
Dictionary<string, string>();
        public string Error
        {
            get { return string.Format("Hay {0} en
total", errors.Count); }
        }
        public string this[string columnName]
        {
            get { return errors[columnName]; }
        }
    }

```

Visual Basic.NET

```

Public Class Persona
    Implements INotifyPropertyChanged
    Implements IDataErrorInfo
    Private _nombre As String
    Public Property Nombre() As String
        Get
            Return _nombre
        End Get
        Set
            If value.Length > 0 Then
                _nombre = value
                OnPropertyChanged("Nombre")
            Else
                errors.Add("Nombre", "Nombre no
puede estar vacío")
            End If
        End Set
    End Property
    Private Bus OnPropertyChanged(p As String)
        RaiseEvent PropertyChanged(Me, New
PropertyChangEdEventArgs(p))
    End Sub
    Public Event PropertyChanged As
PropertyChangedEventHandler
    Private errors As New Dictionary(Of String,
String)()
    Public ReadOnly Property [Error]() As String
        Get
            Return String.Format("Hay {0} en

```

```

        total ", errors.Count)
            End Get
        End Property

        Public Default ReadOnly Property Item(columnName
As String) As String
            Get
                Return errors(columnName)
            End Get
        End Property
    End Class

```

En este ejemplo podemos ver como se valida una propiedad que es evaluada para que la longitud no sea menor que 0. Para que la expresión sea sensible a estos errores hemos de habilitar *ValidatesOnDataErrors* en vez de el *ValidatesOnExceptions* anterior, aunque sean perfectamente compatibles.

XAML

```

Text="{Binding Nombre, Mode=TwoWay,
ValidatesOnDataErrors=True,
NotifyOnValidationError=True}"

```

También es posible implementar el interfaz *InotifyDataErrorInfo* muy similar al *IDataErrorInfo* pero con estructura de ejecución asíncrona.

Pero sin duda el mecanismo más popular de validación son las reglas de validación.

Este mecanismo consiste en crear clases que desciendan de la clase *ValidationRule* y que permiten luego ser añadidas a cualquier expresión de binding. Es esta capacidad de reutilización lo que las hace tan populares.

Por otro lado la implementación es realmente sencilla, solo hemos de sobrecargar en la clase el método *Validate*, que recibe un objeto y devuelve un *ValidationResult*, clase que nos permite determinar si el resultado fue positivo o negativo y devolver también un mensaje.

La inclusión en la expresión de binding se puede hacer mediante la misma técnica que el conversor, simplemente importando el espacio de nombres *clr* como *xml* y listo.

C#

```

public class NewRule : ValidationRule
{
    public override ValidationResult Validate(object
value, CultureInfo cultureInfo)
    {
    }
}

```

**Visual Basic.NET**

```

Public Class NewRule
    Inherits ValidationRule
    Public Overrides Function Validate(value As Object,
        cultureInfo As CultureInfo) As ValidationResult

        End Function
End Class

```

**XAML**

```

<TextBox>
  <TextBox.Text>
    <Binding Path="Age"
      UpdateSourceTrigger="PropertyChanged" >
      <Binding.ValidationRules>
        <c: NewRule />
      </Binding.ValidationRules>
    </Binding>
  </TextBox.Text>
</TextBox>

```

## Integración con Visual Studio

Visual Studio se ha destacado como una herramienta, potente, cómoda y productiva en lo referente a la programación .NET en general, pero con WPF el tratamiento ha sido diferente que con otras tecnologías como Windows Communication Foundation o Windows Workflow Foundation.

La primera experiencia de desarrollo sobre WPF la tuvimos con VS2005 y no fue precisamente placentera. El producto no soportaba el desarrollo de WPF, como era lógico por otra parte ya que fue diseñado para programar sólo sobre la versión 2.0 del Framework, así que debíamos instalar unos “Add-Ons” que nos aportaban poco más, un previsualizador de XAML que permitía obtener una imagen del XAML que íbamos codificando y que solía fallar a poco que utilizáramos una característica poco común. De otro lado también nos permitía el uso de un editor de propiedades que tan sólo era de utilidad para conocer el catálogo de propiedades, ya que su edición era realmente imposible en casi cualquier propiedad que se saliera de lo habitual. Además no contemplaba los eventos.

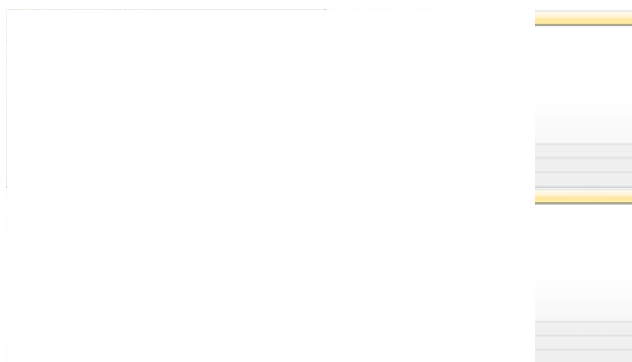
En VS2008 se mejoraron algunas cosas como la fiabilidad del visualizador o la inclusión de los eventos en el editor de propiedades, pero la edición visual seguía cuando menos siendo incómoda.

Todo esto se agravaba especialmente en el caso del binding, ya que no existía ningún tipo de simplificación y ayuda en lo que posiblemente se pueda considerar el mecanismo más difícil de toda la tecnología.

Afortunadamente esto ha cambiado en VS2010. Son muchas las mejoras que se han incorporado, pero en lo tocante al binding son básicamente dos: el editor de expresiones y los orígenes de datos.

El editor de expresiones simplifica drásticamente el cálculo de expresiones.

Como podemos ver en la primera imagen el editor nos permite establecer la mayoría de las opciones de binding mediante un editor visual. Lo cual es sumamente útil porque si bien los bindings sencillos son fáciles de establecer por código, los menos evidentes pueden ser bastante difíciles.



También podemos observar que el origen de datos, en la propiedad Source se puede establecer sin problemas ya provengan estos de una base de datos, de otro elemento dentro del interfaz o de un recurso, como veremos un poco más adelante en este mismo capítulo.

Es también posible establecer el resto de parámetros de las expresiones de binding a través del editor. Como se ve en la segunda imagen podemos establecer la propiedad Path que se nos mostrará contextualizada al Source que hayamos elegido. También se nos permitirá seleccionar el conversor de las propiedades de formato, modo y validación.

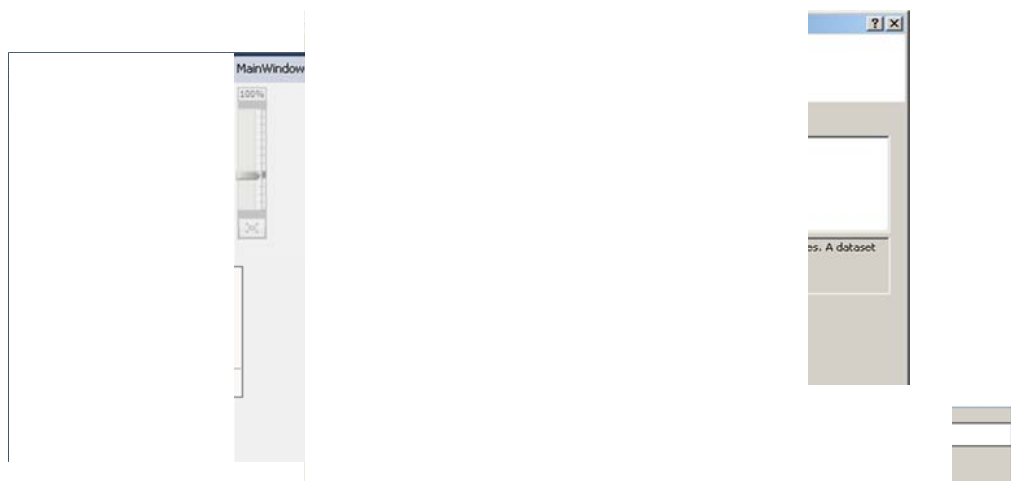
La segunda de las integraciones que nos aporta VS es la integración con la ventana de Orígenes de Datos.

Cuando incluimos en nuestro proyecto un DataSet o unObjectContext (Entity Framework) este automáticamente se asocia con la ventana de orígenes de datos e





inmediatamente se publican tanto sus tablas u ObjectQuerys como los diferentes campos/propiedades que contienen. De esta manera con un simple pulsar y arrastrar podemos depositar los diferentes controles en el formulario. Además esta ventana nos permite seleccionar el control de edición, que por ejemplo en el caso de una tabla puede ser un DataGrid, un ListView o un conjunto de controles que conforman un editor tipo ficha.



Además esta ventana no sólo inserta controles sino que la automatización incluye la inserción del código para rellenar el DataSet o abrir elObjectContext. Al hacer esto inserta también un elemento en el marcado, en concreto en el área de recursos que es un `CollectionViewSource` que será, junto con los `DataProviders`, el objetivo de nuestro siguiente punto.

## DataProviders y CollectionViewSource

Cuando hablamos del binding múltiple hablamos también de la estructura interna que este tenía, de como cada colección creaba un *ICollectionView* y de como este objeto era necesario para intermediar a fin de soportar ordenación y filtrado, ahora abordaremos como.

*CollectionViewSource* es una clase que puede funcionar como estática o dinámica, esto nos permite que a través del método estático ***GetDefaultView*** obtengamos el objeto *ICollectionView* o nos permite insertarla en marcado, como referimos en el apartado anterior para enlazar de manera declarativa con los controles.

Es el primer aspecto que nos va a ocupar, ya que la manipulación del *ICollectionView* nos va a permitir el filtrado y ordenado programático.

El método *GetDefaultView* admite un parámetro, que ha de ser una colección y nos devuelve un objeto que implementa el interfaz *ICollectionView*.

Este interfaz obliga a la clase que lo implementa, sea cual sea, a tener los siguiente miembros entre otros:

- *MoveCurrentToFirst*, sitúa el elemento seleccionado de la colección como el primero.
- *MoveCurrentToLast*, sitúa el elemento seleccionado de la colección como el último.
- *MoveCurrentToNext*, sitúa el elemento seleccionado de la colección como el siguiente al actual.
- *MoveCurrentToPrevious*, sitúa el elemento seleccionado de la colección como el anterior al actual.
- *CurrentItem*, nos da acceso al elemento actual.
- *SortDescriptions*, colección de elementos *SortDescription* que al ser creados solicitan el nombre de una propiedad y una dirección con las cuales ordenar la vista. Al ser una colección el orden de entrada establece el criterio de aplicación de los diferentes elementos de orden.
- *Filter*, es una propiedad de tipo *Predicate<Object>* o *Predicate(Of Object)*, que básicamente quiere decir que el dato a asignar es un delegado, el cual recibe un objeto, que será cada uno de los objetos de la colección, y devuelve un dato booleano de forma que verdadero equivale a que el dato no se filtra, falso a que si.

C#

```

    ...
    ICollectionView cv;

    public ICollectionView CV
    {
        get {
            if (cv == null)
                CV =
                CollectionViewSource.GetDefaultView(this.Resources["I

```

```

        sta"]);
        return cv;
    }

    private void Button_Click(object sender,
        RoutedEventArgs e)
    {
        CV.MoveCurrentToFirst();
    }

    ...

    private void Button_Click_4(object sender,
        RoutedEventArgs e)
    {
        if (CV.SortDescriptions.Count > 0)
        {
            CV.SortDescriptions.Clear();
        }
        else
        {
            CV.SortDescriptions.Add(new SortDescription {
                PropertyName = "Apellidos", Direction =
                ListSortDirection.Descending });
        }
    }

    private void Button_Click_5(object sender,
        RoutedEventArgs e)
    {
        CV.Filter = delegate(object item)
        {
            return ((Persona)item).Nombre.Length > 5;
        };
    }
    ...

```

Visual Basic.NET

```

...
Private m_cv As ICollectionView

Public ReadOnly Property CV() As ICollectionView
    Get
        If m_cv Is Nothing Then
            m_cv =
            CollectionViewSource.DefaultView(Me.Resources("Lista"))
        End If
        Return m_cv
    End Get
End Property

```

```

Private Sub Button_Click(sender As Object, e As
RoutedEventArgs)
    CV.MoveCurrentToFirst()
End Sub

...

