

programación en **Silverlight 4.0**

Marino Posadas



netalia
just for developers

Programación en Silverlight 4.0

Marino Posadas

ADVERTENCIA LEGAL

Todos los derechos de esta obra están reservados a Marino Posadas y **netalia**.

El editor prohíbe cualquier tipo de fijación, reproducción, transformación o distribución de esta obra, ya sea mediante venta, alquiler o cualquier otra forma de cesión de uso o comunicación pública de la misma, total o parcialmente, por cualquier sistema o en cualquier soporte, ya sea por fotocopia, medio mecánico o electrónico, incluido el tratamiento informático de la misma, en cualquier lugar del mundo.

La vulneración de cualesquiera de estos derechos podrá ser considerada como una actividad penal tipificada en los artículos 270 y siguientes del Código Penal.

La protección de esta obra se extiende al mundo entero, de acuerdo con las leyes y convenios internacionales.

© Marino Posadas, 2011

© **netalia**, S.L. 2011

Programación en Silverlight 4.0

Autor: Marino Posadas

Responsable editorial: Paco Marín

Diseño de cubierta: Silvia Gil y Javier Roldán

Maquetación: Silvia Gil

Editado por **netalia**

c/ Robledal, 135

28529 - Rivas Vaciamadrid (Madrid -España)

Tel. (34) 91 666 74 77 - Fax (34) 91 499 13 64 - <http://www.netalia.es>

Precio: 29,50 €

Déposito Legal: M-

ISBN: 978-84-934895-9-5

Impreso por Publidisa, S.A.

"El que enseña a hablar enseña también a pensar"

Anónimo

A la memoria de mis padres.

"Creo que parte de mi amor a la vida se lo debo a mi amor a los libros"

Adolfo Bioy Casares

A Milagros, que siempre me empuja para que siga escribiendo.

"El verdadero amigo es aquel que está a tu lado cuando preferiría estar en otra parte"

Lew Wein

A Paco Marín y Paco Morero, dos amigos.

"Las palabras son enanos; los ejemplos son gigantes"

Proverbio Suizo

"Que otros se jacten de las páginas que han escrito;
a mí me enorgullecen las que he leído"

Jorge Luis Borges

Agradecimientos

Al grupo de MVP de desarrollo español, con quienes tengo más contacto, y Cristina González Herrero a la cabeza. Muchos habéis apoyado mi trabajo, y quiero recordaros aquí. No cito nombres, así que no me olvidado de nadie y nadie se da por olvidado.

Y, por supuesto, a "los chicos/as del DPE" y MSDN en Microsoft Ibérica: Alfonso Rodríguez, David Carmona, Enrique Fernández, Antonio Gómez, David Salgado, César de la Torre, Aurelio Porras, Isabel Gómez... ¡seguid así!

A mis familiares, que sé que —leáis o no leáis mis trabajos— siempre me apoyáis para que continúe con ellos, y a otros profesionales que se han mostrado siempre favorables y con los que me unen intereses o ideas comunes: Luis Miguel Blanco, Xabier Egüés, Silvia Gil, Octavio Hernández, Andy González, y, en general, a los compañeros que me han apoyado, antiguos y actuales. Gracias a todos.

I. Introducción a Silverlight 4.0	13
A quien va dirigido este libro	14
Requisitos previos	14
Aplicaciones R.I.A.	15
Principales características de Silverlight 4:	16
Arquitectura de Silverlight	17
Apartados de la arquitectura	17
Tecnología DeepZoom	19
Media Stream Source (o streaming adaptativo)	19
Documentación	19
Silverlight y los lenguajes	19
Lenguajes dinámicos	20
Las herramientas de desarrollo para Silverlight	20
Características de la adaptación de .NET Framework para Silverlight	21
El proceso de instalación en el cliente y compatibilidad de navegadores	22
Modelos de ejecución e interoperabilidad en Silverlight 4	23
El modelo de seguridad de las aplicaciones Silverlight	24
Los niveles de seguridad en la ejecución del código en modo aislado	25
Procesos iniciados por el usuario	26
Modelo de aplicaciones: servicios extendidos	27
Estructura de una aplicación Silverlight 4	28
La primera aplicación	28
Características de una primera aplicación alojada en una página Web predeterminada	29
El rol de la Clase App	31
La clase MainPage	32
El papel de Expression Blend 4.0 en el desarrollo y diseño	35
Resumen	35

2. El lenguaje XAML	37
Lenguaje declarativo, de marcas y basado en XML	37
El lenguaje XAML	39
Vectorial vs. Mapa de Bits	39
Elementos estáticos: la interfaz de usuario básica	40
Atributos de propiedad y elementos de propiedad	41
Propiedades vinculadas (Attached Properties)	42
Elementos Contenedores	43
Espacios de nombres en XAML	44
Intellisense	44
Atributos de Extensión de Marcado	45
Otras extensiones de marcado	46
Binding	46
Una división fundamental: Elementos de dibujo vs. Controles	48
Controles	48
Shapes	49
El inevitable "Hola Mundo" como un dibujo que responde a eventos	49
Programando un elemento Path en C#	51
Las jerarquías descendientes de Shape	53
Los objetos Line, PolyLine y Polygon	57
Geometries: elementos de definición visual	58
Dibujos vectoriales y el objeto PathGeometry	61
Curvas Bézier	64
Una nota sobre la conversión automática de formatos en ficheros XPS	66
Creación de shapes definidas por el usuario	67
Dependency Properties y Eventos	70
Propiedades de Dependencia y propiedades del CLR	71
Conclusión	72
 3. Contenedores y disposición visual	 73
Controles de terceros	74
La arquitectura visual de una aplicación	75
Disposición visual de los elementos (Layout)	75

Los contenedores: una clase especial de controles	75
Elementos contenedores	76
Configuración de contenedores Grid	76
Diferencias en la presentación con objetos Canvas y StackPanel	78
Márgenes y alineación	81
StackPanel	82
VirtualizingPanel	82
Los otros contenedores	83
DockPanel	83
WrapPanel	85
ViewBox	86
Conclusión	87

4. Expression Blend 4 para desarrolladores 89

El papel de Blend 4.0 y la suite Microsoft Expression 4.0 en el desarrollo con Silverlight	89
Primera aplicación en Blend 4.0	90
El IDE de Expression Blend 4.0	93
Predeterminado vs. No predeterminado en el IDE de Blend 4.0.	95
La sub-ventana de recursos disponibles	95
Creación automática de elementos a partir de recursos gráficos	96
Categorías de recursos	97
División de los controles según Blend	98
Ejemplo de inicio con un gráfico, una imagen y un vídeo	98
Trabajo con objetos de dibujo	99
Cambio del punto de entrada de la aplicación	102
Trabajando con la Ventana de Diseño	103
La superficie de diseño	104
Brushes	105
VideoBrush	109
WebBrowserBrush	110
Brushes clásicos: SolidColorBrush, LinearGradientBrush y RadialGradientBrush	110
Otras sub-ventanas de propiedades	112

Búsqueda de propiedades	113
Los nuevos recursos en Blend 4	114
Conclusión	115

5. Controles más usuales en Silverlight 4 117

Controles en Silverlight 4	117
Controles que muestran/admiten texto como función primaria	119
ListBox y los elementos colectivos	121
Programación básica de eventos	123
Asignación de propiedades tipo DP (DependencyProperty)	125
Algunos controles especiales	126
ContentControl	126
Etiquetas Flotantes (ToolTip y ToolTipService)	128
Los "Presenters"	130
Más sobre el sistema de eventos en Silverlight	131
Estilos y Plantillas	135
Uso de estilos para cambiar la apariencia de un conjunto de controles	135
Utilización de TemplateBinding en plantillas	140
ContentPresenter	141
Edición de estilos y plantillas desde Expression Blend	142
Resumen	143

6. Más allá de los controles 145

La nueva oferta para interfaces de usuario	145
Transformaciones	145
Transformaciones	146
Propiedades RotateTransform y ScaleTransform	146
Creación de transformaciones mediante código	148
Matrices de Transformación y TransformGroups	148
3D en Silverlight: proyecciones	149
Efectos	150
Otros elementos vinculados con Blend	151

Triggers	154
El sistema de animaciones de Silverlight (Silverlight 4 Animation System)	156
Animaciones	156
Animaciones en Silverlight	157
Lanzando animaciones mediante código	157
Creación de animaciones mediante código	159
Interpolación no-lineal	163
Visual State Manager	164
El modelo Parts-And-States	165
Utilización de Visual State Manager para la personalización de una interfaz de usuario	167
Creación de estados y transiciones propios con VSM	168
Construcción de un control personalizado	170
Pasos típicos en la creación de un control de usuario	175
El control TextoDNI	176
Conclusión	181

7. Las opciones de interacción 182

Navegación entre páginas de una aplicación	183
Interacción con el entorno	183
Utilización de ventanas modales	187
Acceso a recursos del sistema	189
Cajas de diálogo OpenFileDialog y SaveFileDialog	189
Acceso al Portapapeles	190
Menús Contextuales	191
Manejo de la impresora	193
Interacción dúplex (en los dos sentidos) con el HTML DOM	196
El puente HTML - Silverlight	200
Interacciones con otras aplicaciones Silverlight	201
Ejecución directa fuera del navegador (Out-of-Browser)	203
Instalación con permisos elevados	207
Actualizaciones	208
Instalación dirigida por el usuario	209

Ventanas de notificación	209
Mostrar contenido HTML en aplicaciones OOB	211
Acceso a otros recursos	212
Soporte de arrastrar-y-soltar (Drag&Drop)	216
Conclusión	218

8. WP7: una nueva plataforma 220

Introducción a Windows Phone 7	221
Otros elementos de hardware	222
Arquitectura de Windows Phone 7 para el desarrollador	222
Otras plataformas de ejecución	223
Programación para WP7	224
Características propias del teléfono: Orientación, Manipulación Gestual y Navegación	228
Manipulación (Touch)	230
Navegación	232
Paso de datos entre páginas	234
Otras plantillas	235
Aplicaciones Panorama y Pivot	236
El control Pivot	237
Recomendaciones generales para el desarrollo con Windows Phone 7	238
Conclusión	242

Bibliografía e información "on-line" 243

Libros	243
Información "On-Line" (ingles)	243
Información "On-Line" (Castellano)	243

Introducción a Silverlight 4.0

Silverlight ha recorrido ya un intenso camino (4 versiones) en el poco tiempo que lleva en el mercado. Hace apenas dos años que nos dirigíamos al lector desde esta misma plataforma (los cuadernos técnicos de **dotNetManía**), y apenas acababa de salir la versión 2.0, que para muchos supuso la primera versión realmente seria y operativa, simplemente por que incorporaba una gran novedad respecto a otras tecnologías equivalentes de sus competidores: no solo permitía trabajar con el lenguaje de definición de interfaces de usuario (XAML) —ya propuesto en la primera versión— sino que incorporaba la posibilidad de completar el código funcional con 2 de los lenguajes fundamentales de .NET Framework: C# y VB.NET.

Ahora, estamos, como suele decirse, en un período de madurez. Esta madurez (que comenzó con la versión anterior), se manifiesta claramente en ésta y no en vano el énfasis principal del máquetin del producto se centra en la posibilidad de utilizarlo como plataforma de aplicaciones Web de Gestión (o como ahora se les llama, LOB, en sus siglas en inglés: *Line Of Business Applications* o aplicaciones de línea de negocio).

La idea es que no se piense desde fuera en Silverlight como una mera herramienta de construcción de sitios al estilo de su único competidor serio: Adobe Flash. Silverlight no olvida sus orígenes y las bondades que le han hecho alcanzar un nivel de adopción record en tan poco tiempo: animaciones, gestión de recursos multimedia, tecnología *DeepZoom*, etc.

Por lo demás, y hasta ver lo que sucede con otras alternativas (Adobe AIR, Java FX, Google Apps, etc.), Silverlight permite utilizar el lenguaje al que el lector está acostumbrado para programar cosas nuevas. Nuevas interfaces, basadas en los avances del lenguaje XAML, que ahora se encuentra totalmente soportado por Visual Studio 2010

¹ Paralelamente a esta obra, el lector podrá acceder a otra —centrada en las aplicaciones LOB— publicada por esta misma editorial.

(incluyendo su diseñador visual), de la misma forma que la última versión de Expression Blend (4.0) soporta la edición de código en C# o VB.NET, algo que no estaba disponible cuando escribíamos sobre la versión 2.0¹

En lo tocante a la documentación "on-line", encontrará una lista de enlaces comprobados de los especialistas en este tema al final de esta obra (Apéndice 1).

A quien va dirigido este libro

La obra va dirigida a jefes de proyecto, desarrolladores y —en general— a todos los profesionales de la programación con conocimiento de la tecnología .NET que deseen adentrarse en el desarrollo de nuevas interfaces de usuario, tanto en la Web, como en aplicaciones de escritorio, gracias a la característica de ejecución fuera del navegador (*Out-of-Browser*), que se presentó en la versión 3.0 y ha madurado considerablemente en esta versión. Comentamos ese aspecto en el Capítulo 7.

Requisitos previos

El conocimiento de la programación con .NET Framework es prácticamente imprescindible, así como de su herramienta de desarrollo, Visual Studio. Utilizaremos la versión 2010, pero a propósitos de seguimiento, puede hacerse perfectamente con la versión gratuita del producto *Visual Studio 2010 Express Edition*) e, igualmente, conviene tener conocimientos generales de la programación para Internet, y sus tecnologías asociadas (incluyendo lenguajes de marcas, y, especialmente, XML).

En cuanto a los lenguajes de programación, es importante la experiencia con cualquiera de los dos lenguajes principales soportados por Visual Studio (C# y Visual Basic .NET), aunque en esta obra utilizaremos C# en los ejemplos. En un balance general, será mayor la cantidad de código XAML que C#, y el de los ejemplos en éste lenguaje es fácilmente comprensible por cualquier programador de Visual Basic .NET.

Sin ser un requisito, hay otras tecnologías relacionadas que pueden ayudar en ciertos pasajes del libro, como LINQ o *Windows Presentation Foundation* (en la parte de XAML).

En cuando a Silverlight, el SDK del producto se descarga (así como ejemplos y abundante documentación y vídeos en inglés), de la página oficial (<http://Silverlight.net>).

Además, para la programación del apartado de *Windows Phone 7*, se precisará el SDK correspondiente, que instala las plantillas necesarias en Visual Studio 2010, así

como una versión gratuita de *XNA Game Studio 4.0*, la herramienta para construcción de juegos de Microsoft (Ver Capítulo 8).

Aplicaciones RIA

Silverlight 4.0 es, pues, una plataforma para la construcción de aplicaciones **RIA (Rich Internet Applications)**, que tiene la característica de conjugar lo mejor del mundo Web con las ventajas de las aplicaciones de escritorio: experiencia de usuario de alta calidad, y distribución remota. Además, a partir de la versión 3.0 de la plataforma, Silverlight permite la ejecución de aplicaciones fuera del navegador. Esta característica de ejecución tiene importantes implicaciones en la distribución, implantación e instalación en el cliente final, como es la posibilidad de asignar permisos especiales (siempre bajo el consentimiento del usuario de la aplicación), que termina con las limitaciones de la ejecución en *sandbox*, y permite acceder a ciertas áreas comunes del sistema local de ficheros y a otros recursos del sistema operativo, como la Impresora, el Portapapeles, etc.

nota

Una aplicación de este tipo se ejecutará en una ventana de Windows, que puede ser personalizada si se desea.

Versión	Tipo MIME	Número de versión	Dirección URL del instalador
Silverlight 1.0	application/x-silverlight	1.0	http://go.microsoft.com/fwlink/?LinkId=110408
Silverlight 2	application/x-silverlight-2	2.0.31005	http://go.microsoft.com/fwlink/?LinkId=124807
Silverlight 3	application/x-silverlight-2	3.0.40624	http://go.microsoft.com/fwlink/?LinkId=149156&v=3.0.40624.0
Silverlight 4	application/x-silverlight-2	4.0.50826	http://go.microsoft.com/fwlink/?LinkId=188043

Tabla 1: Versiones y Tipos MIME

Respecto a las versiones de la plataforma, en la tabla 1 se muestra el tipo MIME y el número de versión vinculado a cada versión principal del complemento (cada versión es compatible con las versiones anteriores). Por ejemplo, si se especifica que la aplicación requiere Silverlight 1.0, la aplicación se ejecutará en Silverlight 3, pero solo podrá usar la API de JavaScript (y no las de C#, VB.NET o cualquier otro lenguaje válido para versiones posteriores).

nota

Hay que tener en cuenta que el tipo MIME "application/x-silverlight-2" se usa para Silverlight 2 y todas las versiones posteriores

Principales características de Silverlight 4

La propia Microsoft, define el producto como "una tecnología que permite crear aplicaciones de vanguardia", destacando entre sus características más sobresalientes:

- **XAML.** El subconjunto de WPF que define los elementos en el navegador para crear la interfaz de usuario. Esto supone crear gráficos, animaciones y elementos multimedia, y otras características de cliente enriquecido, permitiendo ir más allá de lo que HTML permite.
- **Extensiones de JavaScript.** Las extensiones al lenguaje de *scripting* de explorador que permiten controlar la interfaz de usuario, incluida la posibilidad de trabajar con elementos XAML. La comunicación entre los dos mundos (HTML/DOM y XAML) puede, además, realizarse en los dos sentidos.
- **Compatibilidad con otros exploradores y plataformas.** Su ejecución es idéntica en todos los exploradores conocidos (y en sus plataformas). Esto incluye varias versiones de Windows y Mac, y gracias a la plataforma *MoonLight*, *Linux* y *FreeBSD*.
- **Integración con las aplicaciones existentes.** Se integra perfectamente con el código JavaScript y ASP.NET AJAX existente, de modo que complementa la funcionalidad ya creada. De esta forma, Silverlight puede empotrarse como una isla dentro de una página o cubrir toda la interfaz de usuario.
- **Modelo de programación similar y compatible con .NET Framework.** Basado en la Capa de Adaptación de Plataforma (PAL), permite su ejecución en para-

lelo junto a otras versiones de .NET. También se pueden utilizar lenguajes dinámicos, como *IronPython* o *IronRuby* y enlazar con código de un lenguaje funcional como F#.

- Amplio conjunto de herramientas para desarrollo. Tanto Visual Studio .NET (2008/2010) como *Expression Blend*, permiten desarrollar para esta plataforma, si bien el soporte de la versión 4 de Silverlight se alinea con el mismo número de versión en *Blend*.
- Compatibilidad de red. Silverlight incluye compatibilidad con HTTP sobre TCP. Se puede conectar a los servicios WCF, SOAP, REST o ASP.NET AJAX y manejar formatos como Texto plano, XML, JSON o RSS.
- Soporte LINQ. *Language Integrated Query*, el lenguaje integrado de consultas que permite analizar y manejar orígenes de datos diversos con una sintaxis sencilla, intuitiva y muy potente.

Arquitectura de Silverlight

Aunque la evolución de la arquitectura ha ido añadiendo elementos y bloques funcionales desde la primera versión, las divisiones fundamentales podemos decir que son las mismas. Por un lado, tenemos la estructura de presentación de la interfaz de usuario, basada en el lenguaje XAML. Por otro, las librerías soportadas y las extensiones de los lenguajes, más todos los añadidos que se han ido incorporando con la evolución de la plataforma. El esquema gráfico de la arquitectura de Silverlight 4 que adjuntamos a continuación (descrito en la documentación oficial) muestra los distintos módulos y esa separación (más conceptual que real) en bloques operativos y/o funcionales:

Hay que notar que, respecto a las bibliotecas de MS-AJAX y JavaScript, de lo que se dispone es de la posibilidad de comunicarse con ellas; no se han creado bibliotecas especiales para este propósito.

Apartados de la arquitectura

Podemos distinguir en esta arquitectura varias divisiones:

- .NET para Silverlight, que no es más que el conjunto de clases que integran el CLR de del motor de ejecución, con sus principales apartados: Datos, WPF (XAML, en realidad), WCF, el soporte de lenguajes dinámicos y las librerías de

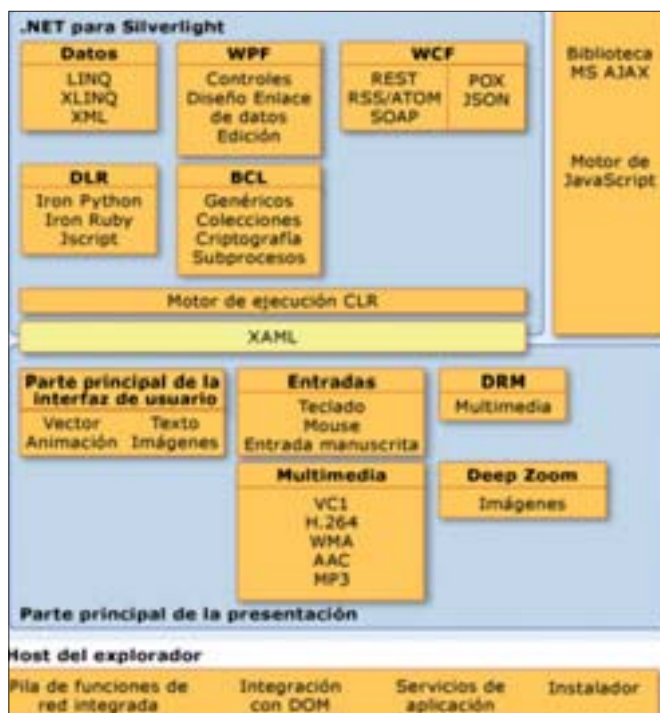


figura 1

Arquitectura de Silverlight 4.0

clase base (BCL), que están especialmente adaptadas para el componente Silverlight.

- Ese apartado conecta gracias al lenguaje XAML con la parte principal de la presentación. Esto incluye todos los elementos de entrada/salida: animaciones, tratamiento de texto, tratamiento de imágenes, sistemas diversos de entradas de teclado (incluyendo *Ink*, o entrada manuscrita y manipulaciones gestuales en Windows Phone 7), los elementos multimedia en sus diversos formatos, la gestión de derechos digitales (DRM) y el motor *DeepZoom* especializado en tratamiento progresivo y secuencial de imágenes.
- Por otro lado tenemos los vínculos con otros elementos externos, como la biblioteca de AJAX, o el motor de JavaScript
- Y, finalmente, los servicios y funciones que son propias del navegador que alojará el complemento, y a los que se tiene acceso desde éste, como la pila de funciones de Red, la integración con el *Modelo de Objetos de Documento* (HTML-

DOM, por distinguirlo del modelo de objetos que crea el complemento, llamado XML-DOM), los servicios de aplicación y los que dan soporte al proceso de instalación OOB (*Out-Of-Browser*) que citábamos al principio.

Tecnología DeepZoom

Otra novedad es la mejora del rendimiento de la tecnología *DeepZoom*, que permite a los usuarios profundizar en el detalle de fotografías o *collages* de fotografías, permitiendo una transición suave entre ellas, para obtener la sensación de profundidad. Las imágenes se pueden escalar desde resoluciones de 2 ó 3 megapíxeles hasta un gigapíxel, pero el usuario no tiene que esperar a la descarga completa, gracias a la tecnología subyacente que permite la descarga concreta de las zonas necesarias. Internamente, utiliza el formato de fichero XML.

Media Stream Source (o streaming adaptativo)

Se denomina así a una potente API que permite la descarga dinámica adaptable de contenidos multimedia. Esto permite a la aplicación que reproduce el contenido multimedia seleccionar la velocidad de descarga basándose en el ancho de banda del cliente y su potencia de CPU. Se implementa mediante una instalación en el servidor donde se aloja el contenido multimedia.

Documentación

En este apartado, la cantidad de documentación es —en la actualidad— enorme, sin hacemos una sencilla búsqueda en Internet. El sitio oficial y la documentación *on line* aparecen listadas en el apéndice 1 de esta obra, junto con otros recursos que permitirán al lector continuar la formación e investigar por su cuenta.

Silverlight y los lenguajes

En lo tocante a los lenguajes, ya hemos apuntado que Silverlight utiliza como lenguaje de definición de elementos visuales el mismo que usa WPF: **XAML** (también traducido como *Lenguaje de Marcado para Presentación*). El usuario que solicita una página que

contiene elementos enriquecidos con Silverlight (o si la página entera está hecha en Silverlight) recibe un único fichero que está compuesto de una DLL donde radica nuestra aplicación compilada (más todos los recursos que pudiera necesitar para su puesta en marcha), y un fichero **.manifest** que define otros aspectos externos de su ejecución.

Este fichero (de extensión **.xap**) no es más que un fichero comprimido **.ZIP** con su extensión cambiada (puede probarse esto renombrando el fichero distribuible y descomprimiéndolo). Cuando el navegador solicita una página con este tipo de contenido, el *runtime* se encarga de la descompresión dinámica y aislada, de la creación del modelo de objetos XAML y de poner a disposición del *runtime* los elementos precisos para la ejecución.

En la fase de desarrollo, el IDE de Visual Studio se encarga de hacer accesible (programable) ese modelo de objetos de forma que pueda ser codificado de manera prácticamente idéntica a la tradicional, usando un lenguaje de su elección (JavaScript, por defecto, en la versión 1.0, ó VB.NET/C# en la versión 2.0 y posteriores).

Lenguajes dinámicos

Otra novedad interesante aparecida en la versión 2.0 es el soporte de lenguajes dinámicos: *JavaScript*, *IronPython 2* y *IronRuby*. Para ello, se incorporó el *Dynamic Language Runtime* (DLR), que permite la compilación y ejecución dinámica de lenguajes de *script*. Si la aplicación hace uso de estos lenguajes, su compilador se empaqueta junto con el distribuible de la aplicación (el fichero **.xap**).

nota

Si es de su interés, el lector podrá encontrar abundantes ejemplos de utilización de estos dos lenguajes en Silverlight en el sitio oficial antes indicado y la documentación de referencia al final de esta obra.

Las herramientas de desarrollo para Silverlight

En el momento de escribir esto, estamos utilizando Visual Studio 2010, Expression Blend 4.0 y .NET Framework 4.0, además del propio SDK de Silverlight. Todo el proceso de instalación (si no se da ninguna incompatibilidad en las condiciones de instalación), es automático.

Con la instalación del SDK (*Silverlight 4 Tools para Visual Studio 2010*) dispondremos de todo lo necesario: nos aparecerán varias plantillas de proyecto Silverlight disponibles como un tipo más de proyecto en los dos lenguajes predeterminados y el Silverlight Toolkit², que se renueva periódicamente y se expone bajo la supervisión del equipo de desarrollo de la plataforma, añadirá un buen paquete de recursos consistente en un conjunto de controles, temas visuales y herramientas para facilitar la programación.

nota

Estrictamente hablando la programación con Silverlight puede realizarse de forma totalmente gratuita, ya que podemos utilizar *Visual Web Developer Express* o *Visual Studio Express* para las plataformas Windows, y otros equivalentes como *Eclipse* para los otros sistemas que hemos citado.

Una vez instalado, el entorno de desarrollo para Silverlight con Visual Studio es el habitual: tecnología *Intellisense*, tanto en la edición del código XAML, como en JavaScript, junto a las típicas capacidades de depuración, extendidas para soportar todos los lenguajes disponibles, soporte síncrono de diseño según se van editando elementos de la interfaz de usuario (incluyendo un diseñador visual al estilo de los de ASP.NET o *Windows Forms*), ayuda local y en línea, y los *gadgets* que son el día a día de Visual Studio.