Private Sub Button_Click_4(sender As Object, e As
RoutedEventArgs)
    If CV.SortDescriptions.Count > 0 Then
        CV.SortDescriptions.Clear()
    Else
        CV.SortDescriptions.Add(New
SortDescription() With { _
                        Key .PropertyName = "Apellidos", _
                        Key .Direction =
ListSortDirection.Descending _
                    })
    End If
End Sub

Private Sub Button_Click_5(sender As Object, e As
RoutedEventArgs)
    CV.Filter = Function(item As Object)
DirectCast(item, Persona).Nombre.Length > 5
End Sub
...

```

El *CollectionViewSource* también puede funcionar como un elemento insertado en XAML que nos facilite el enlace mediante los editores y herramientas de Visual Studio, pero no es el único, también disponemos de los *DataProviders*.

Un *DataProvider* existe para precisamente poder insertar un objeto de intermediación en el marcado que nos permita insertar fácilmente los datos de un objeto o un XML en los interfaces creados por WPF.

Existen dos tipos:

1. *ObjectDataProvider*, que permite insertar objetos definidos en código a través de su propiedad *ObjectInstance* establece el objeto que deseamos, y con *MethodName* y *MethodParameters* el método a ejecutar.
2. *XMLDataProvider*, que permite a través de su propiedad *Source* cargar un XML y utilizarlo como origen de datos. Este caso es especialmente interesante porque es el único en que el origen de datos no es un objeto o una colección de ellos.

Como vemos el *XMLDataProvider* es ciertamente especial, tanto que modifica la expresión de binding transformando la propiedad *Path* en *XPath*, en la cual

se hace constar una propiedad en una expresión basada en *XPath* como podemos ver en el siguiente ejemplo:

```
XAML
<TextBox Text="{Binding XPath=@Nombre}" />
```

## Recursos

Los recursos son uno de los elementos más importantes de WPF. De hecho ya hemos visto que nos permiten cosas como introducir elementos no visuales en marcado, pero además de esto en los siguientes capítulos veremos muchos más usos que nos mostrarán el potencial de los recursos en WPF. Ahora toca explicar qué son y cómo afectan al binding.

Los recursos corresponden a la propiedad *Resources* que se define en la clase *FrameworkElement*. Esto significa que están disponibles en todos los elementos que de ella descienden, “de facto” en todas las clases de WPF.

Esta propiedad está formulada como del tipo *ResourceDictionary*. Este tipo vendría a ser lo mismo que un diccionario de objetos cuyas claves también son objetos. Esto quiere decir que tenemos una colección cuyos elementos tienen cada uno una clave y un valor que son objetos.

Debido a esto en los recursos se puede guardar cualquier cosa, lo que los hace muy versátiles, pero además los recursos tiene una forma de funcionar que les permite en tiempo de ejecución comportarse casi como si fueran una propiedad creada a partir de toda la jerarquía de elementos en XAML. Esto quiere decir que si tenemos un botón dentro de un panel y a su vez el panel dentro de una ventana, y accediésemos en el botón a los recursos mediante una expresión de binding, esta buscaría el origen de datos en los recursos del botón, si no en el panel y por último en la ventana hasta encontrarlo. Comportamiento este muy similar al de los *DataContext*.

Las expresiones de binding de recursos son muy sencillas, como todas ellas van enmarcadas por sendas llaves para a continuación cambiar la palabra binding por *StaticResource* o *DynamicResource*. La única propiedad que ambas tienen es *ResourceKey* que indica, por su nombre el recurso al que nos queremos referir.

```
XAML
<Page Name="root"
    xmlns="http://schemas.microsoft.com/wfx/2006/xaml/pr
```

```

esentati on"

xml ns: x="http://schemas.microsoft.com/wpf/2006/xaml "
>
  <Page.Resources>
    <SolidColorBrush x:Key="MyBrush" Color="Gold"/>
  </Page.Resources>
  <StackPanel>
    <TextBlock DockPanel.Dock="Top"
      HorizontalAlignment="Left" FontSize="36"
      Foreground="{StaticResource MyBrush}" Text="Text"
      Margin="20" />
  </StackPanel>
</Page>

```

Como se puede ver en el ejemplo hemos creado un recurso de tipo brocha con el que luego establecemos el color de la fuente de un TextBlock. Una de las cosas importantes que hay que recalcar en el caso de los recursos, es que todos ellos han de establecer la propiedad *x:Key*, ya no porque de otra manera no se podrían asignar, si no porque estos elementos básicamente representan una colección en la que la propiedad *Key* es obligatoria y si no se asigna la clase correspondiente, *ResourceDictionary*, no se podría crear.

## Conclusión

Los procesos de binding están diseñados para facilitarnos el código, una correcta inclusión de ellos puede hacer que nuestro código se reduzca considerablemente, al tiempo que delegamos tareas pesadas en el propio sistema. WPF nos permite hacer uso de un motor de binding muy potente.

## A Continuación

Los comandos son un buen recurso, disponible en WPF, para crear funcionalidad y asignarla de manera consistente a múltiples controles y de manera consistente a lo largo de nuestro interfaz. En el siguiente capítulo veremos por qué.

# Comandos

Podríamos describir los comandos como pedazos de funcionalidad que se pueden asignar a determinados tipos de controles, botones y menús, pero que además están asociados con un gesto de entrada, básicamente una combinación de teclas, ratón y/o lápiz óptico.

Esto en principio puede parecer bastante trivial, pero fundamentalmente nos permite crear funcionalidad de manera consistente que luego pueda ser desplegada y utilizada a lo largo de la aplicación, o debidamente empaquetada en otras aplicaciones.

De hecho un comando puede establecerse a nivel de control o de aplicación, y estos últimos quedan debidamente registrados para poder ser invocados desde cualquier lado en la aplicación.

## Redefinir un comando

En el modelo de objetos de WPF ya se incluyen la definición de diferentes comandos, por ejemplo *ApplicationCommands*, *NavigationCommands*, *MediaCommands*, *EditingCommands* y *ComponentCommands*. En algunos casos estos comandos contienen ya una funcionalidad asociada, pero en otras

no. En cualquier caso ser capaz de dotar a un comando ya existente de funcionalidad ajustada a nuestra aplicación es muy útil.

De hecho estos comandos no sólo ya existen sino que están vinculados a los eventos de sistema y combinaciones de teclas más comunes. Esto simplifica significativamente adaptar las aplicaciones y sus respuestas a lo esperado.

La clave para redefinir un comando ya existente es recrear su *CommandBinding*. Esta clase se encarga de asociar un *ICommand* con un *ExecutedRoutedEventHandler*. Posteriormente este *CommandBinding* hay que agregarlo a la colección *CommandBindings* de cualquier descendiente de *UIElement*.

*UIElement* es el descendiente de todos los elementos de interfaz en WPF, estas clases se caracterizan por poder interactuar con el usuario.

C#

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
        CommandBinding cb = new
        CommandBinding(ApplicationCommands.Close, new
        ExecutedRoutedEventHandler(Salir));
        this.CommandBindings.Add(cb);
    }
    private void Salir(object sender, RoutedEventArgs e)
    {
        Application.Current.Shutdown();
    }
}
```

Visual Basic.NET

```
Public Partial Class MainWindow
    Inherits Window
    Public Sub New()
        InitializeComponent()
        Dim cb As New
        CommandBinding(ApplicationCommands.Close, New
        ExecutedRoutedEventHandler(AddressOf Salir))
        Me.CommandBindings.Add(cb)
    End Sub
    Private Sub Salir(sender As Object, e As
    ExecutedRoutedEventArgs)
        Application.Current.Shutdown()
    End Sub
End Class
```



Como se puede ver en el ejemplo sobrecargamos el comando existente *Close* con el método *Salir*, que básicamente cierra la aplicación.

Esto básicamente cerrará la aplicación cuanto reciba la señal adecuada. Lo importante de este comando es que además se puede conectar con algunos controles utilizando la propiedad *Command*. Esta está disponible específicamente en los botones y en los *MenuItems*.

XAML

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="20" />
        <RowDefinition Height="20" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>
    <Menu>
        <MenuItem Header="Archivo">
            <MenuItem Header="Salir"
                Command="ApplicationCommands.Close" />
        </MenuItem>
    </Menu>
    <ToolBarTray Grid.Row="1">
        <ToolBar>
            <Button Content="Salir"
                Command="Close" />
        </ToolBar>
    </ToolBarTray>
    <TextBox x:Name="Texto" Grid.Row="2"
        AcceptsReturn="True" />
</Grid>
```

## Crear un comando nuevo

Si el comando es nuevo debemos hacer alguna tarea añadida. Básicamente al no disponer de una clase de comando necesitamos una. Opcionalmente además necesitaríamos también un gesto asociado al comando y por último deberíamos asociar el comando con los diferentes elementos en el interfaz, teniendo en cuenta que el comando al no pertenecer a la API por defecto necesitamos incluirlo en el marcado.

Como primera tarea debemos crear un comando, que es el resultado de una clase común y corriente hacerle implementar el interfaz *ICommand* o simplemente de crear una variable de tipo *RoutedCommand*:

```
C#
    RoutedCommand abrir;
    InputGestureCollection gesto = new
    InputGestureCollection(); gesto.Add(new
    KeyGesture(Key.L, ModifierKeys.Alt)); abrir = new
    RoutedCommand("Abrir", typeof(Application), gesto);
```

```
Visual Basic.NET
    Dim abrir As RoutedCommand
    Dim gesto As New InputGestureCollection()
    gesto.Add(New KeyGesture(Key.L, ModifierKeys.Alt))
    abrir = New RoutedCommand("Abrir",
    GetType(Application), gesto)
```

Además en el ejemplo también podemos ver como se crea una colección de gestos, a la que se agrega una determinada combinación de teclas, alt+L en concreto, y como se vincula con el comando.

Para asignar el comando a un control lo podemos hacer de dos maneras, la primera tan sencilla como asignar el comando a la propiedad Command del control:

```
C#
    MenuAbrir.Command = abrir;
```

```
Visual Basic.NET
    MenuAbrir.Command = abrir
```

La segunda es incluir nuestro espacio de nombres en el marcado y enlazar el comando en marcado. Esto funcionará obviamente si el comando está definido como una clase, de lo contrario el sistema no interpretará que este pueda introducirse en marcado:

```
C#
    class AbrirCommand: ICommand
    {
        public bool CanExecute(object parameter)
        {
            return true;
        }

        public event EventHandler CanExecuteChanged;

        public void Execute(object parameter)
        {
            Application.Current.Shutdown();
        }
    }
```

Visual Basic.NET

```

Class Abri rCommand
    Implements ICommand
    Public Function CanExecute(parameter As Object)
        As Boolean
        Return True
    End Function

    Public Event CanExecuteChanged As EventHandler

    Public Sub Execute(parameter As Object)
        Application.Current.Shutdown()
    End Sub
End Class

```

XAML

```

<Window x:Class="Ejerci ci oVI . Mai nWindow"

xmlns="http://schemas.microsoft.com/wpf/2006/xaml /pr
esentation"

xmlns:x="http://schemas.microsoft.com/wpf/2006/xaml "
    xmlns:mi app="clr-namespace: Ejerci ci oVI "
    Title="Mai nWindow" Height="350" Width="525">
    <Grid>
        <Grid.Resources>
            <mi app: Abri rCommand x: Key="comando" />
        </Grid.Resources>
        <Menu>
            <MenuItem Header="Archi vo">
                <MenuItem Header="Sal ir"
Command="{StaticResource comando}" />
            </MenuItem>
        </Menu>
        <ToolBarTray Grid.Row="1">
            <ToolBar>
                <Button Content="Sal ir"
Command="{StaticResource comando}" />
            </ToolBar>
        </ToolBarTray>
        <TextBox x: Name="Texto" Grid.Row="2"
AcceptsReturn="True" />
    </Grid>
</Window>

```

Para introducir la clase en el marcado continuamos la estrategia que seguimos anteriormente con el conversor. Introducir el espacio de nombres XML y declarar el comando como un recurso, que como vimos es posible porque los recursos pueden ser cualquier objeto y enlazarlo mediante binding con la propiedad Command en marcado. Recordar que hay que hacerlo así porque

como contenido de una propiedad en marcado no se pueden introducir espacios de nombres XML.

## Comandos en los nuevos controles

En ocasiones los comandos no pueden ser asociados a la colección `CommandBindings`. Es muy típico por ejemplo que esto sea así en la construcción de controles nuevos, y se debe fundamentalmente a que se accede desde el lado inadecuado.

C#

```
public class Comentario : ContentControl
{
    static Comentario()
    {
        DefaultStyleKeyProperty.OverrideMetadata(typeof(Comentario), new
        FrameworkPropertyMetadata(typeof(Comentario)));

        CommandManager.RegisterClassCommandBinding(typeof(Comentario), new CommandBinding(_ocular, oculando));

        CommandManager.RegisterClassInputBinding(typeof(Comentario), new InputBinding(_ocular, new
        MouseGesture(MouseAction.LeftClick)));
    }
    ...
}
```

Visual Basic.NET

```
Public Class Comentario
    Inherits ContentControl
    Shared Sub New()

        DefaultStyleKeyProperty.OverrideMetadata(GetType(Comentario), New
        FrameworkPropertyMetadata(GetType(Comentario)))

        CommandManager.RegisterClassCommandBinding(GetType(Comentario), New CommandBinding(_ocular,
        oculando))

        CommandManager.RegisterClassInputBinding(GetType(Comentario), New InputBinding(_ocular, new
        MouseGesture(MouseAction.LeftClick)))
    End Sub
End Class
```

```

    (Comentario), New InputBinding(_ocular, New
    MouseGesture(MouseAction.LeftClick)))
    End Sub
    ...
End Class

```

Como se puede ver en el ejemplo la clase está registrando el comando en el constructor de la clase, que no puede acceder a la propiedad `CommandBindings` porque esta no es una propiedad de clase. Para estos casos se tiene la clase *CommandManager*, que nos provee de dos métodos *RegisterClassCommandBinding*, que registra un *CommandBinding* asociada con una clase que suele ser la clase del control normalmente y *RetisterClassInputBinding*, que nos permite asociar al comando un gesto.

## Comandos en 4.0

En la versión 4.0 del Framework se han incluido algunas características nuevas para el trabajo con comandos, para la simplificación del proceso.

En concreto se ha creado la etiqueta *KeyBinding*. Que se sitúa en una colección de recursos, ya que no es visual y es capaz de enlazar con un comando escrito como propiedad de la ventana a la que pertenece, y esto, y aquí viene el ahorro, sin necesidad de insertar en marcado el espacio de nombres XML.

C#

```

public partial class MainWindow : Window
{
    public EHCommand Comando { get; set; }
    public MainWindow()
    {
        ...
    }
}

```

Visual Basic.NET

```

Public Partial Class MainWindow
    Inherits Window
    Public Property Comando() As EHCommand
        Get
            Return m_Comando
        End Get
        Set
            m_Comando = Value
        End Set
    End Set

```

```

        End Property
        Private m_Comando As EHCommand
        Public Sub New()
            ...
        End Sub
    End Class

```

XAML

```

<Window x:Class="Ejercicio0111.Mai nWindow"

xmlns="http://schemas.microsoft.com/wfx/2006/xaml /pr
esentation"

xmlns:x="http://schemas.microsoft.com/wfx/2006/xaml "
    Title="Mai nWindow" Height="350" Width="525">
    <Window.InputBindings>
        <KeyBinding Command="{Binding Comando}"
Gesture="CTRL+D" />
    </Window.InputBindings>
    <Grid>
        <Button Content="Pul sa me..."
Command="{Binding Comando}" Width="100" Height="30" />
    </Grid>
</Window>

```

## A Continuación

Los comandos nos ayudan a definir funcionalidad de una manera muy regular y exportable, que sin duda permite crear un interfaz de conexión entre las ventanas de la aplicación y la lógica de negocio.

Esto además permite que de manera extraordinariamente rápida podamos realizar drásticos rediseños o propagar la solución a un error encontrado inmediatamente.

## A Continuación

XAML al igual que HTML es compatible con un sistema de estilos, que nos permite variar rápidamente los estilos de una aplicación. Además veremos que es una plantilla, que de manera sorprendente nos va a permitir redibujar cualquier control a nuestro antojo.

# Estilos Y Plantillas

Cuando Microsoft decidió crear WPF varias fueron las cosas en las que se inspiró. Pero quizá una de las más significativas fue el desarrollo web y las tecnologías para hacerlo.

Efectivamente la idea del marcado no era nueva, así como la separación efectiva entre interfaz y lógica de interfaz. Estas características ya se habían utilizado en el desarrollo web. Pero se quiso ir más allá y se introdujo también una técnica de plantillas para emular lo que en el ámbito web hace CSS.

Además, y no contentos con ello, se ha incluido un sistema de plantillas que bien podría emular el comportamiento de los controles de servidor ASP.NET.

Estos controles se definen en el servidor, pero dado que no pertenecen a la especificación estándar de HTML, los navegadores no saben pintarlos. La solución se encuentra en que estos métodos poseen un método llamado `Render` que se ejecuta antes de que la página sea servida, insertando el HTML equivalente al control.

Algo similar hacen los controles en WPF. Cada control tiene lo que se conoce como una plantilla que es un conjunto de Visuales que son insertados en el

lugar del control. La diferencia es que el control de por si no se puede pintar pero los visuales si.

Las plantillas pueden pintar todo el control o sólo una parte, por ejemplo es habitual que controles como el `ListBox` tenga una plantilla para ellos y otra para sus elementos.

## Estilos

En WPF entendemos por estilo una determinada configuración en cuanto a las propiedades de un objeto de una determinada clase. Cuando esta configuración se asigna, todos los valores se asignan, transmitiendo al objeto un estado concreto.

Un estilo en WPF consiste básicamente en una serie de asignaciones establecidas por una tag `<setter>` dentro de una tag `<style>`.

```
XAML
<Button>
  <Button.Style>
    <Style>
      <Setter Property="VerticalAlignment"
Value="Center" />
    </Style>
  </Button.Style>
</Button>
```

En el ejemplo anterior podemos comprobar como un botón tiene asignado en su propiedad *Style* un estilo, que establece su propiedad *VerticalAlignment* a *Center*. Esto produce exactamente el mismo efecto que si hubiéramos asignado la propiedad directamente. ¿Para qué sirve entonces?. Bueno, los estilos se pueden definir como recursos y asignarse discriminadamente mediante binding de recursos.

```
XAML
<Grid>
  <Grid.Resources>
    <Style x:Key="EstiloBoton">
      <Setter Property="VerticalAlignment"
Value="Center" />
    </Style>
  </Grid.Resources>
  <Button Style="{StaticResource EstiloBoton}" />
</Grid>
```



En esta configuración la asignación se convierte en muy apropiada si tenemos más de un botón dentro del ámbito, en este caso un grid, porque ahorramos código en cuanto el estilo establezca más de una propiedad, además de ahorrar en consistencia, hay poca probabilidad de olvidar alguna asignación.

Sin embargo cualquier observador que haya probado estos códigos habrá visto que el *Intellisense* de VS al ir introduciendo las tag *setter* y establecer la propiedad *Property*, nos sale una lista de propiedades cuando menos incompleta. Esto se debe a que todos los estilos se crean para una clase concreta que ha de ser especificada implícitamente, porque si no el sistema entenderá que es para la clase *FrameworkElement*, y solo mostrará sus propiedades.

Para poder cambiar la clase a la que va dirigida un estilo hemos de especificar la propiedad *TargetType* en la que haremos constar, por su nombre, la clase.

XAML

```
<Grid>
  <Grid.Resources>
    <Style x:Key="EstiloBoton"
      TargetType="Button">
      <Setter Property="VerticalAlignment"
        Value="Center" />
      <Setter Property="Background" Value="Red"
    />
    </Style>
  </Grid.Resources>
  <Button Style="{StaticResource EstiloBoton}" />
</Grid>
```

Como vemos en el ejemplo ya podemos establecer propiedades de la clase *Button* insisto, no por el *Intellisense* solamente, sino porque el sistema entiende que el estilo va dirigido a los botones.

Otra manera más específica de asignar propiedades de una clase es la de hacer constar la clase precediendo al nombre de la propiedad en el *Setter*. De esta manera evitamos tener que atribuir un tipo a todo el tipo y podemos crear estilos de aplicación múltiple.

XAML

```
<Grid>
  <Grid.Resources>
    <Style x:Key="EstiloBoton">
      <Setter
        Property="FrameworkElement.VerticalAlignment"
        Value="Center" />
      <Setter Property="Button.Background"
        Value="Red" />
      <Setter Property="RadioButton.IsChecked" />
    </Style>
  </Grid.Resources>
  <Button Style="{StaticResource EstiloBoton}" />
  <RadioButton Style="{StaticResource EstiloBoton}" />
</Grid>
```

```

    Value="True" />
    </Style>
  </Grid.Resources>
  <Button Style="{StaticResource EstiloBoton}" />
  <RadioButton Style="{StaticResource
EstiloBoton}" />
</Grid>

```

Otra característica interesante es que el estilo, cuando se encuentra definido dentro de los recursos acepta una convención especial, en ausencia de la propiedad `x:Key` (de ahí la expresión convención, porque como dijimos en el capítulo de recursos no se pueden tener elementos en ellos sin `x:Key`) los estilos se convierten en estilos de asignación automática para todos aquellos elementos cuya clase coincida y caigan dentro de su ámbito.

XAML

```

<Grid>
  <Grid.Resources>
    <Style TargetType="Button">
      <Setter Property="VerticalAlignment"
Value="Center" />
      <Setter Property="Background" Value="Red"
/>
    </Style>
  </Grid.Resources>
  <Button />
</Grid>

```

Esta sección de código produciría la misma salida que el anterior aunque no existe una asignación específica.

La última de las características básicas que vamos a comentar sobre los estilos es la herencia. Esta característica se soporta y produce, convenientemente usada, un ahorro de esfuerzo en la creación de estilos consistentes para todos los elementos de la aplicación. Esto quiere decir que si diseñamos correctamente nuestros estilos pueden evitarnos trabajo, ya que podemos definir estilos desde lo más general a lo más concreto, que paulatinamente asignen las propiedades en una jerarquía de estilos que nos evite código reiterado. Para que un estilo pueda heredar de otro simplemente tenemos que hacer constar en su propiedad *BaseOn*, que estilo ha de ser su antecesor.

XAML

```

<Grid>
  <Grid.Resources>
    <Style x:Key="EstiloGeneral">
      <Setter
Property="FrameworkElement.VerticalAlignment"
Value="Center" />

```

```

        </Style>
        <Style x:Key="EstiloBoton"
BasedOn="{StaticResource EstiloGeneral}">
            <Setter Property="Button.Background"
Value="Red" />
        </Style>
        <Style x:Key="EstiloRadio"
BasedOn="{StaticResource EstiloBoton}">
            <Setter
Property="RadioButton.IsChecked" Value="True" />
        </Style>
    </Grid.Resources>
    <Button Style="{StaticResource EstiloBoton}" />
    <RadioButton Style="{StaticResource
EstiloRadio}" />
</Grid>

```

## Plantillas de datos

A diferencia de un estilo una plantilla no persigue asignar una determinada configuración a varias de las propiedades de un objeto, sino sólo a una de ellas. En el caso de las plantillas de datos a la propiedad *ItemTemplate*.

Esta propiedad es donde reside la plantilla que se va a aplicar cuando se dibuje un elemento de un objeto que pueda mostrar varios elementos, como por ejemplo un `ListBox`.

Tomemos un proyecto con estas dos clases definidas:

C#

```

public class Persona
{
    public string Nombre { get; set; }
    public string Apellidos { get; set; }
}

public class ListaPersonas : List<Persona>
{
}

```

Visual Basic.NET

```

Public Class Persona
    Property Nombre As String
    Property Apellidos As String
End Class

Public Class ListaPersonas

```

```
Inherits List(Of Persona)
End Class
```

Y una ventana cuyo XAML es el siguiente:

```
XAML
<Window x:Class="MainWindow"

xmlns="http://schemas.microsoft.com/wfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/wfx/2006/xaml"
xmlns:app="clr-namespace:WpfApplication4"
Title="MainWindow" Height="350" Width="525">
  <Window.Resources>
    <app:ListaPersonas x:Key="Lista">
      <app:Persona Nombre="Pepe"
Apellido="Sanchez" />
      <app:Persona Nombre="Jose"
Apellido="Martinez" />
    </app:ListaPersonas>
  </Window.Resources>
  <Grid>
    <ListBox ItemsSource="{StaticResource Lista}"
  />
  </Grid>
</Window>
```

Obtendremos un `ListBox` con una serie de elementos que nos mostrarán en vez del nombre y los apellidos, el nombre completamente calificado de la clase `Persona`.

Si queremos que esto suceda de otra manera uno de los caminos que podemos tomar es la de la plantilla de elementos. Para ello en la propiedad `ItemTemplate` hay que hacer constar un *DataTemplate*.