Características de la adaptación de .NET Framework para Silverlight

El alma de la ejecución de las aplicaciones Silverlight es su versión del CLR, llamada *CoreCLR*. Aunque es un subconjunto de la plataforma .NET Framework completa, suministra las bases para un desarrollo potente y orientado a objetos para tipos de aplicaciones (como las de Internet) que tradicionalmente no contaban con estos recursos.

En la adaptación del *CoreCLR* a la plataforma Silverlight, se ha hecho hincapié en los siguientes aspectos:

- Datos. Se soportan características de LINQ (*Language-Integrated Query*) y de *LINQ-to-XML*, de forma que pueden integrarse datos procedentes de diferen-

² <http://silverlight.codeplex.com/>

tes orígenes. También admite el uso de las clases de seriación y XML para la gestión de datos.

- Las BCL. Se llaman así las librerías de .NET Framework que suministran los recursos esenciales (manejo de cadenas y expresiones regulares, procedimientos de entrada y salida, reflexión, colecciones y globalización).
- WCF (*Windows Communication Foundation*). Dispone de características que facilitan el acceso a los servicios y datos remotos. Esto incluye un objeto navegador, un objeto de solicitud y respuesta HTTP (*request/response*), compatibilidad con peticiones HTTP entre dominios, acceso a fuentes de distribución RSS/Atom, y soporte de los formatos de datos JSON, POX y SOAP.
- CLR (*Common Language Runtime*). Se encarga de la administración de la memoria, la recolección de basura (*garbage collector*), y de la seguridad de tipos y el control de excepciones.
- Controles tipo WPF (*Windows Presentation Foundation*). Suministran un conjunto de controles equivalentes a los ya conocidos de Windows Forms, tales como **Button**, **Calendar**, **CheckBox**, **DataGrid**, **DatePicker**, **HyperlinkButton**, **ListBox**, **RadioButton** y **ScrollViewer**. Se complementan con el Silverlight Toolkit.
- DLR (*Dynamic Language Runtime*): como hemos anticipado antes, se trata de una extensión que permite la compilación y ejecución dinámicas de lenguajes de *scripting*, como JavaScript o IronPython como una alternativa a los lenguajes orientados a objetos. Incluye un elemento para suministrar compatibilidad con otros lenguajes para su uso con Silverlight.

El proceso de instalación en el cliente y compatibilidad de navegadores

El proceso de instalación del componente es muy sencillo, y basta con acceder a una página que contenga un complemento Silverlight para que (salvo programación en contrario), se le pregunte al usuario si desea realizar la descarga e instalación automáticas, exactamente igual que en versiones anteriores.

Como ya hemos mencionado, la única diferencia reside en que, al existir la posibilidad de que las aplicaciones así configuradas se ejecuten fuera del navegador, la instalación puede crear un enlace en el escritorio o en el Menú de inicio y desde ese momento, lanzarse sin necesidad de una conexión a Internet.

Respecto a los navegadores compatibles, la gran mayoría de los más populares soporta Silverlight: I. Explorer 6+, FireFox 3+, Google Chrome 4, Safari 3+, Opera 9+, etc.

De la misma forma, la gran mayoría de las plataformas Windows soportan la ejecución del complemento, y lo mismo sucede con Mac y Linux/FreeBSD a través del proyecto **MoonLight** (una versión de Silverlight adaptada por **Miguel de Icaza**, director del proyecto **Mono** para estos sistemas operativos).

Modelos de ejecución e interoperabilidad en Silverlight 4

La documentación oficial recuerda que, desde esta versión, los modelos de ejecución del complemento han variado, y pueden considerarse 5 grandes grupos, aparte del modelo clásico de ejecución dentro de una página Web:

- Utilizar Silverlight fuera del explorador.
 - En este caso, se permite la ejecución fuera del explorador en equipos Windows y Macintosh para las aplicaciones Silverlight alojadas en web. Requiere que los usuarios instalen la aplicación desde una página web que lo aloje.
 - Estudiaremos esta opción con más detalle en los apartados dedicados a la distribución e implantación de aplicaciones.
- Utilizar Silverlight en un control de explorador (**WebBrowser** o equivalente).
 - Desde una aplicación creada con otra tecnología, se puede recurrir a mostrar una página que contenga un complemento Silverlight, siempre y cuando dispongamos de un control **Browser**, que sea capaz de abrir e interpretar la página que aloja el complemento.
 - La comunicación entre el complemento Silverlight y el programa puede establecerse mediante el puente HTML: un mecanismo de interacción entre el modelo de objetos de documento (HTML/DOM) y Silverlight, que funciona en las dos direcciones.
 - Este tipo de alojamiento dispone de varias posibilidades dependiendo del contexto del programa a utilizar:
 - Controles **Browser** de Windows Forms, WPF o Win32
 - Componentes MSHTML
 - WebKit* de Macintosh OS X.
- Utilizar una interfaz de usuario de Silverlight en otro complemento (*add-in*) o aplicación.

–Se trata de una opción no demasiado conocida, basada en utilizar una interfaz no administrada llamada **ISilverlightViewer**, mediante la que se puede recuperar una representación gráfica (mapa de bits) de una interfaz de usuario Silverlight. Esto puede resultar interesante en el caso de que necesitésemos mostrar toda o parte de la interfaz de Silverlight dentro de otro complemento o una aplicación Win32 que no disponga o utilice un control **WebBrowser**.

- Utilizar Silverlight en una aplicación basada en Windows a través de COM.
 - Dada la cantidad de aplicaciones COM todavía en funcionamiento y soporte, Silverlight suministra varias interfaces COM de forma que una aplicación de este tipo alojar un complemento Silverlight. Se trata de una opción más compleja de programar, pero de gran flexibilidad en ciertos casos puntuales.
- Utilizar Silverlight en otra aplicación a través de la interfaz del explorador .
 - Finalmente, podemos utilizar el modelo de programación de *add-ins* (complementos) de un navegador. La aplicación que hace de *host* es la encargada en este caso de incrustar el complemento Silverlight igual que si se tratara de un explorador. Es particularmente útil para las aplicaciones basadas en Macintosh que no disponen de acceso a interfaces COM.

La página oficial de MSDN [http://msdn.microsoft.com/es-es/library/dd550717\(v=VS.95\).aspx](http://msdn.microsoft.com/es-es/library/dd550717(v=VS.95).aspx) recoge estas posibilidades y contiene enlaces a otras páginas de ampliación en el caso de que necesitemos estas opciones alternativas.

Y en el caso de que nuestra arquitectura sea tal que contenga varios complementos Silverlight dentro de una página Web, es posible sincronizar el comportamiento de todos o algunos de ellos a través de un sistema de mensajería entre complementos, que apareció en la versión 3.0 de la plataforma y que veremos con ejemplos en el capítulo 7.

El modelo de seguridad de las aplicaciones Silverlight

La versión 2.0 introdujo un modelo de seguridad de ejecución basado en el tipo de ejecución en *sandbox* (áreas protegidas de ejecución) que significaba que no se podían efectuar llamadas directas de ninguna clase a una API de tal forma que pudieran suponer un peligro potencial para la seguridad del usuario. A diferencia de lo que sucedía con el modelo CAS (*Code Access Security*), éste nuevo modelo se basaba en el concepto de *Transparencia de Seguridad* (*Security Transparency*), que se introdujo en .NET 2.0.

Sin embargo, al aparecer los distintos modelos de ejecución que hemos visto en el punto anterior, el modelo de seguridad se ha enriqueció de forma que pueda dar cabida a esas opciones, sin que esto suponga un riesgo para la seguridad.

De forma que, si bien el modelo de ejecución dentro del navegador sigue manteniendo las características de ejecución aislada propias de las *sandbox*, en los modelos de ejecución fuera del navegador, se puede configurar la ejecución para que se produzca en un entorno de confianza, eliminando muchas de las restricciones de aquellas. Esto es algo que deberá de configurarse mediante programación y se dispone de varias opciones para ello.

Los niveles de seguridad en la ejecución del código en modo aislado

En mi anterior libro comentaba a este respecto que existen 3 niveles de seguridad en la ejecución: *Transparente*, *Crítico para la seguridad* y *Crítico*. Este modelo establece cómo se gestionan los permisos en la pila de llamadas y tiene que ver directamente con los dos tipos de código desde el punto de vista de la seguridad: el código de aplicación y el código de la plataforma de ejecución.

- El código transparente no puede modificar tales permisos.
- El código crítico para la seguridad es una capa de código accesible por el código transparente que recoge las peticiones "comprometidas" del código transparente, de forma que no permite la ejecución de llamadas potencialmente inseguras.
- El código crítico es el que puede realizar operaciones fuera del espacio aislado de seguridad, como por ejemplo, acceder al sistema de archivos.

Cuando el código transparente necesita de los servicios de una llamada que pertenezca a este código crítico, la efectúa a través de la segunda capa, que es la que decide si tal comunicación es posible o no, dependiendo de otros factores. La documentación oficial del MSDN incluye el gráfico ilustrativo de esta arquitectura que podemos ver en la figura 2.

Por ejemplo, si un fragmento de código transparente intenta llamar a una función calificada como código crítico se inicia una excepción del tipo **MethodAccessException**.

De esta forma, se nos recuerda que "el código crítico seguro ayuda a garantizar que el código transparente puede realizar operaciones críticas de manera segura", lo que significa que se efectúan ciertas comprobaciones antes de pasar el control a una API calificada como *Crítica*.

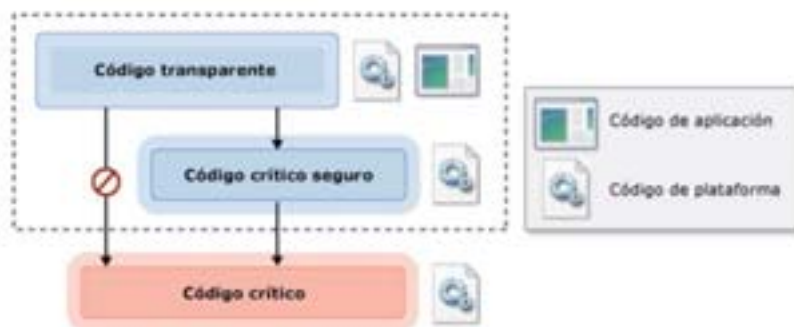


figura 2 Arquitectura de seguridad en la ejecución de las aplicaciones Silverlight 4

nota

Un caso típico es el del acceso al sistema de archivos. Si la aplicación necesita almacenar información en el equipo del usuario, lo que hará es utilizar la característica de almacenamiento aislado, y esto teniendo en cuenta las restricciones que son inherentes a este tipo de almacenamiento (la cuota de disco, los permisos de lectura/escritura, etc.).

Procesos iniciados por el usuario

Otra forma de garantizar que la ejecución tiene lugar en un contexto adecuado a la seguridad, es solicitando la intervención del usuario antes de continuar o iniciar un proceso que pueda implicar a la seguridad.

Esto es especialmente necesario a partir de Silverlight 4, ya que esta versión introduce aspectos novedosos de acceso a los recursos en la máquina local, como son la posibilidad de utilizar una *webcam* o un micrófono, o de mandar imprimir contenidos y acceder al portapapeles.

nota

Si el lector debe de utilizar en profundidad aspectos que implican la seguridad en sus aplicaciones, lo recomendado es que se descargue el documento "Security Guidance for Writing and Deploying Silverlight Applications", disponible en la dirección <http://www.microsoft.com/downloads/details.aspx?FamilyID=7cef15a8-8ae6-48eb-9621-ee35c2547773&displaylang=en>

Modelo de aplicaciones: servicios extendidos

Las librerías mencionadas y los recursos que albergan, permiten dotar a Silverlight 4 de características avanzadas en su modelo de aplicaciones, propias de las soluciones RIA actuales. En concreto, podemos distinguir 4 subsistemas importantes:

- El sistema de activación, que permite al complemento descargar el paquete .xap de la aplicación así como librerías externas que se necesiten al inicio.
- La clase **Application**, que analizaremos a continuación, que encapsula los servicios requeridos normalmente por las aplicaciones Silverlight, tales como la notificación de los eventos propios de su ejecución y la interfaz para el complemento Silverlight.
- El sistema de extensibilidad, que incorpora servicios tales como las capas de acceso a datos personalizadas.
- El subsistema de administración de recursos. Esto incluye referencias y mecanismos de reserva de los identificadores URI, la posibilidad de cargar recursos a petición, el uso compartido de estilos y plantillas y la compatibilidad con la globalización.

nota

Este último punto puede resultar crucial en la construcción de aplicaciones de gran tamaño desde el punto de vista del rendimiento. La solución propuesta y recomendada en estos casos es que se implemente un pequeño conjunto funcional inicial con una interfaz de usuario adecuada, y que se vayan descargando el resto de librerías solo en el caso de que se necesiten, mediante un sistema de peticiones.

- Servicios de navegación. Se implementan mediante un Marco de Navegación (*Navigation Frame*) basado en páginas y marcos (objetos **page** y **frame**), donde los marcos actúan como contenedor de controles de página y facilitan la navegación entre ellas, mostrando una sola en cada momento.
- Pantallas de presentación (*splash screens*). Silverlight presenta una animación simple predeterminada que va contando el porcentaje descargado de la aplicación. Pero podemos personalizarlas creando el tipo de animación que nos convenga. La llamada a estas pantallas, se realiza separadamente, mediante la API de JavaScript, y pueden construirse de forma sencilla en el lenguaje XAML, siendo llamadas al inicio por la etiqueta **<object>** de HTML que aloja el complemento Silverlight.

Estructura de una aplicación Silverlight 4

Cuando a continuación creemos una primera aplicación Silverlight utilizando el API administrada, veremos que se genera —como resultado de la compilación— un fichero .xap que no es sino un archivo comprimido en formato ZIP pero con el nombre cambiado.