```
XAML
<ListBox ItemsSource="{StaticResource Lista}">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <Ellipse Width="10" Height="10" Fill="Red" />
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

Como podemos ver ahora, y tras la aplicación de la plantilla en vez del nombre de la clase lo que aparece son unos círculos rojos. Eso es porque la plantilla contiene eso en un círculo. Pero evidentemente lo que queremos es que aparezcan los nombres y los apellidos, así que como es una plantilla de

elementos, el DataContext de la plantilla será cada uno de los elementos la lista, por lo que ahora podemos introducir expresiones de binding contextualizadas al elemento al que se le aplica la plantilla.

```
XAML
<DataTemplate>
  <StackPanel Orientation="Horizontal">
    <Ellipse Width="10" Height="10" Fill="Red" />
    <TextBlock Text="{Binding Path=Nombre}" />
  </StackPanel>
</DataTemplate>
```

Así ya podemos ver el nombre junto al círculo.

Evidentemente la plantilla al igual que los estilos se puede situar en los recursos. De esta manera si queremos aplicársela a más de un ListBox es mucho más fácil.

Por otro lado, y al igual que los estilos hay una manera para que la plantilla se aplique allá donde haga falta sin necesidad de aplicarla explícitamente, es asignando la propiedad *DataType* que actúa como el TargetType en los estilos, pero en este caso refiriéndose al tipo de dato (clase) del cual depende, en este caso persona.

```
XAML
<Window.Resources>
  ...
  <DataTemplate DataType="{x:Type app:Persona}">
    <StackPanel Orientation="Horizontal">
      <Ellipse Width="10" Height="10" Fill="Red" />
      <TextBlock Text="{Binding Path=Nombre}" />
    </StackPanel>
  </DataTemplate>
</Window.Resources>
```

## Plantillas de controles

Si las plantillas de datos están diseñadas para aplicarse allá donde el dato las necesita, las plantillas de controles están específicamente diseñadas para repintar todo un control. Tomemos un botón:

```
XAML
<ControlTemplate x:Key="plantilla"
  TargetType="Button">
  <Border Width="{TemplateBinding Width}"
```

```

Height="{TemplateBinding Height}" BorderThickness="1"
BorderBrush="{TemplateBinding Foreground}">
  <Grid>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="20*" />
      <ColumnDefinition Width="80*" />
    </Grid.ColumnDefinitions>
    <Rectangle Fill="{TemplateBinding Background}"
      Grid.Column="0" />
    <ContentPresenter Grid.Column="1"
      VerticalAlignment="Center"
      HorizontalAlignment="Center" />
    </Grid>
  </Border>
</ControlTemplate>

```

Si definimos una plantilla como la anterior cambiaremos por completo su aspecto al asignársela mediante el correspondiente binding de recurso.

Cosas interesantes que podemos observar en la plantilla son:

1. El tipo de la plantilla no es *DataTemplate* sino *ControlTemplate*, y ha de ser así porque pretendemos redefinir el control por completo, al menos en su aspecto.
2. Evidentemente existen datos del control que vamos a necesitar en el diseño de la plantilla, por ejemplo sus dimensiones, si no el aspecto se puede ver seriamente afectado. Para ello tenemos un binding especial para plantillas, el *TemplateBinding* que acompañado del nombre de la propiedad nos permite enlazar con cualquier propiedad.
3. Para que este binding funcione necesitamos establecer el *TargetType* correspondiente, porque igual que en los estilos las plantillas sin tipo piensan estar diseñadas para la clase *FrameworkElement*.
4. El control puede tener contenido, que como sabemos puede ser un texto, una imagen o cualquier tipo de panel y con él un número ilimitado de elementos WPF. Para que las plantillas puedan representarlo y no simplemente cubrirlo existe el marcador *ControlTemplate*, que allá donde se ponga reinsertará el contenido original del control.

Sin embargo la diferencia entre estilos y plantillas reside en que estas últimas no pueden ser de manera automática como si lo hacen los estilos. Si prescindimos de la propiedad *x:Key* en una plantilla recibiremos un error en la compilación. Ahora no podemos negar que si esto se pudiera hacer sería sumamente útil, ya que nos permitiría cambiar el aspecto de todos los botones de una ventana o incluso de una aplicación.

Pues no tenemos conseguir este efecto, es sumamente fácil ya que lo único que hay que hacer es crear un estilo que contenga una plantilla. Esto es posible porque a fin de cuentas una plantilla no es más que el contenido de una propiedad, *Template*.

XAML

```
<Style TargetType="Button">
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate x:Key="planti11a"
        TargetType="Button">
        <Border Width="{TemplateBinding Width}"
          Height="{TemplateBinding Height}" BorderThickness="1"
          BorderBrush="{TemplateBinding Foreground}">
          <Grid>
            <Grid.ColumnDefinitions>
              <ColumnDefinition Width="20*" />
              <ColumnDefinition Width="80*" />
            </Grid.ColumnDefinitions>
            <Rectangle Fill="{TemplateBinding Background}"
              Grid.Column="0" />
            <ContentPresenter Grid.Column="1"
              VerticalAlignment="Center"
              HorizontalAlignment="Center" />
          </Grid>
        </Border>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

## Triggers

Si aplicamos el estilo anterior a un botón o conjunto de botones obtendremos una desagradable desilusión, si bien es cierto que el estilo del botón está completamente cambiado hemos perdido en el proceso la interactividad. Ya no parece pulsarse cuando hacemos clic. Una solución a este problema puede ser agregar al estilo lo siguiente:

XAML

```
<Style.Triggers>
  <Trigger Property="IsPressed" Value="True">
    <Setter Property="Background" Value="Red" />
    <Setter Property="Foreground" Value="Red" />
  </Trigger>
</Style.Triggers>
```

Como podemos comprobar al pulsar ahora el botón su color cambia a rojo, pero ¿por qué?

Esto se debe a los *Trigger*, en español disparador, que como vemos establece como si fuera un estilo alternativo o aplicable sólo en determinadas ocasiones, estas que el propio Trigger establece. En el ejemplo cuando la propiedad *IsPressed* está a *True*.

De hecho estos Triggers nos permiten unas posibilidades ilimitadas. Para empezar en cuanto a la detección, además del Trigger tenemos:

- *MultiTrigger* que cuenta con la propiedad *Conditions* que nos permite establecer más de una condición.
- *DataTrigger* que en vez de Property tiene Binding lo que nos permite vincular acciones a datos, por ejemplo cambiar el color de una celda, fila o columna de DataGrid en virtud de un valor.
- *MultiDataTrigger*, misma idea que el MultiTrigger pero aplicada a los DataTriggers.

Además de estos tipos tenemos otro que nos resultará muy útil el *EventTrigger*. Es este un disparador que se vincula a un RoutedEvent, detectando cuando este es elevado por el control. Si lo tratamos de manera separada se debe a que su contenido no puede ser en ningún caso una serie de Setters, ya que sólo admite animaciones a través de lo que se conoce como *Storyboard*.

Toda vez que las animaciones serán tratadas más adelante pospondremos hasta entonces la profundización en estos eventos.

## Recursos compartidos y temas

Como hemos visto a lo largo de todo el capítulo los estilos y las plantillas nos dan un número ilimitado de posibilidades de personalizar nuestras aplicaciones hasta límites insospechados hasta ahora.

Además las personalizaciones no tienen por qué estar vinculadas al objeto en concreto, si no que depositadas en recursos pueden aplicarse a todos los elementos de un determinado ámbito: control, panel, ventana, página,... y aplicación. Si existe un diccionario de recursos vinculado a la aplicación y además es jerárquicamente superior a todo lo demás. Por lo que si definimos algo a ese nivel se aplicará a todos los elementos.



Esto nos da una excelente oportunidad de tratar el aspecto visual de una aplicación como algo unificado y separado incluso del propio diseño del layout de la aplicación.

El problema suele ser que un diccionario de recursos que pretenda contener todos los estilos de una aplicación se volvería un “Mare magnum” imposible de manejar. Es ahí donde entra la propiedad *MergedDictionaries* de la clase *ResourceDictionary*.

Esta propiedad se define como una colección de diccionarios, que nos permite definir un diccionario como suma de otros. Esto se traduce en la práctica en que, como podemos ver, VS nos permite agregar diccionarios de recursos a nuestros proyectos, pero claro sólo uno de ellos es el de la aplicación, ahora bien, con la propiedad *MergedDictionaries* simplemente podemos cargar todos diccionarios que queramos sin sobrecargar el de aplicación. Basta con abrir el archivo *App.xaml* y agregar al *Application.Resources* lo siguiente:

```
XAML
<Application.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary Source="" />
    </ResourceDictionary>
  </Application.Resources>
```

Haciendo constar en *Source* la ruta al archivo de diccionario y por supuesto insertando todos los *ResourceDictionary* que queramos en la propiedad *MergedDictionaries*.

Una última cosa antes de terminar. Es posible también cargar diccionarios desde archivos en tiempo de ejecución. De hecho es posible cargar cualquier XAML. Para ello lo único que hay que hacer es utilizar la clase *XamlReader*.

Esta clase posee un método *Load* que carga un fichero XAML y devuelve un objeto. Leer por tanto un diccionario de recursos externo sería tan sencillo como:

```
C#
XamlReader xml = XamlReader.Create("Estilo.xaml");
Application.Current.Resources =
  (ResourceDictionary)XamlReader.Load(xml);
```

```
Visual Basic.NET
Dim xml As XamlReader = XamlReader.Create("Estilo.xaml")
Application.Current.Resources =
  CType(XamlReader.Load(xml), ResourceDictionary)
```

## Conclusión

Los estilos son una característica básica en HTML que se muestra muy útil al hora de especificar el aspecto visual de un sitio. En WPF tras la adaptación de esta tecnología se va más allá y se integran también las plantillas que nos permiten variabilizar completamente el aspecto visual de una aplicación, permitiendo de este modo precisar, a “posteriori” y sin necesidad de tocar ni una sola coma del código cualquier aspecto gráfico de la aplicación.

## A Continuación

Si algo podemos considerar una ventaja indiscutible en WPF es el apartado gráfico. Podemos crear gráficos 2D y 3D, crear y controlar animaciones, aplicar transformaciones y mucho más. El número de recursos es sumamente amplio y nos aporta una gran capacidad de personalización de aplicaciones. En el siguiente capítulo veremos cómo.

# Gráficos y Animaciones

WPF es un sistema de presentación que a diferencia de *Windows Forms* persigue la creación de interfaces impactantes, con mucha interactividad y además incluyendo contenido audiovisual con tanto ahínco como persigue la creación de aplicaciones de formularios.

Es por esto que se incluye en la API junto con clases como `TextBox` o `Button`, clases como *Rectangle*, *FloatAnimation* o *MediaPlayer*. Y es por esto que las aplicaciones además de servicios de datos y aplicaciones para diseñarlos y tratarlos tienen también, servicios y aplicaciones para crear novedosos interfaces destinados a mejorar la usabilidad, la imagen de marca y la percepción de nuestras aplicaciones.

Todo el soporte gráfico se basa en la clase `Visual`. Esta clase es la abstracción de la cual desciende todos los elementos en WPF.

Esta clase nos aporta soporte para renderizado, gestión de límites, detección de entradas y transformaciones, sin embargo no tiene soporte para eventos, layout, estilos, Data Binding o globalización.

## Graficos y renderizado

El número de elementos a cubrir en este punto valdría en si mismo para escribir un libro, con lo cual el lector no ha de esperar un tratamiento exhaustivo de lo que aquí exponemos, tan solo se pretende esbozar un ejemplo en cada uno de los palos que componen el apartado gráfico que básicamente serían: efectos, *brushes*, *drawings*, geometrías, *shapes* y transformaciones.

Los efectos son algoritmos que aplicados a los diferentes elementos que componen un interfaz WPF consiguen transformar la manera que tienen de presentarse acorde con el carácter del efecto. Es decir, un *DropShadowEffect* aplicado a un botón dibuja el botón como si este tuviera sombra. Toda esta parte de la API sufrió en el Framework 3.5 una intensa remodelación que condujo a la sustitución de los antiguos *BitmapEffect*, no funcionales ya en 4.0, por los *Effect*. Actualmente la API consta de las siguientes clases:

<i>Clase</i>	<i>Descripción.</i>
<i>BlurEffect</i>	<i>Genera un efecto blur.</i>
<i>DropShadowEffect</i>	<i>Genera un efecto sombra.</i>
<i>Effect</i>	<i>Clase base para crear efectos personalizados.</i>
<i>PixelShader</i>	<i>Es un wrapper de alto nivel en torno al soporte de los efectos basados en Pixel Shader.</i>
<i>ShaderEffect</i>	<i>Clase base para crear efectos personalizados basados en Pixel Shader.</i>


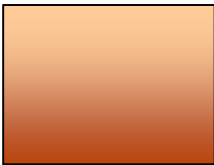
La aplicación de los efectos se hace a través de la propiedad *Effects*, tal como se ve en el ejemplo:

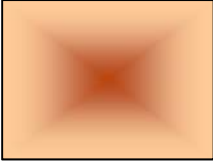
XAML

```
<Button Width="100" Height="30" Name="Button1"
Content="Pul same">
  <Button.Effect>
    <DropShadowEffect />
  </Button.Effect>
</Button>
```

Las *Brushes* o brochas son básicamente una serie de clases cuyos objetos sirven para rellenar de una u otra manera los gráficos. Podemos tener brochas que se encarguen de definir el fondo de un elemento o el trazo de su borde.

Existen varios tipos de brochas:

<i>Brocha</i>	<i>Tipo – ejemplo</i>	<i>Resultado</i>
<i>Sólida, representa un color simple.</i>	<pre> &lt;Rectangle Width="75" Height="75"&gt;   &lt;Rectangle.Fill&gt;     &lt;SolidColorBrush Color="Red" /&gt;   &lt;/Rectangle.Fill&gt; &lt;/Rectangle&gt; </pre>	
<i>Gradiente lineal, establece una brocha entre dos o más colores que transita en función de una recta.</i>	<pre> &lt;Rectangle Width="75" Height="75"&gt;   &lt;Rectangle.Fill&gt;     &lt;LinearGradientBrush&gt;       &lt;GradientStop Color="Yellow" Offset="0.0" /&gt;       &lt;GradientStop Color="Orange" Offset="0.5" /&gt;       &lt;GradientStop Color="Red" Offset="1.0" /&gt;     &lt;/LinearGradientBrush&gt;   &lt;/Rectangle.Fill&gt; &lt;/Rectangle&gt; </pre>	

<i>Gradiente radial, establece una brocha entre dos o más colores que transita en función del radio de una circunferencia.</i>	<pre>&lt;Rectangle Width="75" Height="75"&gt;   &lt;Rectangle.Fill&gt;     &lt;RadialGradientBrush       GradientOrigin="0.75,0.25"&gt;       &lt;GradientStop         Color="Yellow" Offset="0.0"       /&gt;       &lt;GradientStop         Color="Orange" Offset="0.5"       /&gt;       &lt;GradientStop         Color="Red" Offset="1.0" /&gt;     &lt;/RadialGradientBrush&gt;   &lt;/Rectangle.Fill&gt; &lt;/Rectangle&gt;</pre>	
--	---	---

Estas brochas se basan en colores, en concreto los gradientes se establecen entre *GradientStop* que son clases que establecen un color y un *offset*, medida relativa relacionada con el tamaño del objeto, y que pueden ser al menos dos o más.

También existen brochas no vinculadas con el color:

<i>Brocha</i>	<i>Tipo</i>
<i>Brocha de imagen, permite establecer como brocha una imagen.</i>	<pre>&lt;Rectangle Width="75" Height="75"&gt;   &lt;Rectangle.Fill&gt;     &lt;ImageBrush ImageSource="imagen.jpg"   /&gt;   &lt;/Rectangle.Fill&gt; &lt;/Rectangle&gt;</pre>

<p><i>Brocha de tipo Drawing, permite establecer un dibujo como imagen.</i></p>	<pre> &lt;Rectangle Width="75" Height="75"&gt;   &lt;Rectangle.Fill&gt;     &lt;DrawingBrush&gt;       &lt;DrawingBrush.Drawing&gt;         &lt;DrawingGroup&gt;           &lt;GeometryDrawing Brush="White"&gt;             &lt;GeometryDrawing.Geometry&gt;               &lt;RectangleGeometry                 Rect="0,0,100,100" /&gt;             &lt;/GeometryDrawing.Geometry&gt;           &lt;/GeometryDrawing&gt;         &lt;/DrawingBrush.Drawing&gt;       &lt;/DrawingBrush&gt;     &lt;/Rectangle.Fill&gt;   &lt;/Rectangle&gt; </pre>
<p><i>Brocha visual, que permite establecer como brocha un elemento visual, es decir cualquier control como un botón o un MediaElement, que nos permitía este último establecer como brocha un vídeo.</i></p>	<pre> &lt;Rectangle Width="75" Height="75"&gt;   &lt;Rectangle.Fill&gt;     &lt;VisualBrush TileMode="Tile"&gt;       &lt;VisualBrush.Visual&gt;         &lt;MediaElement Source="video.wmv" /&gt;       &lt;/VisualBrush.Visual&gt;     &lt;/VisualBrush&gt;   &lt;/Rectangle.Fill&gt; &lt;/Rectangle&gt; </pre>

*Drawings* son elementos que describen elementos visuales:

- *GeometryDrawing* – dibuja una shape.
- *ImageDrawing* – dibuja una imagen.
- *GlyphRunDrawing* – dibuja texto.
- *VideoDrawing* – reproduce ficheros de audio y video.

- *DrawingGroup* – dibuja un grupo de dibujos combinados en uno compuesto.

Los objetos *Drawing* repiten muchas de las características que tienen otros elementos, como por ejemplo determinados controles. ¿Cuándo, por tanto, usar unos y otros? En general los objetos *Drawing* son elementos que favorecen el rendimiento a costa de perder características de control, por ejemplo control de foco, lógica de layout, etc.

Las *shapes* son rutinas de dibujo que representan formas simples o complejas. Todas ellas derivan de la clase abstracta *Shape*, que define la propiedad *Fill*, brocha de fondo y las propiedades *stroke*, que te permiten definir el patrón y la brocha de borde.

<i>Rectangle</i>	<i>Dibuja un rectángulo</i>	<code>&lt;Rectangle Width="30" Height="30" /&gt;</code>
<i>Ellipse</i>	<i>Dibuja una elipse</i>	<code>&lt;Ellipse Width="30" Height="30" /&gt;</code>
<i>Line</i>	<i>Dibuja una línea</i>	<code>&lt;Line X1="0" Y1="0" X2="1" Y2="2" /&gt;</code>
<i>Polyline</i>	<i>Dibuja una línea multipunto</i>	<code>&lt;Polyline Points="10,10,1,1" /&gt;</code>
<i>Poligon</i>	<i>Idem a la polyline pero cierra con el primer y último punto</i>	<code>&lt;Polygon Points="10,10,1,1,30,15" /&gt;</code>



<i>Path</i>	<i>Shape que se establece como un conjunto de geometrías y o segmentos combinados o no.</i>	<pre> &lt;Path Stroke="Black" StrokeThickness="1" Fill="#CCCCFF"&gt;    &lt;Path.Data&gt;      &lt;GeometryGroup FillRule="EvenOdd"&gt;        &lt;LineGeometry StartPoint="10,10" EndPoint="50,30" /&gt;        &lt;EllipseGeometry Center="40,70" RadiusX="30" RadiusY="30" /&gt;        &lt;RectangleGeometry Rect="30,55 100 30" /&gt;      &lt;/GeometryGroup&gt;    &lt;/Path.Data&gt;  &lt;/Path&gt; </pre>
-------------	---	--

Una transformación es un mecanismo para mapear puntos en un espacio de coordenadas a otro. Esto se realiza mediante una matriz de transformación. Manipulando las matrices se pueden producir efectos como rotaciones, traslaciones.

Aunque se pueden manipular las matrices a bajo nivel WPF aporta clases que encapsulan las transformaciones más comunes:

<i>Clase</i>	<i>Descripción</i>
<i>RotateTransform</i>	<i>Rota un elemento un ángulo específico.</i>
<i>ScaleTransform</i>	<i>Escala un elemento en función de los factores ScaleX y ScaleY.</i>
<i>SkewTransform</i>	<i>Transformación de sesgado en función de dos ángulos AngleX y AngleY.</i>
<i>TranslateTransform</i>	<i>Mueve un elemento según dos coordenadas X e Y.</i>

Para utilizar una transformación se la asignamos a un `FrameworkElement` en una de las dos propiedades de transformación:

- *LayoutTransform* – una transformación que se aplica antes de la situación del control del proceso de layout. Este proceso trabaja sobre el tamaño y la posición el elemento una vez transformado.
- *RenderTransform* – Esta modificación trabaja sobre la apariencia del elemento pero se aplica antes del proceso de layout. Este sitúa el control ya transformado.

XAML

```
<Border Margin="30"
    HorizontalAlignment="Left" VerticalAlignment="Top"
    BorderBrush="Black" BorderThickness="1" >
    <StackPanel Orientation="Vertical">
        <Button Content="A Button" Opacity="1" />
        <Button Content="Rotated Button">
            <Button.RenderTransform>
                <RotateTransform Angle="45" />
            </Button.RenderTransform>
        </Button>
        <Button Content="A Button" Opacity="1" />
    </StackPanel>
</Border>
```

## Renderizado 3-D

WPF nos permite crear funcionalidad 3D. Esta funcionalidad está dirigida a crear entornos más ricos, complejas representaciones de los datos o mejorar la experiencia de usuario, no para crear aplicaciones de alto rendimiento, como por ejemplo juegos.

Los gráficos 3D se encapsulan en un elemento llamado *Viewport3D*. Este elemento es tratado como un elemento visual 2D que es una ventana al mundo 3D.

En el sistema bidimensional el origen se encuentra en el punto superior izquierdo del contenedor, sin embargo en el 3D en el centro del área. La coordenada X crecerá hacia la derecha y decrecerá hacia la izquierda, la Y positiva hacia arriba y negativa hacia abajo y la Z positiva desde el centro hacia el observador.

Este espacio se define como un marco de referencia para los objetos (espacio mundo), en el se pueden depositar cámaras, luces u objetos, esto últimos

pueden ser visibles desde un determinado punto de vista (cámara) o no. El punto de vista no cambia la posición ni modifica los objetos.

Una escena 3D en realidad es una representación 2D de un espacio 3D, en función de un punto de vista. Este se puede especificar mediante una cámara. Si asumimos la cámara como el punto de vista de un espectador, también podemos decir que es la manera que tiene ese espectador de percibir una escena.

Una *ProjectionCamera* permite especificar proyecciones y las propiedades de estas para cambiar la perspectiva del observador, mientras que una *PerspectiveCamera* nos permite restringir la escena con un punto de desvanecimiento, posición, dirección de encuadre. Además cuenta con dos planos el *NearPlaneDistance* y *FarPlaneDistance* que establecen los límites de proyección, de forma que los elementos que se encuentren fuera del espacio delimitado por ellos no son visibles.

La *OrthographicCamera* especifica una proyección ortogonal como otras cámaras establece posición, dirección, etc. pero no tiene efecto horizonte, esto significa que los elementos no parecen empequeñecer cuando se alejan.

La clase abstracta *Model3D* que representa objetos 3D, necesarios para crear escenas. WPF soporta *GeometryModel3D* que nos permite establecer un modelo en función de una primitiva, actualmente sólo se soporta la clase *MeshGeometry3D* que necesita especificar lista de posiciones (vértices de triángulos) y lista de vértices (agrupación de esto de 3 en 3).

También se especifican coordenadas de texturas para situar sobre la forma y normales que nos permiten especificar la cara de la faceta a la que se aplica la textura, así como calcular la incidencia de la luz sobre una cara concreta del objeto.

XAML

```
<GeometryModel 3D>
  <GeometryModel 3D. Geometry>
    <MeshGeometry3D
      Posi tions="-1 -1 0  1 -1 0  -1 1 0  1 1
0"
      Normal s="0 0 1  0 0 1  0 0 1  0 0 1"
      TextureCoordi nates="0 1  1 1  0 0  1 0
"
      Tri angl eÍ ndi ces="0 1 2  1 3 2" />
    </GeometryModel 3D. Geometry>
  <GeometryModel 3D. Materi al >
    <Di ffuseMateri al >
      <Di ffuseMateri al . Brush>
        <Sol i dCol orBrush Col or="Cyan"
Opaci ty="0. 3"/>
```