Este fichero alberga todos los ensamblados necesarios para la ejecución de la aplicación así como el archivo **.manifest** propio de las aplicaciones .NET, y todos aquellos recursos que el usuario haya decidido empotrar para estar disponibles desde el mismo inicio. Puede tratarse de gráficos, iconos, elementos multimedia, o archivos de otras clases que sean requeridos por la aplicación, como ficheros de ayuda, etc.

nota

En el caso de que una aplicación sea alojada en un servidor Web que no reconozca la extensión de archivos **.xap** (el tipo *mime*), podemos distribuirlo renombrando el archivo con la extensión **.zip**, ya que lo primero que hará el sistema de alojamiento es descomprimirlo y, a partir de ahí, iniciar su ejecución.

La primera aplicación

Una vez recorridas las características principales de esta versión de la plataforma, es tiempo de comenzar por una aplicación básica que nos servirá para revisar la estructura de las aplicaciones predeterminadas y evaluar su arquitectura partiendo de los elementos que se crean en Visual Studio desde las plantillas iniciales. Abriremos, por tanto, el IDE de Visual Studio 2010, seleccionando el lenguaje correspondiente (C#, en nuestro caso), y seleccionaremos el apartado de aplicaciones Silverlight. El entorno de desarrollo nos ofrecerá una pantalla como la que vemos en la figura 3, con diferentes plantillas predeterminadas.

Observamos que, además de los dos tipos clásicos de aplicación (aplicación estándar y librería de clases), se ofrecen varias alternativas adicionales, con aspectos que aparecieron a partir de la versión 3.0.

Por ejemplo, para las aplicaciones de negocio (*Business Applications*), se crea una pequeña infraestructura de aplicación con ciertos elementos predeterminados que se suponen de utilidad en este tipo de soluciones.

Para las aplicaciones de navegación, se aportan 2 páginas extra (al margen de la predeterminada), con la capacidad de utilizar el sistema de navegación entre páginas

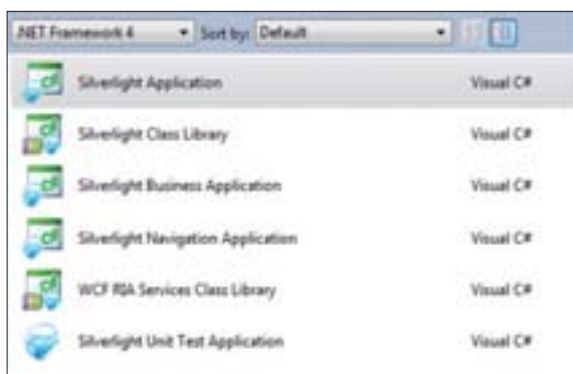


figura 3

Figura 2: Plantillas Silverlight disponibles en Visual Studio 2010 tras la instalación del SDK

introducido en la versión 3.0, y moverse entre ellas de forma que podamos construir con esta base una aplicación o un sitio web completo.

El tipo WCF RIA Services, suministra los fundamentos para la creación de una librería de acceso a servicios de datos, y finalmente, la opción "Unit Test Application", aporta la capacidad de realizar pruebas unitarias durante el proceso de desarrollo, aproximándonos de esa forma, al modelo de construcción de aplicaciones MVVM (*Model-View-ViewModel*).

Características de una primera aplicación alojada en una página Web predeterminada

Si optamos por la primera opción, el entorno nos preguntará si deseamos alojar el complemento en un nuevo sitio Web, así como el nombre de este y la opción de alojamiento que prefiramos:

- Proyecto de Aplicación Web ASP.NET
- Sitio Web de ASP.NET
- Sitio Web de ASP.NET mediante el modelo Vista/Controlador (MVC)

Seleccionaremos el primero para comenzar y veremos los elementos de la solución que el IDE genera para nosotros (figura 4).

Una vez seleccionada la primera opción, se creará una solución con dos proyectos: el de la aplicación Silverlight propiamente dicha, y el del sitio web que contiene dos páginas que alojarán al complemento (cualquier de ellas sirve en tiempo de ejecu-

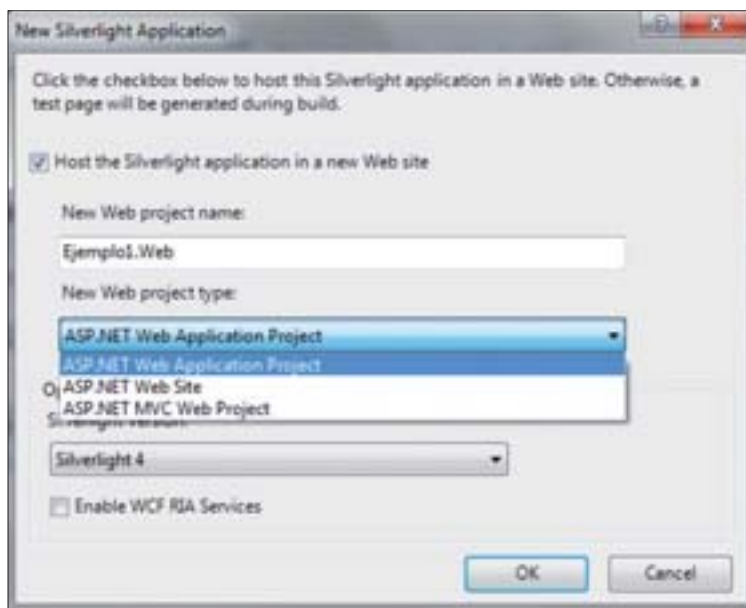


figura 4 Opciones de construcción de aplicaciones Silverlight con sitio web asociado

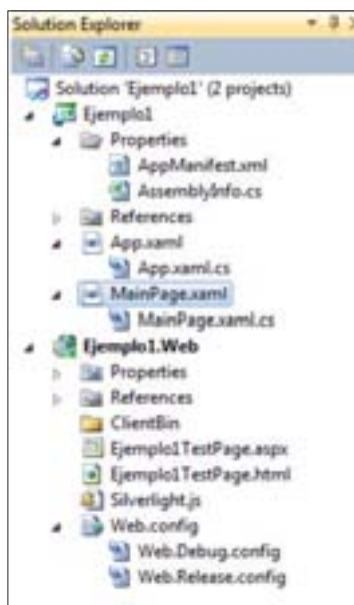


figura 5 Estructura de ficheros de la aplicación predeterminada anterior

ción). Basta comprobar la estructura de ficheros (figura 5) generada en el Explorador de soluciones.

En la primera, observamos que la aplicación Silverlight consta de varios ficheros de configuración, igual que cualquier otra aplicación .NET, más los ficheros propios de la aplicación: **App** y **MainPage** (ambos, con extensiones **.xaml** + **.cs**), que constituyen el cuerpo principal.

En el primer caso, se establecen las condiciones iniciales de ejecución, y se generan los procedimientos de evento fundamentales para la clase **Application**, encargada de cargar y poner en marcha la ejecución. Esta clase, proporciona un evento **Startup** que permite la programación de las condiciones iniciales, así como otra serie de servicios necesarios para la ejecución y el control de errores, por ejemplo.

El rol de la clase App

La clase **Application** (alojada en **App.xaml** y **App.cs**) pone a disposición del desarrollador un conjunto de servicios propios de las soluciones RIA, que van desde la extracción de recursos del paquete **.xap/.zip** descargado, o el establecimiento del punto de entrada de la aplicación (dentro del evento **Startup**), hasta el control de la duración de la aplicación, la gestión de excepciones no controladas en código, la integración con el *host* que aloja la aplicación y un marco completo de extensibilidad que permite al usuario personalizar más la ejecución.

Esto último, es posible rellenando la propiedad **LifeTimeObjects** con servicios de extensión de la aplicación. Este es el mecanismo recomendado en lugar de la utilización de clases que hereden del objeto **Application**. La ventaja de estos servicios es que permiten la integración completa con los eventos de la aplicación y además, facilitan el uso de varias extensiones simultáneamente.



figura 6

Gráfico estructural contenedor/contenido de una aplicación Silverlight 4

El MSDN de Microsoft, muestra un gráfico estructural de los contenidos (figura 6), junto a la relación de los componentes en el proceso de activación inicial.

Si bien el código para el punto de entrada se almacena en el fichero **App.cs**, el código **App.xaml** tiene otra función bien dispar. Sirve para almacenar recursos en lenguaje XAML de forma que estén accesibles para toda la aplicación (entendiendo por recursos los estilos, plantillas, temas y otros elementos visuales típicos de estos entornos). Por el mero hecho de situarlos ahí, adquieren un carácter global para cualquier otro fichero XAML que quiera utilizarlos posteriormente.

La clase MainPage

La clase **MainPage** representa al control inicial (o página principal) que suministra la interfaz de usuario que se mostrará al comienzo de la ejecución.

Se trata (al igual que **App**) de una clase parcial, definida en dos partes: el código C# que almacena un constructor de la clase, y que suele llamar al método **InitializeComponent**, y un archivo de extensión XAML que es el que contiene la definición visual de la interfaz.

Para anticipar básicamente la estructura de éste último, veamos el que se crea de forma predeterminada (listado 1) según el proceso anterior.

```
<UserControl x:Class="Ejemplo1.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">

    <Grid x:Name="LayoutRoot" Background="White">

        </Grid>
    </UserControl>
```

Listado 1

El código muestra un contenedor de interfaz, denominado **UserControl**, que contiene una rejilla o cuadrícula (**Grid**) que es la que albergará toda la interfaz de esta página. Como veremos después, el elemento **Grid** es uno de los contenedores más flexi-

bles que existen, y por eso se le utiliza de forma predeterminada (en el capítulo siguiente analizaremos con más detalle lo relacionado con el lenguaje XAML).

Conviene indicar que estas características de definición de interfaces del lenguaje XAML, junto a la posibilidad de crear totalmente desde cero cualquier interfaz de usuario (sin tenerse que basar en nada preexistente) fue una de las razones que llevaron a los arquitectos de Microsoft a decidirse por WPF en la propia construcción de Visual Studio 2010. Con esta versión del IDE, el Editor de XAML funciona de manera que, cuando realizamos cualquier modificación en el código XAML, ésta se refleja automáticamente en el diseñador visual, y —al revés—, cualquier modificación en el diseñador visual, modifica el código XAML subyacente. Curiosamente, esto es algo que comenzó a usarse en Visual Studio 2005 con la versión 2.0 de Silverlight, pero dada su inestabilidad, se abandonó para la siguiente versión. Finalmente, contamos con un diseñador con todas sus opciones.

Como el paradigma de desarrollo es muy similar al de modelos anteriores, si seleccionamos un elemento de la interfaz, e inspeccionamos la Ventana de propiedades, veremos que existen mecanismos de diseño visual asociados con ellas, para facilitarnos la selección de diferentes opciones, como sucede con el color de fondo de nuestro control (propiedad **Background** en la figura 7).

Y lo mismo sucede en muchas otras propiedades en las que los valores posibles pueden considerarse acotados (valores enumerados, recursos disponibles, etc.).



figura 7

Propiedad Background configurable en la Ventana de Propiedades

Desde el punto de vista de la aplicación Web, destacaremos que dispone de dos páginas de inicio alternativas. Si bien ambas funcionan de la misma forma, una de ellas es una página HTML tradicional y la otra una página ASP.NET. En cualquiera de los dos casos, la forma de alojar el control es mediante una etiqueta **<object>** cuyos parámetros se ajustan para apuntar al fichero XAP correspondiente, indicar la versión mínima que se requiere para la ejecución, configurar —eventualmente— una pantalla de inicio personalizada, el ancho y alto del complemento, etc.

El código generado para esa etiqueta en nuestro ejemplo básico, sería el siguiente (listado 2):

```
<div id="silverlightControlHost">
  <object data="data:application/x-silverlight-2," type="application/x-silverlight-2"
    width="100%" height="100%">
    <param name="source" value="ClientBin/Ejemplo1.xap" />
    <param name="onError" value="onSilverlightError" />
    <param name="background" value="white" />
    <param name="minRuntimeVersion" value="4.0.50401.0" />
    <param name="autoUpgrade" value="true" />
    <a href="http://go.microsoft.com/fwlink/?LinkId=149156&v=4.0.50401.0"
      style="text-decoration: none">
      </a>
    </object>
    <iframe id="_sl_historyFrame" style="visibility: hidden; height: 0px; width: 0px;
      border: 0px"></iframe>
  </div>
```

Listado 2

Téngase en cuenta que, si bien el IDE aloja el complemento dentro de una etiqueta **<div>** esto no es imprescindible, y cualquier elemento capaz de albergar una etiqueta **<object>** de HTML puede servir para este propósito.

En cuanto a la inclusión de un elemento **<iframe>** a continuación de la etiqueta **<object>**, baste comentar que se trata de un truco para evitar un error que se produce con algunas versiones del navegador Safari: en concreto, impide que Safari almacene la página en la memoria caché, lo que imposibilitaría que el complemento se recargase al volver nuevamente a una página visitada con anterioridad.

El lector quizá haya observado que existe un fichero en el sitio web llamado **silverlight.js**. Es el que proporciona las funciones auxiliares de JavaScript y permite la llamada inicial al complemento entre otras funciones.

Si todo va correctamente, (y hemos cambiado el color de fondo de nuestro diseño, al ejecutar obtendremos simplemente, una pantalla del navegador con el fondo en ese color.

El papel de Expression Blend 4.0 en el desarrollo y diseño

Todo lo anterior, nos permite programar en un entorno de gran potencia y simplifica mucho las cosas, pero para un control total de los elementos de la interfaz y —sobre todo— para la creación y pruebas con algunos elementos, Visual Studio suministra las capacidades habituales, pero no los elementos específicos a los que los diseñadores visuales están acostumbrados.

Por ejemplo para crear gráficos irregulares (elementos **<Path>**) que respondan a eventos como un control cualquiera, o para crear animaciones y ver cómo se comportan sin necesidad de ejecutarlas, (y muchas otras tareas por el estilo) la herramienta más adecuada es **Microsoft Expresión Blend 4.0**.

Además, para los diseñadores de interfaces visuales acostumbrados a utilizar otros recursos del mercado, Expression Blend permite la importación a lenguaje XAML de elementos creados con otros entornos populares como los de la suite de Adobe, por ejemplo.