```

        </DiffuseMaterial.Brush>
    </DiffuseMaterial>
</GeometryModel3D.Material>
<!-- Translate the plane. -->
<GeometryModel3D.Transform>
    <TranslateTransform3D
        OffsetX="2" OffsetY="0" OffsetZ="-1"    >
    </TranslateTransform3D>
</GeometryModel3D.Transform>
</GeometryModel3D>

```

Los objetos para ser visibles necesitan materiales además de estar iluminados. Los modelos para ello pueden aplicar un descendiente de la clase abstracta *Material*, cada una de las cuales provee de un aspecto concreto en función del material además de proveer una propiedad para especificar una brocha:

*DiffuseMaterial* especifica que el material se aplica iluminado difusamente, no refleja la luz.

*SpecularMaterial* en este caso el material refleja la luz.

*EmissiveMaterial*, este tipo de material emite luz del coincidente con la textura.

Las escenas también han de ser iluminadas, para ello contamos con diferentes tipos de luces.

*AmbientLight*: Luz ambiente que ilumina todos los objetos por igual.

*DirectionalLight*: Iluminan como una fuente distante, con dirección pero no localización.

*PointLight*: Iluminan como un origen cercano, tiene posición pero no dirección, y un rango fuera del cual ya no ilumina.

*SpotLight*: Hereda de *PointLight* disponiendo de posición y dirección.

XAML

```

<ModelVisual3D.Content>
    <AmbientLight Color="#333333" />
</ModelVisual3D.Content>

```

# Animación

Las animaciones son pequeños algoritmos que en un espacio de tiempo transitan desde un valor a otro cuyo tipo dependerá del tipo de la animación. Así una animación numérica puede transitar entre 1 y 2 en un segundo.

La utilidad de una animación es que se puede conectar con una propiedad de un objeto, cuyo tipo sea compatible y de manera automática esta se verá modificada transitando entre los valores de la animación en el tiempo especificado.

Un StoryBoard es una agrupación de animaciones que permite crear efectos complejos por la suma de estas. Además el StoryBoard crea unas propiedades attached que nos permiten vincular las animaciones con los diferentes elementos y sus correspondientes propiedades.

XAML

```
<Storyboard>
  <DoubleAnimation From="0" To="5"
    Duration="00:00:00.5" Storyboard.TargetName="effect"
    Storyboard.TargetProperty="ShadowDepth" />
</Storyboard>
```

En el ejemplo podemos ver como una animación que dura apenas medio segundo y que transita entre 0 y 5 se aplica a la propiedad ShadowDepth perteneciente a un elemento llamado effect, que es un ShadowEffect.

Existen decenas de tipos de animaciones e incluso varias versiones de una misma animación. Double, Single, Point, Point3D, String, etc.. son tipos que soportan animaciones, en algunos casos sólo del tipo UsingKeyFrames.

Estas animaciones se diferencian de las normales en que contienen una serie de KeyFrames, valores intermedios, que nos permiten establecer puntos de inicio y final intermedios. Con ellos logramos cosas como transiciones no lineales o incluso y según el caso mejoras de rendimiento.

La clase StoryBoard tiene una serie de métodos que desde programa nos permite ejecutarla, pararla, cancelarla, etc. Pero lo mejor de los StoryBoard es que existen una serie acciones de trigger que nos permiten hacer lo mismo.

Si recordamos cuando hablábamos de los trigger mencionábamos uno en especial, EventTrigger que nos permite enlazar con las siguientes acciones:

<i>Acción de Trigger</i>	<i>Efecto</i>
<i>BeginStoryboard</i>	<i>Arranca un Storyboard</i>
<i>ResumeStoryboard</i>	<i>Restaura un Storyboard después de parado</i>
<i>PauseStoryboard</i>	<i>Para un Storyboard</i>

Esto es sumamente útil porque nos permite crear StoryBoards sin necesidad de incluir código en nuestra aplicación. Esto produce que los diseñadores y programadores puedan trabajar conjuntamente de una manera simultánea, a tiempo que se reduce la complejidad del código.

XAML

```
<Button x:Name="button2" Width="100" Height="30"
Content="Pulsame">
  <Button.Triggers>
    <EventTrigger RoutedEvent="Button.Click">
      <BeginStoryboard>
        <Storyboard>
          <DoubleAnimation From="1" To="0"
Storyboard.TargetName="button2"
Storyboard.TargetProperty="Opacity" />
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger>
  </Button.Triggers>
</Button>
```

## Visual State Manager

Además de los Triggers existe otra manera de lanzar una animación y esta es Visual State Manager. Esta forma fue originalmente desarrollada para Silverlight, ya que el modelo de objetos de este no permite los Triggers, como mecanismo sencillo para hacer transitar visualmente los objetos desde un punto a otro. El mecanismo se reveló tan sencillo y fue tan bien acogido por la comunidad de desarrolladores que finalmente ha realizado el camino hacia la tecnología padre, a diferencia de lo que normalmente suele pasar que las novedades se incorporan primero a WPF o son tan específicas que finalmente permanecen en Silverlight.

En todo caso utilizar el Visual State Manager es tan sencillo como anidar un nodo `VisualStateManager.VisualStateGroup`, dentro de este definir un `VisualStateGroup` que puede acoger diferentes estados en nodos `VisualState`, a los cuales se les otorga un nombre y que básicamente acogen un `Storyboard` que se ejecutará cuando le comuniquemos al Visual State Manager que queremos que el objeto transite a un determinado estado.

XAML

```
<Button Width="100" Height="30" Name="Button1"
Content="Pulsame">
  <VisualStateManager.VisualStateGroups>
    <VisualStateGroup>
      <VisualState Name="normal">
        <Storyboard>
          <DoubleAnimation BeginTime="00:00:00.5" From="1"
To="0" Duration="00:00:00.5"
Storyboard.TargetName="Trans"
Storyboard.TargetProperty="ScaleX" />
        </Storyboard>
      </VisualState>
      <VisualState Name="sobre">
        <Storyboard>
          <DoubleAnimation BeginTime="00:00:00.5" From="0"
To="1" Duration="00:00:00.5"
Storyboard.TargetName="Trans"
Storyboard.TargetProperty="ScaleX" />
        </Storyboard>
      </VisualState>
    </VisualStateGroup>
  </VisualStateManager.VisualStateGroups>
<Button.RenderTransform>
  <ScaleTransform x:Name="Trans" />
</Button.RenderTransform>
</Button>
```

En el ejemplo anterior se crean dos estados en un botón que acogen sendas animaciones. La ejecución de esas animaciones tendrá lugar cuando voluntariamente informemos al Visual State Manager de que queremos que suceda invocando el método *GoToElementState* o *GoToState* si los estados están definidos en una plantilla.

C#

```
VisualStateManager.GoToElementState(Button1, "sobre",
true);
```

Visual Basic.NET

```
VisualStateManager.GoToElementState(Button1, "sobre",
true)
```

## Tratamiento de medios

Como ya vimos en capítulos anteriores existe un amplio soporte para medios en WPF. Tenemos controles y otro tipo de elementos que nos permiten reproducir desde imágenes estáticas hasta vídeos o audio.

Las imágenes incluyen iconos, fondos e incluso partes de animaciones. El trabajo básicamente se realiza a través de la clase `Image` que nos permite introducir una imagen en marcado o hacer tratamiento no visual de ella.

Ejemplos de las capacidades de esta clase pueden ser:

### 1. Cambio de formato de pixel.

XAML

```
<Image Width="200" >
  <Image.Source>
    <FormatConvertedBitmap Source="imagen.jpg"
      DestinationFormat="Gray4" />
  </Image.Source>
</Image>
```

### 2. Aplicación de una máscara de corte.

XAML

```
<Image Width="200" Source="imagen.jpg">
  <Image.Clip>
    <EllipseGeometry Center="75,50" RadiusX="50"
      RadiusY="25" />
  </Image.Clip>
</Image>
```

### 3. Diferentes formas de adaptación de la imagen al espacio disponible.

XAML

```
<Image
  Source="imagen.jpg"
  Stretch="Fill" />
```

En cuanto al vídeo y al audio disponemos de una clase, `MediaElement`, que nos permite de manera sencilla reproducir vídeos, archivos de audio e incluso streaming.

XAML

```
<MediaElement Source="Clock.avi" Width="450"
  Height="250" />
```



## Conclusión

Casi todo en WPF está orientado a potenciar las capacidades gráficas. Es este sin duda el principal objetivo de esta tecnología y el factor diferenciador. En el capítulo hemos intentado dar una idea general de lo que esta API permite hacer en este campo.

## A continuación

El tratamiento documental es algo que habitualmente tiene mucha importancia en las aplicaciones pero que solemos pasar desapercibido a la hora de evaluar un API de programación. Comúnmente es debido a que suelen ser servicios externos, generadores de informes y demás los que asumen esta responsabilidad.

Sin embargo WPF incorpora un soporte documental que nos permite ciertamente mucha flexibilidad, a continuación veremos cómo.



# Documentos

A diferencia de las versiones de Windows, WPF contiene un soporte para la creación, lectura y gestión de la seguridad de documentos de alta calidad, así como las clases necesarias para su tratamiento.

## Documentos en WPF

WPF soporta dos tipos de documentos, "flow documents" y "fixed documents". Los primeros están indicados para su uso en aquellas aplicaciones que pretenden mostrar documentos agradables y adaptables para su lectura, los segundos en cambio están más orientados para la creación de aplicaciones que busca precisión en la reproducción de documentos, WYSIWYG o impresión.

WPF nos aporta una serie de controles que facilitan el uso y la muestra de ambos tipos de documentos. En el caso de los documentos fijos disponemos del control `DocumentViewer` y en el caso de los de flujo `FlowDocumentReader`, `FlowDocumentPageViewer` y `FlowDocumentScrollViewer`.

El control `DocumentViewer` nos permite operaciones como impresión, copia, zoom y búsqueda además de soportar, como muchos otros controles WPF, la aplicación de un nuevo estilo completo o parcial que adapte el control al estilo

visual de nuestra aplicación. La restricción principal de este control es que es de sólo lectura, la edición no está soportada.

En el caso de los documentos no fijos disponemos de más tipos de controles según necesitemos más o menos características y queramos balancear el peso del control.

El control `FlowDocumentReader` permite que el usuario cambie dinámicamente el modo de visualización, más modestos `FlowDocumentPageViewer` y `FlowDocumentScrollView` nos aportan visualizaciones fijas, como página o una simple cinta sin fin, que se pueden adaptar a otras necesidades y son más baratos en término de recursos.

XAML

```
<FlowDocumentReader>
  <FlowDocument>
    <Paragraph>
      <Bold>Algo de texto en un párrafo</Bold>
      y algo de texto que no está en negrita.
    </Paragraph>
    <List>
      <ListItem>
        <Paragraph>Elemento 1</Paragraph>
      </ListItem>
      <ListItem>
        <Paragraph>Elemento 2</Paragraph>
      </ListItem>
      <ListItem>
        <Paragraph>Elemento 3</Paragraph>
      </ListItem>
    </List>
  </FlowDocument>
</FlowDocumentReader>
```

Como se puede ver en el ejemplo el documento de flujo se está especificando también como sintaxis de marcado. A continuación veremos un poco más en detalle este tipo de documentos.

## Documentos de flujo

Un documento de flujo esta diseñado esencialmente para mejorar la experiencia de lectura, para ello se ha de adaptar al espacio disponible, la resolución o preferencias del usuario.

Estos documentos pueden ser consumidos por los controles específicos además de por `RichTextBox`, que nos aporta también la capacidad de editar estos documentos.

XAML