Resumen

Entre ambas herramientas, disponemos de un marco completo para la construcción de aplicaciones, cubriendo cualquier necesidad que puedan surgir en el ciclo de vida, tanto para el desarrollador individual, como para un gran equipo de trabajo, que necesita prestaciones de grupo más avanzadas.

En el siguiente capítulo, comenzaremos por el estudio del lenguaje XAML y la construcción de interfaces de usuario básicas.

El lenguaje XAML

XAML es un lenguaje de marcado para presentación; esto es, un lenguaje que sirve para definir los elementos visuales de una interfaz de usuario. Sus siglas son un acrónimo de **eXtended Application Markup Language**, y debutó junto a Windows Presentation Foundation, con la idea de permitir al usuario construir el contexto visual de una interfaz de usuario con elementos vectoriales, sin perder calidad a la hora de mostrarse en los potenciales clientes.

Con la evolución de .NET y la aparición de Silverlight, los creadores de esta plataforma utilizaron desde el inicio un subconjunto de XAML para el mismo propósito. Si bien en la actualidad el lenguaje XAML propio de Silverlight sigue siendo un subconjunto del utilizado en WPF, la verdad es que cada vez se aproxima más a su antecesor desde el punto de vista funcional y sintáctico.

Lenguaje declarativo, de marcas y basado en XML

Lo primero a recordar es que es un lenguaje declarativo. Esto es, establece definiciones de elementos visuales y de sus propiedades o atributos, aunque no permite la creación dinámica de esos elementos (aunque esto se puede conseguir utilizando el lenguaje .NET subyacente). Y, además, utiliza —igual que XML— una estructura jerárquica de elementos basada en contenedores.

Esto es, siempre habrá un elemento de orden superior, que contenga a otros, y estos, a su vez, a otros y así sucesivamente. Este modelo resulta particularmente adecuado para las interfaces de usuario visuales, donde un elemento (ventana) contiene a otros (controles), que a su vez hacen lo mismo con otros sub-controles.

Para conseguirlo, se basa en una sintaxis universal, como es XML¹ que es también jerárquica (todo documento XML debe de tener un contenedor global o nodo

¹ En realidad, XML no se construyó a partir de cero, sino que se basa en un lenguaje de marcas anterior llamado SGML (*Standard Generalized Markup Language* o Lenguaje de Marcas Estándar Generalizado).

raíz). La entidad responsable de su definición y mantenimiento (www.w3c.com) lo define como "un mecanismo de carácter sintáctico para definir lenguajes de marcas para cualquier uso". De ahí que se utilizase XML para la construcción de XAML. Y de ahí que XAML sea —casi al 100%— sensible a la diferencia entre mayúsculas y minúsculas, tal y como lo es XML. Las únicas excepciones están en la definición de los valores de algunos atributos o propiedades, y esto, solo en algunos casos.

Al total de caracteres que delimitan totalmente una marca (etiqueta de inicio, atributos, contenido y etiqueta final) se le llama **elemento**. La especificación obliga a que todo documento XML posea al menos un elemento, que recibe el nombre de **elemento raíz**, *ninguna parte del cual puede aparecer como contenido de otro elemento* (y que, eventualmente, puede estar vacío).

Así, el más básico de los documentos XML que podemos imaginar —según la especificación— es aquel que contiene un único elemento (el raíz) vacío. En el caso de XAML estaríamos definiendo un contenedor sin contenido alguno, y podría ser un objeto **UserControl** (lo habitual) o un objeto **Page**, **ChildWindow** o **Application**.

De esta forma, entendemos por elemento el conjunto de fragmentos que componen la etiqueta o *tag*: La marca de cabecera o identificador, los atributos, el contenido de la etiqueta y la marca de cierre. La figura siguiente describe la estructura de un elemento `<h1>` de HTML:



figura 1 El concepto de elemento en un lenguaje de marcas

Como el cuerpo del elemento (el "HOLA" del gráfico anterior) puede contener otros elementos contenedores y no contenedores, la estructura jerárquica está garantizada. Esto supone en la práctica una complejidad sintáctica que es aliviada gracias a

que data del año 1986, y que estableció el instituto ISO como un mecanismo para permitir la creación de lenguajes de marcas de cualquier clase. RTF y HTML no son sino aplicaciones concretas, el primero en el contexto de los procesadores de texto y el segundo para Internet.

la labor de los editores y/o diseñadores XAML, que mantienen y soportan un sistema de IntelliSense, al tiempo que dilucidan visualmente lo definido, tal y como aparecerá después del proceso de interpretación visual (*rendering*).

El lenguaje XAML

Dicho esto, desde el punto de vista formal, un fichero XAML es, simplemente, un fichero de texto plano con sintaxis XML, cuyo único requisito formal inicial es que sea un documento XML bien formado² y que el nombre de su elemento raíz esté definido por el estándar. Un documento XAML puede tener cualquier extensión, aunque, por convención, asumimos que **.xaml** es la más descriptiva.

En la implementación de XAML, cada elemento que representa a un objeto visual es una clase, o sea que se trata de un lenguaje orientado a objetos y todas esas clases están definidas en las librerías del *runtime*. Usualmente (pero no obligatoriamente) suele disponer de una clase parcial complementaria, escrita en un lenguaje .NET válido: C#, VB.NET o un lenguaje dinámico, tal y como hemos visto antes.

Desde el punto de vista operativo, cualquier elemento definido en la parte XAML es reconocido por su clase subyacente, de forma que podemos hacer referencia a él por su identificador, y obtendremos en el editor de código fuente la lista de propiedades, métodos y eventos que tenga definidos.

También es perfectamente posible —y en ocasiones se hace así— crear elementos de la interfaz de usuario utilizando exclusivamente código .NET. No obstante, la perfecta conjunción de ambos se produce cuando cada uno ocupa el rol para que el que ha sido pensado: XAML para definir interfaces de usuario, y .NET para codificar los comportamientos y acciones de esa interfaz, al menos de forma general.

Vectorial vs. Mapa de Bits

Otra cuestión a tener presente es que todo dibujo o elemento de la IU creado con XAML es de carácter vectorial, y no un mapa de bits. Eso significa que no hay

² Los documentos XML admiten dos niveles de conformidad para garantizar su coherencia. Se dice que un documento XML está bien formado cuando cumple con los requisitos sintácticos del estándar XML publicado por la W3C. Se dice que es válido, cuando, estando bien formado, puede contrastarse su contenido con un esquema que lo define, sea del tipo XML-Schemas o DTD (*Document Type Definition*).

pérdidas de calidad por el cambio de tamaño (escalados) en ninguno de los dos sentidos. De eso se encarga el núcleo **Presentation Core**.

La lista de elementos disponibles en XAML para la construcción de la interfaz de usuario es, al día de hoy, bastante larga, pero siempre podríamos diferenciar los elementos por el propósito para el que sirven. Así, tendríamos elementos dinámicos y estáticos, contenedores y contenidos, etc. Comenzamos nuestra revisión con los elementos estáticos de la interfaz de usuario.

Elementos estáticos: la interfaz de usuario básica

Como anticipamos previamente, el fichero **.XAP** que generamos al crear una aplicación Silverlight 4, tiene como contenido principal una DLL cuyo punto de entrada es el constructor de la clase **App** (que hereda de **Application** y establece el contexto de trabajo). Este objeto, tiene un manejador de evento (**Application.Startup**), donde se produce la asignación de una nueva instancia de la clase **MainPage** al objeto contenedor de toda la parte visual: **RootVisual**.

Recordando el código generado por Visual Studio 2010 para esa clase, vemos que, en la parte XAML, el elemento principal de la interfaz es un objeto **UserControl**, con algunas propiedades asignadas:

```
<UserControl x:Class="Ejemplo1.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006">

  <Grid x:Name="LayoutRoot" Background="#FF60A8C0">
    <Grid>

</UserControl>
```

Lisatdo I

Este elemento **UserControl**, recibe un tamaño inicial predeterminado establecido por las propiedades **Width** y **Height** del control, y contiene un elemento de disposición visual (una rejilla o *Grid*, de nombre **LayoutRoot**), que sirve de contenedor del resto de elementos, y que aparece vacío.

Atributos de propiedad y elementos de propiedad

En XAML, podemos utilizar atributos directamente (tal y como aparecen en el ejemplo anterior) o podemos usar un mecanismo diferido de creación de atributos cuando se trata de objetos más complejos, y ese atributo se expresa como un conjunto de otros elementos XAML. Un ejemplo sencillo de funcionamiento podemos verlo al establecer el atributo de color de fondo (**Background**) del objeto **Grid** del ejemplo.

Si se trata de un dato simple y podemos expresarlo mediante una secuencia entrecomillada, usaremos la sintaxis inicial:

```
<Grid x:Name="LayoutRoot" Background="#FF60A8C0">
<Grid>
```

Listado 2

Pero si no es así, o si el color de fondo debe construirse a partir de otros elementos, como cuando deseamos mostrar un degradado entre dos colores, el atributo se puede expresar como un "elemento de propiedad", o sea como elementos XAML incluidos en la propiedad **Background**, que la definen sin ambigüedad. Como por ejemplo, en el listado 3:

```
<Grid x:Name="LayoutRoot">
  <Grid.Background>
    <LinearGradientBrush EndPoint="1,0.5" StartPoint="0,0.5">
      <GradientStop Color="Black" Offset="0" />
      <GradientStop Color="White" Offset="1" />
    </LinearGradientBrush>
  </Grid.Background>
</Grid>
```

Listado 3

Otro ejemplo sencillo de este patrón lo vemos al definir las filas y/o columnas de un objeto **Grid**. Si queremos que nuestra rejilla tenga dos columnas, podemos definirlas mediante el código del listado 4.

Y si deseamos que tenga más de una fila (la predeterminada), o si queremos que esa fila tenga unas dimensiones concretas (un valor dado) o se ajuste al contenido (*), utilizaríamos el código del listado 5.

```
<Grid.ColumnDefinitions>
  <ColumnDefinition Width="248*" />
  <ColumnDefinition Width="252*" />
</Grid.ColumnDefinitions>
```

Listado 4

```
<Grid.RowDefinitions>
  <RowDefinition Height="*" /> <!--RowDefinition solo tiene sentido aquí, -->
  <RowDefinition Height="*" /> <!--dentro del contenedor Grid.RowDefinitions -->
  <RowDefinition Height="50" />
</Grid.RowDefinitions>
```

Listado 5

Esto es algo muy común en XAML y además, permite definir estilos independientes, con sus propiedades y valores que luego pueden ser aplicados a uno o más elementos de la interfaz. Como vemos, el atributo **Background**, se describe como un elemento contenedor. Su contenido es una definición basada en otro elemento (**LinearGradientBrush**), que a su vez, contiene dos puntos de definición del gradiente. Se comienza por el color negro y se realiza una transición hacia el blanco utilizando todo el espacio disponible.

Esta técnica se usa continuamente en XAML para la definición de interfaces más complejas, y también para la definición de estilos visuales aplicables a más de un elemento.

Propiedades vinculadas (*Attached Properties*)

Una situación peculiar se produce cuando queremos referirnos a un valor del elemento contenedor **desde un elemento contenido**. Por ejemplo, queremos decir que nuestro elemento debiera de estar en la 2ª fila y no en la primera.

En XAML, existe lo que se denominan propiedades vinculadas. Esto es, propiedades de un elemento, que realmente apuntan a otra propiedad de su contenedor. Esto es muy útil cuando deseamos controlar la ubicación de un elemento en su contenedor. En nuestra rejilla con 3 filas, podemos indicar a un **TextBlock** interior que se ubique en la segunda fila mediante la sintaxis:

```
<TextBlock Text="Saludos" Grid.Row="1" />
```

Listado 6

Donde **Grid.Row** hace referencia a la segunda fila (todos los elementos comienzan por cero). Como el valor predeterminado de los atributos es asignado inicialmente en la creación del objeto, esas propiedades (de tipo **Int32**) asumen un valor de 0, lo que significa: "fila 1, columna 1", para cualquier elemento contenido donde no se indique otra cosa.

Elementos Contenedores

En XAML se distinguen dos tipos de elementos respecto a su capacidad para albergar otros, los llamados contenedores: **Grid**, **Canvas**, **StackPanel**, **WrapPanel**, **ViewBox**, etc., y el resto. La razón es que, aunque todos pueden contener otros elementos. Los no contenedores deben utilizar su propiedad **Content** y pueden alojar a un único elemento. ¿Cómo solucionar el problema cuando necesitamos más de un elemento, pongamos, dentro de un botón (que no es un contenedor puro)? Pues haciendo que esa propiedad **Content** tenga por contenido a un único elemento que sí sirva de contenedor, como los objetos **Grid**, **Canvas** o **StackPanel**.

Cada tipo de contenedor se adapta mejor a algunas circunstancias, y será el diseñador quien decida lo que mejor se adecúa a su situación. Por ejemplo, mientras los contenedores **Grid** ubican sus elementos mediante atributos de márgenes o alineaciones horizontal y vertical, el contenedor **Canvas** admite posicionamiento absoluto, esto es, podemos indicar a que distancia (en coordenadas cartesianas desde el origen —0,0—), queremos que se dibuje el elemento.

De esta forma, si deseamos que nuestro supuesto botón tenga una imagen, además de un texto, podremos usar una sintaxis como esta (listado 7).

```
<Button Width="180" Height="70">
  <Button.Content>
    <StackPanel Orientation="Horizontal" >
      <Image Source="W7_Logo.png" Width="32" />
      <TextBlock Text=" Pulsar" VerticalAlignment="Center" />
    </StackPanel>
  </Button.Content>
</Button>
```

Listado 7

Lo que nos producirá un botón con una apariencia similar a esta:

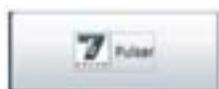


figura 2

Botón del ejemplo anterior visto en el diseñador

Este principio podemos aplicarlo a cualquier situación similar que implique varios contenidos. En el ejemplo del botón, hemos utilizado un contenedor **StackPanel** con orientación horizontal, para que la imagen salga primero y después el texto.

Espacios de nombres en XAML

En XAML, todos los elementos se corresponden con una clase de .NET incluida en el **CoreCLR**. Y sus atributos, expresados en sintaxis XML, se corresponden (decimos que se *mapean*), con propiedades o eventos de esos objetos. Lo contrario, sin ser cierto en su totalidad, sí que lo es de forma parcial, ya que, aquellas clases que disponen de un constructor por defecto y algún miembro público pueden ser instanciadas y usadas desde el lenguaje XAML, y de hecho esto es así en muchas ocasiones, sobre todo para referirnos a objetos de negocio, cuando deseamos mostrar datos.

El mecanismo por el cual podemos hacer referencia a esas clases ajenas a XAML es el mismo que utiliza XML: los **espacios de nombres**. Son declaraciones de una secuencia única de caracteres (se utilizan URI's muchas veces, pero podría ser cualquier otra) que se asocian con una especificación o una librería y sirve para definir sin ambigüedad qué elementos están permitidos en un fichero XML (o XAML, en este caso).

En la definición del control principal, notamos la presencia de varias declaraciones de espacios de nombres (**xmlns**): el predeterminado, que apunta al núcleo de presentación y uno secundario (prefijo **x**) que define extensiones de marcado. Usamos estas extensiones para nombrar elementos, asignar claves (**Keys**) que permitan referenciar después estilos u otros recursos programables, etc. Además existen otras que apuntan a librerías propias de Expression Blend.

nota

Con la evolución del lenguaje XAML a través de las distintas versiones, esta notación se suministra en muchos casos como un alias: *Name* (sin la *x*), aunque, internamente la operativa sea la misma en los dos casos.

Intellisense

El IDE de Visual Studio incluye Intellisense incluso en esta parte declarativa, por lo que, cuando asignamos manualmente un espacio de nombres en nuestros controles, se nos ofrecen los posibles elementos a referenciar, incluyendo, si hemos compilado la aplicación, la DLL generada en nuestro proyecto (figura 3).



figura 3 *Intellisense en la asignación de espacios de nombres desde Visual Studio 2010*

Atributos de Extensión de Marcado

En el segundo de los espacios de nombres definidos en el código (**x:**), se hace referencia a la librería que contiene las extensiones del lenguaje. Estas extensiones sirven sobre todo para crear sistemas de identificación de los elementos XAML y podríamos resumirlas en el cuadro siguiente (tabla 1).

Característica	Descripción
x:Class	Reúne los fragmentos de una clase parcial o identifica de forma única una clase definida solamente en lenguaje XAML
X:Key	Suministra un identificador único a recursos definidos en lenguaje XAML. Los identificadores deben comenzar por una letra o el símbolo de guión bajo (_) y solo pueden contener letras, dígitos o el propio guión bajo.
X:Name	Suministra una forma de identificar un elemento en XAML para poder referenciarlo desde el código subyacente (C#, en este caso). La normativa de nomenclatura es igual que en el caso anterior.
X:Null	Corresponde al valor null de C# o <i>Nothing</i> en VB.NET. Se puede utilizar como extensión de marcado ({x:Null}) o como un elemento de propiedad (<x:Null>) para asignar valores nulos en ciertos casos.

Tabla 1: Descripción de los atributos de extensión disponibles en XAML

En XAML utilizamos el atributo de extensión **x:Name** para asignar un nombre a cualquier elemento. Explicaremos esto con más detalle después, pero ahora es importante que quede claro (en XAML) el concepto de espacio de nombres o *namespace*.

En aplicaciones reales, el usuario creará sus propias clases de ayuda, clases de negocio y clases de objetos de acceso a datos o a servicios. La forma de referenciarlas desde XAML de manera que podamos usarlas de forma declarativa, será mediante *namespaces* personalizados, donde definimos prefijos que apuntan a elementos contenidos en las DLL correspondientes.

Otras extensiones de marcado

Binding

Define un destino de datos (un valor) para enlazar a una propiedad. Es una forma de diferir la asignación del valor de una propiedad, que incluso permite asignar a una propiedad el valor de otra. **Binding** crea un objeto intermedio de expresión e interpreta el enlace a datos que se aplica al elemento en tiempo de ejecución.

Propiedades vinculadas al enlace Binding

Binding usa la sintaxis `bindingPropertyName = value` y dispone de un conjunto de propiedades de lectura y escritura que modifican su comportamiento. Estas propiedades se pueden enumerar en cualquier orden e irán separadas por comas.

En la tabla siguiente se enumeran las propiedades disponibles de **Binding**. Algunos de los valores de propiedad necesitan tipos de objetos que no admiten una conversión de tipos nativa y exigen marcado dentro de la extensión **Binding** (tabla 2).

StaticResource

Asigna un valor para cualquier atributo de propiedad XAML al evaluar una referencia a un recurso definido en un objeto **ResourceDictionary** (que debe estar en el ámbito XAML del elemento).

Uso: `<object property="{StaticResource key}" .../>`

TemplateBinding

Vincula el valor de una propiedad de una plantilla de control con el valor de alguna otra propiedad expuesta en el control. Solo puede usarse en una definición de **ControlTemplate** en XAML. Un análisis mediante Blend de las propiedades internas de los controles predeterminados, muestra con claridad su utilización.

Propiedad	Descripción
Path	Especifica la ruta de acceso a la propiedad del origen de enlace. Se puede establecer la propiedad Path mediante una cadena que siga a Binding , como {Binding Employee.FirstName} o explícitamente, como {Binding Path=Employee.FirstName} .
Converter	Indica un convertidor predefinido por el lenguaje o por el usuario.
ConverterCulture	Especifica la referencia cultural a usar por un convertidor.
ConverterParameter	Especifica un parámetro para el convertidor que se puede usar en su lógica subyacente.
Mode	Especifica la forma del binding como una de las 3 siguientes: <i>OneTime</i> (una sola vez al inicio), <i>OneWay</i> (solo del objeto vinculado al elemento de la IU) o <i>TwoWay</i> (en los dos sentidos).
Source	Indica el origen de datos. Requiere una referencia de objeto, como una referencia StaticResource . Si no se especifica, el contexto de los datos establece el origen.
NotifyOnValidationError, ValidatesOnExceptions, ValidatesOnDataErrors ValidatesOnNotifyDataErrors	Estas 4 propiedades habilitan mecanismos de validación/notificación en el enlace. Pueden ser true o false (predeterminado=false).
ElementName	Especifica un origen de datos haciendo referencia a otro elemento (con una propiedad Name o un atributo x:Name). El elemento debe existir en el mismo ámbito XAML.
RelativeSource	Admite los valores Self o TemplatedParent . Permite indicar un origen de datos respecto a otro elemento de destino. Tanto si se indica como atributo o como propiedad de binding , al establecer la propiedad RelativeSource en XAML se necesita el uso de la extensión de marcado RelativeSource (ver descripción después de este cuadro).
UpdateSourceTrigger	Especifica la temporización de las actualizaciones del origen de enlace, como una de las cadenas siguientes: Default o Explicit .
StringFormat	Establece el formato de cadena a usar en la salida.
FallbackValue	Especifica el dato a mostrar cuando no se pueda resolver la ruta de acceso de origen (diferido).
TargetNullValue	Define el valor que se mostrará cuando el origen sea null .

Tabla 2: Lista de propiedades *Binding* disponibles

RelativeSource

Permite indicar el origen de un enlace en relación al gráfico de objetos (árbol XAML) en tiempo de ejecución. Admite las sintaxis:

```
<Binding RelativeSource="{RelativeSource Self}" .../>
0
<Binding RelativeSource="{RelativeSource TemplatedParent}" .../>
```

Una división fundamental: elementos de dibujo vs. controles

Al principio, cuando observamos código XAML, puede parecer que estamos en un entorno similar al de Windows Forms, solo que los elementos visuales son dibujados mediante un sistema de tipo vectorial y con ciertas peculiaridades que le hacen más flexible, pero no es así.

Si bien es cierto que Visual Studio ofrece una serie de controles predeterminados que simulan los más populares de Windows Forms, no lo es menos que son "diseños preparados" a partir de los elementos fundamentales de XAML, o sea, de los elementos de dibujo. Sin excepción.

El elemento base de XAML que sirve de fundamento para todos los demás elementos visuales es la clase abstracta **UIElement** (elemento de Interfaz de Usuario). De él, deriva **FrameworkElement**, que suministra un conjunto de API comunes que participan en la interpretación visual de los elementos XAML de Silverlight.

Controles

Y, a partir de aquí, se produce una división fundamental en dos "familias de objetos" visuales. Una, son los controles: heredan de la clase **Control** y suministran API propias para la encapsulación de funcionalidad y apariencia visual basadas en un objeto **ControlTemplate**, que constituye su plantilla.

Variando esta plantilla, se puede cambiar el aspecto de cualquier control. De modo que los controles que aparecen en la Barra de herramientas, no son más que eso, conjuntos de elementos más pequeños encapsulados dentro de una DLL y que ofrecen un aspecto parecido al de sus homónimos de Windows Forms. Pero no hay uno solo que no podamos modificar editando su plantilla con Expression Blend.

Shapes

La otra gran familia de objetos visuales es **Shape** (que también hereda de **FrameworkElement**). De ella derivan todos los elementos de dibujo vectorial 2D disponibles en la Caja de herramientas de Blend, y también los que no lo están, porque son el resultado de una "sesión de dibujo" (o de una codificación a través del mini-lenguaje M que suministra la clase **Path**, que deriva de **Shape**, aunque esto último es mucho más complejo).

La herencia de **FrameworkElement** garantiza la presencia de las API de los controles, y es posible hacer todo lo que se hace con aquellos. Se pueden crear en el editor a partir de otras **Shapes** disponibles o también dibujarlos directamente con Blend. Disponen de eventos programables y métodos, exactamente igual que los controles habituales de otros entornos.

nota

Como Blend está más centrado en el diseñador, el usuario observará elementos Shape disponibles que no se encuentran en Visual Studio. De todos modos, las DLL que los contienen siempre pueden referenciarse en XAML, por lo que esto no supone problema alguno.

El inevitable "Hola Mundo" como un dibujo que responde a eventos

Según esto, y si queremos seguir la tradición de escribir un "Hola Mundo" como primer programa, utilizando una de esas formas o *shapes*, lo mejor es utilizar Expression Blend 4.0. Seleccionamos una nueva aplicación Silverlight 4.0, y nos aparecerá la ventana de diseño visual con una superficie vacía basada en un *grid*, exactamente igual que en nuestro ejemplo anterior desde Visual Studio, pero con la nueva interfaz de Blend.

Una vez ahí, en la parte izquierda del IDE de Blend, aparece una lista de iconos que despliegan opciones de dibujo. Las 7 primeras (primera barra vertical de la figura 3) corresponden con herramientas de diseño y utilidades para el dibujo. Los iconos siguientes (segunda barra vertical) agrupan herramientas y controles predeterminados. Son elementos que podemos utilizar en la superficie de diseño de forma directa: rectángulos, elipses, objetos contenedores (**Grids**, **Canvas**, etc.), elementos de texto, controles y un subconjunto denominado "Activos" (*Assets* en la versión inglesa), donde se puede acceder a todos los elementos visuales a los que hacen referencia las li-

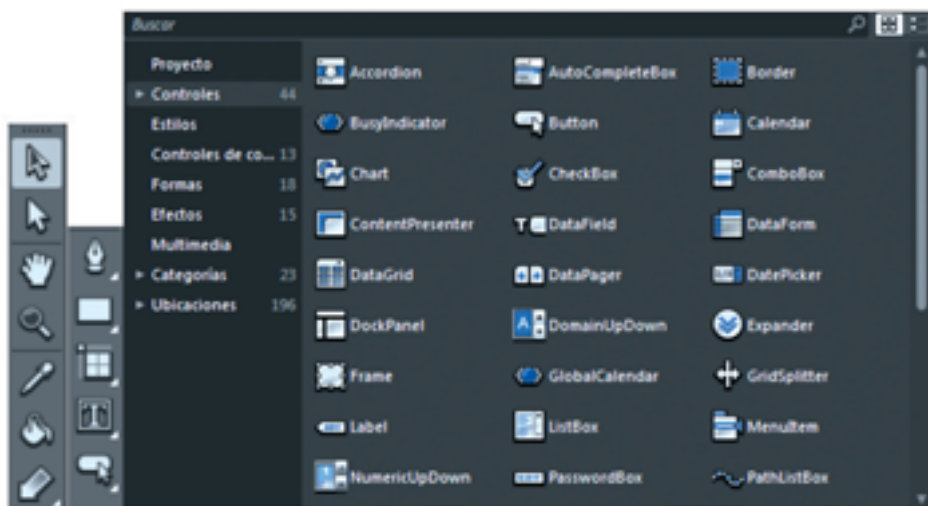


figura 4 Barras de Herramientas en Blend 4 y ventana desplegada de Activos (*Assets*)

brerías predeterminadas. Al lado, se encuentran otros recursos que también se utilizan en diseño, como las formas, los estilos, los efectos, los elementos multimedia, etc.

Bien, pues, para nuestro "Hola Mundo", lo que haremos es crearlo como si lo estuviéramos dibujando con un lápiz. Para ello, desplegaremos el icono de la pluma y veremos que el IDE nos ofrece dos herramientas de dibujo manual:

- Dibujo asistido de curvas "Bézier" (Pluma).
- Dibujo a mano alzada (Lápiz)



figura 4 Herramientas de dibujo en Blend 4: la pluma (para dibujo asistido) y el lápiz (mano alzada)

Utilizaremos la segunda, e intentaremos escribir la frase "hola mundo" haciendo que cada letra se corresponda con un solo trazo, por facilidad en el análisis y manipulación posterior. Lo de menos es que nos quede un resultado similar al de la escritura de un niño. Después de tan notable esfuerzo creativo (...), tendremos un resultado visual similar al siguiente:



figura 5 "Hola Mundo" en XAML dibujado con la herramienta *Pencil*

Si observamos el código XAML generado por nuestra sesión de dibujo, veremos que se han generado tantos elementos **Path**, como trazos hemos realizado. Un total de 11, teniendo en cuenta que el signo de admiración al final está compuesto por 2 trazos. El código tiene un aspecto similar a esto (cada dibujo es único, y por tanto, el lector obtendrá los mismos tipos de elementos (**Path**), pero valores distintos.