```
<RichTextBox>
  <FlowDocument>
    <Paragraph>
      Esto es un contenido editable.
    </Paragraph>
  </FlowDocument>
</RichTextBox>
```

Los elementos más comúnmente usados son los siguientes:

<i>Elemento</i>	<i>Uso</i>
<i>Paragraph</i>	<i>Delimitador de párrafo.</i>
<i>Section</i>	<i>Agrupación de varios párrafos.</i>
<i>BlockUIContainer</i>	<i>Contenedor de controles de UI dentro de un documento.</i>
<i>List</i>	<i>Elemento lista que contiene ListItems.</i>
<i>Tabla</i>	<i>Elemento tabla que contiene TableRow.</i>
<i>TableRow</i>	<i>Fila de una tabla que contiene TableCell.</i>
<i>TableCell</i>	<i>Celda de una tabla.</i>
<i>ListItem</i>	<i>Elemento unitario de una lista.</i>
<i>Run</i>	<i>Elemento para contener texto sin formato.</i>
<i>HyperLink</i>	<i>Hiperenlace contenido en el documento.</i>
<i>Bold</i>	<i>Estilo negrita.</i>
<i>Italic</i>	<i>Estilo cursiva.</i>
<i>Underline</i>	<i>Estilo subrayado.</i>
<i>InlineUIContainer</i>	<i>Contenedor de controles de UI dentro de un documento, dentro de una línea de texto.</i>

<i>Floater</i>	<i>Elemento destinado embeber contenido como imágenes o anuncios, puede paginar pero no se puede posicionar ni redimensionar más allá de la columna de texto que ocupa.</i>
<i>Figure</i>	<i>Elemento destinado embeber contenido como imágenes o anuncios, es posicionable, dimensionable pero no puede paginar.</i>
<i>LineBreak</i>	<i>Salto de línea.</i>

## Serialización y almacenaje de documentos

WPF es capaz de manejar un documento en memoria, pero una de las características claves del manejo de documentos es también guardarlos y cargarlos desde un soporte de almacenamiento. Esto es lo que llamamos serialización.

El proceso de cargar o guardar documentos ha de ser transparente a la aplicación, en general esta debe llamar a métodos Read o Write de un serializador, sin tener que pensar en el formato.

Además las aplicaciones proveen habitualmente de múltiples formatos para leer y escribir documentos. Esa es la razón principal por la cual la arquitectura de serializadores documentales de WPF es modular.

Las características principales que los serializadores tienen o han de tener son:

- Acceso directo a los objetos del documentos a través de un interfaz de alto nivel.
- Operatividad síncrona y asíncrona.
- Soporte para plug-ins y mejoras de las capacidades:
  - Amplio acceso para su uso desde cualquier aplicación NET.
  - Mecanismo de descubrimiento de plug-ins.
  - Despliegue e instalación sencillas.
  - Soporte de interfaz para opciones personalizadas.

Por defecto el serializador instalado es XPS, que es un mecanismo nativo para crear documentos en WPF y en Windows a partir de Vista.

Además de este serializador también tenemos la oportunidad de desarrollar los nuestros propios o adquirirlos a un fabricante.

Cuando esto sucede el sistema dispone de varios formatos y necesitamos poder conocer cuales están disponibles.

```
C#
    SerializerProvider serializerProvider = new
    SerializerProvider();
    foreach (SerializerDescriptor serializerDescriptor in
    serializerProvider.InstalledSerializers)
    {
        if (serializerDescriptor.IsLoadable)
        {
        }
    }
}
```

```
Visual Basic.NET
Dim serializerProvider As New SerializerProvider()
For Each serializerDescriptor As SerializerDescriptor
In serializerProvider.InstalledSerializers

    If serializerDescriptor.IsLoadable Then
    End If
Next
```

Para localizar los plug-ins simplemente creamos una clase *SerializerProvider* recorriendo la colección *InstalledSerializers* y asegurándonos de que se puede usar a través del método *IsLoadable*. Tras lo cual no resta sino guardar el archivo:

```
C#
    Stream package = File.Create(fileName);
    SerializerWriter serializerWriter =
    serializerProvider.CreateSerializerWriter(
    selectedPlugin, package);
    IDocumentPaginatorSource idoc = flowDocument as
    IDocumentPaginatorSource;
    serializerWriter.Write(idoc.DocumentPaginator, null);
    package.Close();

Visual Basic.NET
Dim package As Stream = File.Create(fileName)
Dim serializerWriter As SerializerWriter =
serializerProvider.CreateSerializerWriter(selectedPlug
In, package)
Dim idoc As IDocumentPaginatorSource =
```

```
TryCast(flowDocument, IDocumentPaginatorSource)
serializer.Writer.Write(idoc.DocumentPaginator,
Nothing)
package.Close()
```

Para ello y debido a la curiosa estructura de los archivos XPS hemos de crear un paquete que junto con el plug-in nos permite crear el serializador. Por último serializamos el documento.

## Anotaciones

Las anotaciones son notas o comentarios que se añaden a los documentos y que nos permiten marcar información o resaltar puntos de interés.

Existe dos tipos de notas *Sticky Notes* y *Highlights*.

Las primeras contienen información en un papel que se "pega" al documento, además de cumplir con su función nos permiten mejorar la experiencia de usuario en dispositivos como Tablet o Tablet PC.

C#

```
AnnotationService _annotationService = new
AnnotationService(docViewer);
FileStream _annotationStream = new FileStream(
_annotationStorePath, FileMode.OpenOrCreate,
FileAccess.ReadWrite);
XmlStreamStore _annotationStore = new
XmlStreamStore(_annotationStream);
_annotationService.Enable(_annotationStore);
```

Visual Basic.NET

```
Dim _annotationService As New AnnotationService(docViewer)
Dim _annotationStream As New FileStream(_annotationStorePath,
FileMode.OpenOrCreate, FileAccess.ReadWrite)
Dim _annotationStore As New XmlStreamStore(_annotationStream)
_annotationService.Enable(_annotationStore)
```

Para utilizar estas anotaciones se ha de crear un servicio asociado a un documento y un almacén de notas (*XmlStreamStore*), para luego habilitar este servicio vinculado al almacén.

Los Highlights son dibujos que utilizamos para resaltar partes del texto, similar a pintar con un rotulador de marcado.



La creación y borrado de anotaciones, sticky notes y highlights se realiza a través de comandos de la clase *AnnotationService*, conveniente porque además de invocar por código se puede vincular a menús y botones.

## Conclusión

El tratamiento de documentos es también una de las características de WPF. Se nos permite crear documentos adaptados a lectura o impresión, con servicios que nos permiten partiendo de los primeros crear los segundos. Por último tenemos la capacidad de crear notas para comentar documentos ya existentes.

Esto incrementa considerablemente las capacidades de los dispositivos programados con WPF, ya que nos permite integrar un buen conjunto de tecnologías que van desde los e-Books al tratamiento documental avanzado.

## A continuación

Otra de las características de cualquier plataforma de desarrollo es la extensibilidad. En WPF podemos extender la plataforma de muchas maneras pero la más común es la de crear nuevos controles. En el siguiente capítulo veremos cómo.



# Controles

Siguiendo un modelo que no es nuevo, en WPF, podemos optar por dos caminos diferentes a la hora de crear controles. De un lado se encuentran los controles de usuario, fáciles de desarrollar pero con un nivel de encapsulamiento pequeño o los controles personalizados, más complicados pero mejor resueltos a nivel de interfaz de programación.

Sin embargo a diferencia de otros mecanismos de programación no es tan frecuente el desarrollo de nuevos controles, ya que WPF proporciona múltiples mecanismos para personalizar controles.

## Controles de usuario

Los controles de usuario son una seria opción antes de introducirse en el farragoso mundo de la creación de controles personalizados.

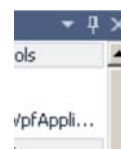
Y lo son porque la complejidad de desarrollo es relativamente baja, pudiendo por otro lado insertarlos en una librería sin mayores problemas e insertarlos en aquellas aplicaciones que necesitemos.

Para crear un control de usuario lo único que debemos hacer es agregarlo a nuestro proyecto, y obtendremos un diseñador muy similar a una ventana, sólo que en este diseñaremos un control en vez de toda una ventana.

XAML

```
<UserControl x:Class="UserControl1"
  xmlns="http://schemas.microsoft.com/wpf/2006/xaml"
  xmlns:x="http://schemas.microsoft.com/wpf/2006/xaml"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:d="http://schemas.microsoft.com/expression/bl end
/2008"
  mc:Ignorable="d"
  d:DesignHeight="300" d:DesignWidth="300">
  <Grid>
  </Grid>
</UserControl>
```

Tras la compilación el control, como control personalizado, aparecerá instalado en nuestra ToolBox y ya sólo nos restaría arrastrarlo como si de un control convencional se tratase.



## La jerarquía de objetos en WPF

Bien, si a pesar de todas las opciones para no hacerlo seguimos pensando que desarrollar un nuevo control es el camino a tomar, entonces debemos conocer la jerarquía de objetos básica de WPF ya que afinaremos más a la hora de desarrollar el nuevo control, evitando funcionalidad innecesaria o incluyendo funcionalidad requerida que, de otra forma, deberíamos implementar.

Todos los objetos descenden de una misma clase que implementa las funcionalidades



básicas llamada *FrameworkElement*, del que descienden tres clases que dividen las intenciones del desarrollador, *Panel*, en el caso de que queramos distribuir otros componentes mediante una lógica predeterminada, *Decorator* si lo que queremos es un elemento que amplíe funcionalidades a elementos ya existentes y *Control* si queremos crear una clase que sea capaz de interactuar con el usuario.

Si nos decidimos por el *Control*, cosa más habitual, entonces deberíamos establecer si queremos un *ContentControl*, que sería la base para la mayoría de los controles habituales, un control capaz de tener contenido o un *ItemControl* que es un control que sólo puede existir dentro de otro. Una vez decidido deberíamos estudiar la rama en cuestión a fin de descender de la clase más apropiada.

## Pasos para desarrollar un nuevo control

Como ya dijimos el primer paso para crear un nuevo control es seleccionar la clase base de la que vamos a descender el control. Después debemos acometer los siguientes pasos:

Definir la API, hemos de pensar cuidadosamente que funcionalidad y de que manera se va a exponer.

Crear el interfaz, crearemos este interfaz creando la clase con los miembros apropiados.

Template, crearemos la plantilla, necesaria como ya explicamos en el capítulo de plantillas.

Funcionalidad, completaremos la funcionalidad, que no es posible abordar hasta ahora ya que está íntimamente ligada a la plantilla.

El interfaz del control estará definido en función de:

- Propiedades, dependientes o attached, también normales.
- Eventos, eventos enrutados que nos permiten relacionarnos con los demás elementos del interfaz.
- Comandos, con las funcionalidades preestablecidas e importantes preparadas o al menos previstas.

Sin embargo los métodos son parte importante en todas las clases, en los controles WPF sólo sirven como apoyo, ya que estos como hemos podido comprobar a lo largo del libro no se usan salvo a la manera tradicional, desde código.

C#

```
public class Comentario : ContentControl
{
    static Comentario()
    {
        DefaultStyleKeyProperty.OverrideMetadata(typeof(Comentari o), new
        FrameworkPropertyMetadata(typeof(Comentari o)));

        CommandManager.RegisterClassCommandBinding(typeof(Come
        ntari o), new CommandBinding(_ocul tar, ocul tando));

        CommandManager.RegisterClassInputBinding(typeof(Coment
        ari o), new InputBinding(_ocul tar, new
        MouseGesture(MouseAction.LeftClick)));
    }
    public Brush ComentarioBackground
    {
        get { return
        (Brush)GetValue(Comentari oBackgroundProperty); }
        set {
        SetValue(Comentari oBackgroundProperty, value); }
    }
    public static readonly DependencyProperty
    Comentari oBackgroundProperty =
        DependencyProperty.Register("Comentari oBackground",
        typeof(Brush), typeof(Comentari o), new
        UIPropertyMetadata(Brushes.Aqua));
    static RoutedCommand _ocul tar = new
    RoutedCommand();
    static private void ocul tando(object sender,
    ExecutedRoutedEventArgs e)
    {
        Comentario obj = (Comentari o)sender;
        object sb =
        (obj.Template.FindName("panel", obj) as
        Grid).Resources["Ocul tar"];
        (sb as Storyboard).Begin();
    }
}
```

Visual Basic.NET

```
Public Class Comentario
    Inherits ContentControl
    Shared Sub New()
```

```

DefaultStyleKeyProperty.OverrideMetadata(GetType(Comentario), New
FrameworkPropertyMetadata(GetType(Comentario)))
    CommandManager.RegisterClassCommandBinding(GetType(Comentario), New CommandBinding(_ocular, AddressOf
ocularando))
    CommandManager.RegisterClassInputBinding(GetType(Comentario), New InputBinding(_ocular, New
MouseGesture(MouseAction.LeftClick)))
    End Sub
    Public Property ComentarioBackground() As Brush
        Get
            Return
DirectCast(GetValue(ComentarioBackgroundProperty),
Brush)
        End Get
        Set
            SetValue(ComentarioBackgroundProperty, value)
        End Set
    End Property
    Public Shared ReadOnly
ComentarioBackgroundProperty As DependencyProperty =
DependencyProperty.Register("ComentarioBackground",
GetType(Brush), GetType(Comentario), New
UIPropertyMetadata(Brushes.Aqua))