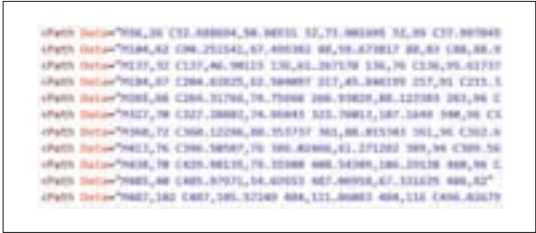


figura 6 Código XAML resultante del dibujo anterior

Cada trazo es un objeto **Path**. Y lo que ese objeto dibuja en pantalla, se guarda en notación vectorial en su atributo **Data**, utilizando un mini lenguaje interno de representación llamado M, en el que no vamos a entrar aquí, pero que describe cómo dibujar cada uno de los trazos en el mismo sentido en que nosotros lo hemos hecho con la herramienta. Esto permite volver a reproducir los trazos en cualquier otro sitio con total fidelidad y a cualquier escala y —al tratarse de un objeto— dispone de métodos y eventos a los que podemos llamar.

Programando un elemento *Path* en C#

De hecho, si seleccionamos uno cualquiera de esos objetos **Path** (cualquiera de las letras), veremos que, en la Ventana de propiedades aparece la configuración que

ese elemento tiene en la actualidad. Eso incluye colores de fondo y de primer plano, características de diseño, de orientación, etc. Podemos cambiar cualquiera de ellas, y también podemos recurrir a la pestaña eventos, en caso de que deseemos modificar el comportamiento de la aplicación en tiempo de ejecución.

En la parte superior de la Ventana de propiedades, podemos seleccionar el icono a continuación del actual  (el del rayo, que nos mostrará los eventos disponibles para el objeto **Path** activo), donde veremos los eventos definidos que no son sino los disponibles en **FrameworkElement**. Supongamos que deseamos programarlo para que cuando el cursor pase por encima de la letra "m" se lance una caja de diálogo del sistema recordando que aquello se ha puesto en marcha y el elemento ha reconocido la acción de usuario.

Una vez en la ventana de eventos, podríamos utilizar el evento **MouseEnter** correspondiente a la letra "M". Programarlo es bien sencillo: escribimos el nombre de un procedimiento al lado del evento (por ejemplo **Saludar**), y nos vamos al código C# subyacente, donde se habrá creado el manejador de evento con el nombre indicado.

Finalmente, para hacer que se ponga en marcha la caja de diálogo añadiremos una llamada al método **Show** de la instancia estática de la clase **Message** de la siguiente forma:

```
private void Saludar(object sender, System.Windows.Input.MouseEventArgs e)
{
    MessageBox.Show("Saludos desde Silverlight");
}
```

Listado 8

Cuando, ejecutemos el proyecto, veremos como al pasar el cursor por la letra M, se lanza la caja de diálogo correspondiente, en forma similar a como esperaríamos del comportamiento de una aplicación ASP.NET o Windows Forms. (figura 7):

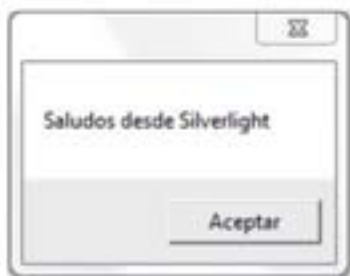


figura 7

Caja de diálogo resultante del código anterior.

Si en lugar del lápiz hubiéramos utilizado la pluma el resultado hubiera sido el mismo, desde el punto de vista del código generado: una serie de objetos **Path**, que describen la definición vectorial que hemos generado con la herramienta. La razón de que este objeto disponga de esa larga lista de eventos que hemos visto, es que también hereda de **Shape**.

Si queremos un resultado más profesional o más a tono con lo habitual, la solución estará en utilizar los controles ya creados capaces de mostrar texto. Lo más fácil es utilizar el bloque de texto (control **TextBlock**), asociado al icono  de la barra de herramientas, y anotar en su propiedad **Text** lo que queramos mostrar. El control, dispondrá además, de los atributos propios de configuración de una tarea de este estilo, como selección del tipo de fuente, tamaño de ésta, efectos de la fuente, colores de fondo y primer plano, y muchas otras más.

En este caso, el código resultante sería mucho más simple, y probablemente similar al de nuestro primer ejemplo, con Visual Studio (listado 9):

```
<TextBlock Height="74" HorizontalAlignment="Left" Margin="45,114,0,0" Name="textBlock1"
Text="Saludos desde Silverlight !" VerticalAlignment="Top" Width="327"
FontWeight="Bold" Foreground="#FF6F1515" FontSize="48" />
```

Listado 9

Omitimos la salida correspondiente por pantalla, que el lector puede imaginarse fácilmente. La cuestión es que —en ambos casos— estamos utilizando objetos de definición visual de tipo vectorial, con métodos y eventos asociados (como cualquier otro objeto en .NET) y que permiten su programación posterior en un lenguaje como C#.

nota

La capacidad de edición visual de objetos **Path** está limitada a *Expression Blend*, no disponiendo —de momento— Visual Studio de esa posibilidad, ya que se supone que es una opción más vinculada con los diseñadores visuales o de interfaces de usuario.

Las jerarquías descendientes de **Shape**

Aparte de lo que podríamos hacer con este objeto (realmente, cualquier cosa), el trabajo diario suele llevarnos a manejar elementos ya contruidos (controles) más próximos a nuestras necesidades si estamos diseñando aplicaciones de negocio. Pero con-

viene tener presentes a aquellos elementos que derivan de esta clase base y suministran los fundamentos del conjunto de controles típicos (o de los nuevos que deseemos crear ...).

De esta forma, derivan de **Shape** otros elementos de dibujo fundamentales que hacen de "ladrillos de construcción" de todo lo demás, **Ellipse**, **Line**, **Polygon**, **PolyLine** y **Rectangle**. El diagrama jerárquico es el siguiente (figura 8):

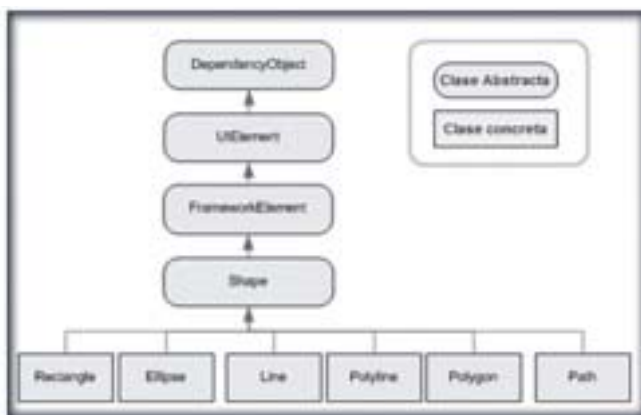


figura 8 Diagrama jerárquico de clases de los objetos *Shape*

Vamos a revisar someramente las formas disponibles, las propiedades comunes de que disponen y a poner algunos ejemplos de uso. Más adelante, trataremos las diferencias y modo de utilización de las geometrías.

A diferencia de los objetos **Geometry** (que veremos a continuación), los objetos **Shape** disponen de los recursos necesarios para poder dibujarse a sí mismos, y su posición en la jerarquía de clases, les otorga otras ventajas programáticas, como la capacidad utilizar cualquier contenedor válido (igual que cualquier **UIElement**) y, sobre todo, de soportar los mismos eventos que sus homólogos.

El conjunto de propiedades comunes más destacables, son las siguientes:

- **Stroke**. Describe cómo se dibuja el borde de la forma
- **StrokeThickness**. Establece el grosor del borde de la forma.
- **Fill**. Indica cómo se rellena el interior de las formas.

Propiedades de datos que permiten especificar coordenadas y vértices medidos en píxeles independientes del dispositivo.

Los objetos **Shape** suelen estar ubicados dentro de elementos **Canvas**, ya que el posicionamiento absoluto suele ser necesario para el dibujo, pero no hay problema para que se encuentren incrustadas en otros contenedores. Ya hemos visto el código para dibujar un rectángulo según estos principios, y podemos aplicarlo al resto de elementos. Por ejemplo para obtener una elipse sin relleno, (dentro de un **Grid**), nos bastaría con esto (listado 10):

```
<Grid x:Name="LayoutRoot">
  <Ellipse Margin="0,0,208,224" Fill="Yellow" Stroke="#FF000000" StrokeThickness="3"/>
</Grid>
```

Listado 10

Y si queremos ubicar varios objetos **Shape** en un contenedor tipo **StackPanel**, para que se presenten verticalmente, el problema no es muy complicado (listado 11):

```
<Grid x:Name="LayoutRoot">
  <StackPanel Orientation="Vertical">
    <Ellipse Fill="Navy" Stroke="#FF000000" StrokeThickness="3"
      Height="90" Width="180" />
    <Ellipse Fill="Yellow" Stroke="#FF000000" StrokeThickness="3"
      Width="120" Height="120" />
    <Ellipse Fill="Red" Stroke="#FF000000" StrokeThickness="3"
      Width="100" Height="180" />
  </StackPanel>
</Grid>
```

Listado 11

Y tendríamos una salida en el navegador similar a la siguiente (figura 9).

Podemos utilizar la propiedad **Stretch** para determinar la manera en que la forma aprovecha el espacio de que dispone a su alrededor. Sus valores pueden ser:

- *None*. Ninguno
- *Fill*. Relleno completo del contenedor modificando las propiedades **Width** y **Height**.

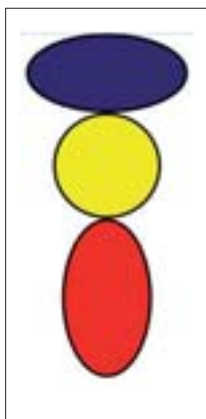


figura 9 Disposición final de los 3 objetos elipse del código anterior

- *Uniform*. Se cambia el tamaño de la anchura y la altura, pero, proporcionalmente a la figura, hasta que uno de los dos llegue a los bordes del contenedor.
- *UniformToFill*. Se cambia el tamaño de la anchura y la altura hasta que la forma *rellena* todo el espacio disponible.

A continuación vemos cómo las 3 elipses anteriores se ubican cada una en una fila de un **Grid**, y no se asigna ningún valor a las propiedades **Width** y **Height**, pero se establecen distintos modos de relleno (listado 12).

```
<Ellipse Grid.Row="0" Fill="Navy" Stretch="UniformToFill"
  Stroke="#FF000000" StrokeThickness="3" />
<Ellipse Grid.Row="1" Fill="Yellow" Stretch="Fill"
  Stroke="#FF000000" StrokeThickness="3" />
<Ellipse Grid.Row="2" Fill="Red" Stretch="Uniform"
  Stroke="#FF000000" StrokeThickness="3" />
```

Listado 12

La diferencia resultante puede apreciarse perfectamente incluso en la ventana de diseño de Visual Studio (figura 10).

Como vemos en el gráfico, en la elipse inicial, el crecimiento es uniforme (radial), pero en la figura resultante solo se muestra la parte que cabe en su contenedor. La elipse central establece un crecimiento no uniforme hasta alcanzar los límites de

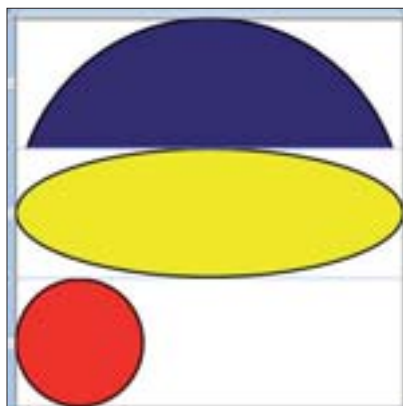


figura 10

Las 3 elipses mostrando el efecto de aplicar su atributo *Stretch*, sin valores de anchura o altura

la fila; como ésta es un rectángulo, obtenemos la elipse inscrita correspondiente. La tercera ha crecido uniformemente en ambas propiedades, lo que ha resultado en un crecimiento radial a partir de un foco único, convirtiéndose en un círculo.

Los objetos Line, PolyLine y Polygon

Como podemos intuir por sus nombres, sirven para el trazado de líneas, líneas quebradas (abiertas o cerradas) y polígonos. Disponen de propiedades específicas para indicar el grosor del trazo (**StrokeThickness**), el color del trazo (**Stroke**), la forma de los extremos de las líneas (**LineCap**), la forma de las uniones entre dos trazos (**LineJoin**), la posibilidad de que el trazo sea una línea de puntos (**Dashes**), y, especialmente, la propiedad que permite definir los trazos de la línea en términos de coordenadas (**Points**). Muchos de los principios de posicionamiento comentados antes, son perfectamente válidos aquí.

En el caso de los polígonos, disponemos también de las propiedades **Fill** (para el color de relleno del interior del polígono) y **FillRule**, que determina un criterio para seleccionar qué partes de un polígono se rellenan y cuáles no, dependiendo del número de líneas que haya que cruzar para salir del polígono desde la zona a rellenar.

En el ejemplo siguiente, se trazan varios ejemplos de líneas, **Polylines** y un polígono que resulta ser una estrella de cinco puntas. Como no se puede salir de la zona central de la estrella sin atravesar dos líneas, al escoger el atributo **FillRule= "EvenOdd"**, dicha zona central queda sin rellenar, como puede verse en la forma de estrella de la figura 12. La otra opción disponible (**NonZero**), lo hubiera rellenado como el resto de secciones (listado 13).

Y, a continuación mostramos la salida correspondiente al código anterior (figura 11), con todas las formas poligonales.

```

<Canvas x:Name="LayoutRoot" Background="White" >
  <Line Stroke="Maroon" StrokeThickness="5" X1="10" Y1="10" X2="220" Y2="100" />
  <Line Stroke="Red" StrokeThickness="5" X1="0" Y1="0" X2="110" Y2="100"
    Canvas.Left="25" Canvas.Top="100" />
  <Line Stroke="Blue" StrokeThickness="5" X1="25" Y1="100" X2="110" Y2="100" />
  <Polyline Stroke="Navy" StrokeThickness="5"
    Points="265,50 400,200 375,100 500,150" />
  <Polyline Stroke="Navy" StrokeThickness="5"
    Points="300,300 415,370 475,200 500,250 300,300" />
  <Polygon Stroke="Violet" StrokeThickness="5" Fill="Aqua"
    Canvas.Left="100" Canvas.Top="175" FillRule="EvenOdd"
    Points="15,200 68,70 110,200 0,125 135,125" />
</Canvas>

```

Listado 13



figura 11 Ejemplo de líneas, poli-líneas y polígono

Geometries: elementos de definición visual

Obviamente, ni los elementos vistos hasta ahora, ni los controles que veremos más adelante, son adecuados para la creación de ciertos dibujos complejos de tipo vectorial. Para ganar más flexibilidad y potencia en la definición visual de elementos, XAML suministra unos objetos llamados **Geometries**, capaces de definir contenidos visuales, aunque no se puedan mostrar a sí mismos, por lo que requieren de un objeto contenedor que lo haga. Esto es debido a que no heredan de **UIElement**, a diferencia de todos los anteriores. Por tanto, no disponen de propiedades tales como **Opacity**, **OpacityMask**, **Clip**, **RenderSize**, etc.

Las geometrías son objetos que heredan de la clase abstracta **Geometry** y aparecen en identificadas en la jerarquía con un sufijo **Geometry** (hay un total de 5, de los cuales, el último es realmente un contenedor para los anteriores).

- **LineGeometry**
- **RectangleGeometry**
- **EllipseGeometry**
- **PathGeometry**
- **GeometryGroup**

Las geometrías son más versátiles que las formas, si bien el hecho de que no puedan dibujarse a sí mismas, y que solo definan sus propiedades visuales, exige que sea otro objeto el encargado de esa interpretación visual, y por ello siempre van incluidas en un **UIElement** que hace de contenedor e intérprete visual.

Por ejemplo, un objeto **PathGeometry** puede definir todos los puntos de su contenido en su atributo **Data**, pero usaremos un objeto **Path** —tipo **Shape**— para que lo dibuje. Las geometrías son muy útiles igualmente para definir regiones de corte (*Clipping areas*).

De hecho, hay dos propiedades de los **FrameworkElement** que admiten como valor un objeto **geometry**: **Path.Data** y **UIElement.Clip**. Ahora bien, al ser **Geometry** una clase abstracta, debemos de utilizar alguna de sus clases derivadas o heredar nosotros de **Geometry**³. Las clases derivadas son las siguientes:

- **LineGeometry**. Representa una línea. Equivalente a la forma *Line*.
- **RectangleGeometry**. Representa un rectángulo. Equivale a la forma *Rectangle* opcionalmente con esquinas redondeadas.
- **EllipseGeometry**. Representa una elipse. Equivale a la forma *Ellipse*.
- **PathGeometry**. Representa una figura compleja compuesta de arcos, curvas y líneas y que puede ser abierta o cerrada.
- **GeometryGroup**. Añade cualquier número de objetos **Geometry** a un solo **Path** usando la propiedad **FillRule** para determinar qué regiones rellenar.

nota

Esto tiene una ventaja adicional, y es que —cuando usemos la propiedad **Data** de un objeto **Path** para definir formas—, si estas son de tipo estándar, no se necesita utilizar el lenguaje M indicado en el ejemplo de "Hola Mundo", sino que podemos recurrir a estas definiciones.

³ Téngase en cuenta que —hasta esta versión 4.0 de Silverlight— la clase *Shape* no podía heredarse, ya que estaba marcada como *Sealed*.

Esto es, la forma de dibujar un rectángulo mediante un objeto **Path** que use un elemento **Geometry** sería similar a esta (listado 14):

```
<Path Fill="Yellow" Stroke="Blue">
    <Path.Data>
        <RectangleGeometry Rect="110,100 200,350"></RectangleGeometry>
    </Path.Data>
</Path>
```

Listado 14

Y la mejor manera de combinar dos geometrías, será utilizando un elemento **GeometryGroup** (listado 15):

```
<Path Fill="Yellow" Stroke="Blue" Margin="5" Canvas.Top="10" Canvas.Left="10" >
    <Path.Data>
        <GeometryGroup>
            <RectangleGeometry Rect="0,10 100,100" />
            <EllipseGeometry Center="110,350" />
        </GeometryGroup>
    </Path.Data>
</Path>
```

Listado 15

De igual forma, podemos usar objetos **UIElement** para aplicar un objeto **Geometry** a su propiedad **Clip**, obteniendo mecanismos para la creación de recortes. Por ejemplo, para recortar una imagen, utilizando un objeto **Image** y esta técnica podríamos construir lo siguiente (listado 16):

```
<Grid x:Name="LayoutRoot" Background="Beige">
    <Canvas>
        <Image Source="imagenes/gracias.jpg" Width="200" Height="150" Canvas.Top="25">
            <Image.Clip>
                <EllipseGeometry RadiusX="100" RadiusY="75" Center="100,75"/>
            </Image.Clip>
        </Image>
    </Canvas>
</Grid>
```

Listado 16



figura 12 Imagen con recorte utilizando un objeto *Geometry*

Que nos ofrecería un resultado visual de la figura 12.

Realmente, aquí estamos jugando con dos conceptos distintos para producir el recorte: con el objeto **EllipseGeometry** para obtener los recortes superior e inferior, y con el tamaño de la foto respecto al del **Canvas** para producir los recortes laterales. Como podrá darse cuenta el lector cuando pruebe todo esto, las posibilidades son innumerables. Y de la misma forma, podemos utilizar otros objetos **Geometry** para generar casi cualquier tipo de composición.

Llegados aquí, la cuestión es ¿y qué pasa con el dibujo complejo de tipo vectorial? ¿Es suficiente con estos recursos gráficos? Ignoro si, con mucha paciencia, podría llegar a hacerse cualquier cosa, pero en ciertas áreas, como en el diseño industrial, interiorismo, publicidad, etc., el trabajo sería muy arduo. Afortunadamente, tenemos otro poderosísimo recurso para sacarnos del atolladero: El objeto **Path** cuando se usa conjuntamente con un objeto **PathGeometry**.

Dibujos vectoriales y el objeto **PathGeometry**

Es el objeto de dibujo por excelencia. Con él, se puede dibujar lo que sea, si bien, la sintaxis es más complicada, y también más largo el proceso de definición del contenido. Por suerte, muchas herramientas de dibujo preparadas para trabajar con XAML como Expression Blend, Expression Design o Adobe Illustrator, pueden almacenar sus datos en formato XAML y usarlos después de forma sencilla. Al final de este capítulo comentaremos cómo generar estos ficheros más complejos, pero veamos lo fundamental del uso de este objeto.

Cada objeto **PathGeometry** dispone de una propiedad llamada **Figures** (una colección) que puede almacenar tantos objetos **PathFigure** como necesite. Y cada **PathFigure** puede contener cualquier número de líneas o curvas tanto cerradas como abiertas. Cada uno de ellos, presenta 4 propiedades fundamentales:

- **StartPoint.** Un elemento **Point** que indica el comienzo de la figura.
- **Segments.** Una colección de objetos **PathSegment** usados para dibujar.
- **IsClosed.** Valor *booleano*, que, si es verdadero, hace que Silverlight añada una línea recta para conectar los puntos de inicio en fin, en el caso de que no coincidan.
- **IsFilled.** Valor *booleano*, que si es verdadero, hace que el área interior de la figura se rellene utilizando el valor indicado en la propiedad **Path.Fill**.

A su vez, cada uno de los segmentos de que consta la figura puede ser de alguna de las clases siguientes:

- **LineSegment.** Crea una línea recta entre dos puntos.
- **ArcSegment.** Crea un arco de elipse entre dos puntos.
- **BezierSegment.** Crea una curva Bézier entre dos puntos.
- **QuadraticBezierSegment.** Crea una curva Bézier con un punto de control en lugar de dos, con lo que resulta más fácil (y rápido) de calcular.
- **PolyLineSegment.** Crea series de líneas rectas. Se puede hacer lo mismo con múltiples objetos **LineSegment**, pero esta solución es más concisa.
- **PolyBezierSegment.** Crea series de curvas Bézier.
- **PolyQuadraticBezierSegment.** Crea series de curvas Bézier más simples.

Con ésta técnica, el código siguiente dibuja un triángulo rectángulo mediante dos líneas muy simples y después fuerza el cierre de la figura mediante el atributo **IsClosed** (listado 17).

Obteniendo esta salida en la Ventana de diseño (figura. 13).

```
<Canvas>
  <Path Stroke="Blue" StrokeThickness="3">
    <Path.Data>
      <PathGeometry>
        <PathFigure IsClosed="True" StartPoint="50,50">
          <LineSegment Point="50,150" />
          <LineSegment Point="100,150" />
        </PathFigure>
      </PathGeometry>
    </Path.Data>
  </Path>
</Canvas>
```

Listado 17



figura 13 Salida del código anterior

Lógicamente, no era necesario recurrir a un objeto **Geometry** para dibujar un triángulo, pero en el código se aprecia una forma sencilla de programar este tipo de objetos.

Arcos

Los arcos se consiguen asumiendo que cada uno de ellos es, en realidad, un segmento de una elipse, de la que tenemos que suministrar su tamaño (**ArcSegment.Size**), el punto de inicio a partir del que queremos dibujar el segmento, y el punto que define el final del segmento. El punto final se identifica mediante el objeto **ArcSegment.Point** y el tamaño de la elipse. Como vemos en el ejemplo siguiente, una vez entendido lo anterior, el código resulta sencillo de entender (listado 18).

```
<Path Stroke="Blue" StrokeThickness="3">
  <Path.Data>
    <PathGeometry>
      <PathFigure IsClosed="False" StartPoint="10,10" >
        <ArcSegment Point="250,150" Size="200,300" />
      </PathFigure>
    </PathGeometry>
  </Path.Data>
</Path>
```

Listado 18

Lo que genera un único arco con la forma que se aprecia en la figura 14-a.

Hay dos propiedades (en este ejemplo) que se han asumido de forma implícita, para seleccionar el segmento más corto de la curva posible (en lugar del más largo) y para el sentido de la curvatura. En el primer caso, se asumió el valor **ArcSegment.Is-**

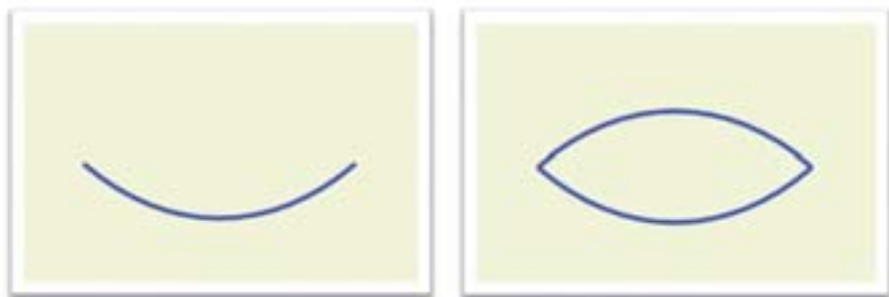


figura 14a-14b

Dibujo de un arco a partir de los puntos inicial y final y el tamaño de la elipse asociada. La segunda imagen se obtiene duplicando el código anterior y cambiando la propiedad *SweepDirection*

LargeArc predeterminado (o sea, falso), con lo que se seleccionó el arco más corto. Incluso así, hay dos formas de dibujar ese arco entre dos puntos y cada una produce una curvatura opuesta a la anterior.

Este valor se decide con la propiedad **ArcSegment.SweepDirection**, que puede adoptar los valores **Clockwise** (sentido de las agujas del reloj) y **CounterClockwise** (sentido contrario). De hecho, para ver esta diferencia basta con duplicar el código del arco y cambiar esta propiedad para obtener dos arcos iguales, pero complementarios, como muestra la figura 14-b.

Curvas Bézier

Se trata de una curva paramétrica estudiada en Análisis Numérico, muy importante en computación gráfica y sus campos relacionados⁴. Las curvas Bézier conectan dos segmentos lineales usando una función matemática compleja que incorpora dos puntos de control (o *n* puntos, generalizando para otras situaciones) para determinar cómo se dibuja la curva.

Son muy flexibles y, a partir de un punto de entrada, uno final y dos puntos de control, podemos construir una variedad enorme de curvas suaves (continuas o derivables). La imagen de la figura 15 da una idea —siquiera intuitiva— del proceso a seguir en la construcción de este tipo de curvas.

⁴ Para más información, ver http://en.wikipedia.org/wiki/Bezier_curve

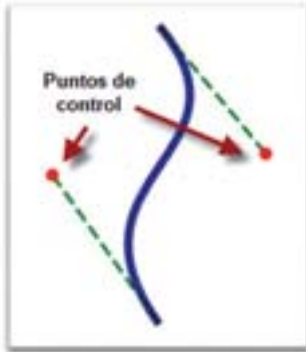


figura 15

Curva Bézier con sus puntos de control

El control del dibujo se realiza a partir de 3 puntos suministrados a la curva (más el inicial que lo aporta el contenedor): Los dos puntos de control, llamados **BezierSegment.Point1** y **BezierSegment.Point2**, y el punto final (**BezierSegment.Point3**). En el siguiente ejemplo (listado 19), se utilizan dos curvas en las que la segunda da comienzo en el punto donde termina la primera para generar una curva totalmente irregular, como la de la figura 16.

```
<Path