    Shared _ocular As New RoutedCommand()

    Private Shared Sub ocularando(sender As Object, e
As ExecutedRoutedEventArgs)
        Dim obj As Comentario = DirectCast(sender,
Comentario)
        Dim sb As Object =
TryCast(obj.Template.FindName("panel", obj),
Grid).Resources("Ocular")
        TryCast(sb, Storyboard).Begin()
    End Sub
End Class

```

En el ejemplo anterior vemos como hemos creado una clase que deriva de Content control con una propiedad dependiente que nos servirá para establecer la brocha a aplicar al control.

También tenemos un comando que lanza una StoryBoard. ¿Pero dónde está definida esta StoryBoard?

La respuesta es sencilla cuando creamos un control personalizado automáticamente Visual Studio crea una carpeta dentro del proyecto llamada *Themes*, en ella se introduce un archivo *Generic.xaml*.

Este archivo almacena la plantilla y el estilo del control y en ella como se puede ver hemos definido la StoryBoard:

XAML

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/wfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/wfx/2006/xaml"
    xmlns:local="clr-namespace:ComentarioLib"
    <Style TargetType="{x:Type local:Comentario}">
        <Setter Property="Template">
            <Setter.Value>
                <ControlTemplate TargetType="{x:Type local:Comentario}">
                    <Grid x:Name="panel">
                        <Grid.Resources>
                            <Storyboard
                                x:Key="Ocultar">
                                    <DoubleAnimation
                                        From="1" To="0" Duration="00:00:02"
                                        Storyboard.TargetName="panel"
                                        Storyboard.TargetProperty="Opacity" />
                                </Storyboard>
                            </Grid.Resources>
                            <Border
                                Background="{TemplateBinding ComentarioBackground}"
                                CornerRadius="24"
                                BorderBrush="Black"
                                Margin="0,0,0,30" Padding="24"
                                BorderThickness="2">
                                    <ContentPresenter />
                                </Border>
                            <Grid
                                VerticalAlignment="Bottom" HorizontalAlignment="Right"
                                Margin="0,0,30,0">
                                    <Polygon
                                        Points="10,0,40,0,0,30" Fill="{TemplateBinding ComentarioBackground}"
                                        VerticalAlignment="Bottom" HorizontalAlignment="Right" />
                                    <Line X1="10" Y1="0"
                                        X2="0" Y2="30" Stroke="Black" StrokeThickness="2" />
                                    <Line X1="10" Y1="0"
                                        X2="40" Y2="0" Stroke="{TemplateBinding ComentarioBackground}" StrokeThickness="3" />
                                    <Line X1="40" Y1="0"
                                        X2="0" Y2="30" Stroke="Black" StrokeThickness="2" />
                                </Grid>
                            </Grid>
                        </Grid>
                    </ControlTemplate>
                </Setter.Value>
            </Setter>
        </Style>
```



```

        </Control Template>
    </Setter.Value>
</Setter>
</Style>
</ResourceDictionary>

```

Importante el hecho de que en el `ResourceDictionary` importamos el espacio de nombres de la aplicación, para que así el `TargetType` pueda hacer un binding al tipo de la clase que definimos.

## Conclusión

Aunque la extensibilidad es uno de los capítulos más importantes en cualquier API de desarrollo. En WPF, sin embargo, esta extensibilidad pasa a un segundo nivel ya que tenemos mecanismos para crear nuevos controles más sencillos de lo habitual. Aún así existen mecanismos para crear controles con los que encapsular funcionalidades que de manera reiterada, queremos usar en nuestros proyectos.

## A continuación

En el mundo tan globalizado en que vivimos las aplicaciones necesitan competir en un mercado cada vez más abierto y para ello que una aplicación funcione con independencia del país, el idioma o las configuraciones regionales de dramática importancia.

De esto se encarga la globalización y la internacionalización, en el siguiente capítulo veremos como llevar a cabo estas actividades en WPF.



# Localización e Inter- Operabilidad

En este capítulo trataremos dos temas que si bien no son parte del núcleo principal de servicios de aplicación en WPF, si que pueden tener una importancia capital en según que aplicaciones. La internacionalización de aplicaciones y la interoperabilidad con Windows Forms.

Cuando limitamos nuestra aplicación a un sólo idioma estamos limitando su mercado a una pequeña porción de los consumidores potenciales, al tiempo que competimos en inferioridad de condiciones con nuestros competidores.

Este proceso no sólo involucra el soporte de más de un idioma, sino en ocasiones también el diseño de la aplicación, los formatos y hasta los alfabetos se ven afectados.

De la traducción se ocupa la localización y del resto la globalización.

La interoperabilidad con Windows Forms es un aspecto ciertamente muy colateral, pero que puede permitirnos la adopción de WPF de una manera más rápida.

Esto es así debido a que un buen mecanismo de interoperabilidad evita la necesidad de una traducción completa de toda una aplicación.

## Localización y Globalización

Existen una serie de prácticas, que son comunes a todas las aplicaciones en .NET para globalizar y que por lo tanto no vamos a tratar aquí. Sin embargo existen algunas recomendaciones que en forma de buenas prácticas sí trataremos.

Trate de no crear en código los interfaces, de esta manera quedarán expuestos a las API de internacionalización.

Evite las posiciones absolutas y los tamaños fijos.

Utilice el panel Grid y su posicionamiento y dimensionamiento en vez del Canvas.

Aprovisione de espacio extra en los textos, porque el texto localizado suele ser de diferente tamaño.

Utilice el *TextWrapping*.

Establezca el atributo *xml:lang* allá donde pueda, ya que es el atributo que establece el lenguaje.

Cuando se creen aplicaciones de navegación establecer la propiedad *FlowDirection* en el texto, a fin de que esta no se herede del navegador y pueda funcionar mal en un entorno idiomático diferente al esperado.

## Localizando una aplicación

Para localizar una aplicación WPF hemos de tener en cuenta en primer lugar, que los interfaces se especifican usando XAML, lo cual implica que WPF toma una serie de características de XML en cuanto a la localización y globalización.

La primera de ellas es la referencia de caracteres. En ocasiones nosotros podemos necesitar insertar caracteres como una referencia numérica, bien en decimal o en hexadecimal, basada en el código de posición en el conjunto de caracteres actual.

En el caso de especificarlo con número decimal, este ha de ir precedido por “&#” y finalizar con “;”. En caso de que el número se especifique con hexadecimal la combinación de inicio sería “&#x”

```
XAML
    &#1000;
    &#x3E8;
```

XAML también con diversas codificaciones, en concreto *ASCII*, *UTF-16* y *UTF-8*, la especificación de cada una de ellas se realiza conforme al procedimiento XML, o sea;

```
XAML
    <?xml encoding="UTF-8"?>
```

Otra característica relacionada con XML es el atributo “xml:lang”. Este atributo especifica el lenguaje de un elemento. Este atributo puede especificar cualquiera de los valores de *CultureInfo*.

También se incluye soporte para todos los sistemas de escritura de Framework, incluido el soporte para OpenFonts. El soporte de renderización está basado en la tecnología Microsoft ClearType sub-pixel lo que aumenta la legibilidad significativamente.

Por último destacar que el soporte de flujo en el idioma se encuentra soportado por el atributo “FlowDirection” que admite los valores *LeftToRight* para los derivados del latín, este asiático y similares y *RightToLeft* para árabe, hebreo y similares.

Tras esta introducción de las características de globalización genéricas pasemos a completar un ejemplo concreto.

Si pretendemos localizar una aplicación lo primero que debemos tener en cuenta es cuál es el idioma neutral. Este idioma es el que la aplicación entenderá como suyo, como idioma que será usado tanto en los ambientes que coincidan con él como en aquellos ambientes que no coincidan con ninguno de los presentes en la aplicación. Para establecer el idioma neutral debemos ir a las propiedades de la aplicación, a la pestaña Aplicación y a la información del ensamblado, donde podremos establecerlo. Una vez hecho esto el proceso podrá comenzar.

Al igual que el resto de las aplicaciones .NET, las aplicaciones WPF son capaces de soportar ensamblados localizados. Estos ensamblados son básicamente de recursos que se compilan a partir de un ensamblado normal. Esto quiere decir que cuando tenemos un ensamblado y lo compilamos con un idioma distinto al neutral un ensamblado de recursos se genera en un subdirectorío con el nombre de la cultura seleccionada. Para conseguir esto hemos de incluir en el archivo de proyecto .csproj o .vbproj la siguiente línea dentro del grupo de propiedades correspondiente a la plataforma y configuración seleccionadas para la compilación:

```
XML
  <UI Cul ture>en-US</UI Cul ture>
```

Quedando el proyecto de la siguiente manera:

```
XML
  <PropertyGroup Condi ti on="
    '$(Confi gurati on)|$(Pl atform)' == ' Debug|x86' ">
    <Pl atformTarget>x86</Pl atformTarget>
    <DebugSymbol s>>true</DebugSymbol s>
    <DebugType>ful l</DebugType>
    <Opti mi ze>fal se</Opti mi ze>
    <OutputPath>bi n\Debug\</OutputPath>
    <Defi neConstants>DEBUG; TRACE</Defi neConstants>
    <ErrorReport>prompt</ErrorReport>
    <Warni ngLevel >4</Warni ngLevel >
    <UI Cul ture>en-US</UI Cul ture>
  </PropertyGroup>
```

Al ejecutar la compilación el siguiente directorio se creará:



Como se puede ver en la imagen un nuevo ensamblado con el mismo nombre que la aplicación pero con la extensión Resources.dll se crea.

Pero, ¿qué contiene este ensamblado?. Básicamente contiene los recursos de la propia aplicación etiquetados como para ser cargados cuando la aplicación se encuentre en un entorno coincidente con el de la cultura.



# Interoperabilidad

Bajo el nombre de “interoperabilidad” se esconde el camino para hacer trabajar una o más tecnologías en principio exclusivas, como podría ser el caso de WPF y Windows Forms.

No sólo Windows Forms sino DirectX o la API Win32, pueden ser objeto de interoperabilidad con WPF, si bien al ser este un libro escrito sobre la tecnología .NET poco sentido tendría abordar semejantes interoperabilidades.

Una vez centrado el objeto de nuestro interés diremos que abordar la interoperabilidad entre WPF y WinForms es tanto como abordar como integrar WPF en aplicaciones WinForms y viceversa.

El control WindowsFormsHost nos permite la interoperabilidad con Windows Forms desde WPF. Los siguientes escenarios de interoperación se soportan cuando una aplicación WPF ha de alojar un control Windows Forms:

- El control WPF ha de acoger uno o más controles usando XAML o código.
- El control Windows Forms container puede contener controles Windows Forms que a su vez acojan otros.
- Puede acoger formularios maestro/detalle, sea el formulario maestro bien WPF bien Windows Forms.
- Puede acoger uno o mas controles ActiveX o controles compuestos.
- Puede contener controles híbridos usando XAML o código.

En cuanto al sistema de "Layout" o distribución de los controles de la aplicación, existen las siguientes limitaciones:

- En ocasiones los controles Windows Forms sólo pueden ser redimensionados hasta un punto. Ej.: Combobox.
- Los controles Windows Forms no se pueden rotar o desplazar.
- En la mayoría de los casos los controles Windows Forms no soportan escalado.
- Los controles Windows Forms al tener HWND siempre se dibujan por encima de los controles WPF.
- Los controles Windows Forms soportan autoescalado basado en el tamaño de fuente mientras que en WPF no es así.



En algunas propiedades WPF tienen su equivalente en Windows Forms, estas son tratadas como propiedades del control.

En el caso de Windows Forms existe también un control llamado `ElementHost` que nos permite alojar contenido WPF en aplicaciones Windows Forms. Se soportan, por contra, los siguientes escenarios cuando una aplicación Windows Forms acoge un control WPF:

- Usar uno o más controles WPF usando código.
- Asociar una hoja de propiedades con uno o más controles WPF.
- Alojar una o más páginas WPF en un formulario.
- Arrancar una aplicación WPF.
- Alojar un formulario maestro detalle con el maestro Windows Forms o WPF.
- Alojar controles WPF personalizados.
- Alojar controles híbridos.

## Conclusión

Con WPF podemos crear fácilmente aplicaciones internacionales que nos permitan llegar a, cuantos más mercados mejor. Este soporte es plenamente compatible con el soporte general de .NET.

Además prevee mecanismos para que la integración con las tecnologías circundantes, léase Windows Forms, API Win32 o DirectX puedan utilizarse de la manera más efectiva posible.

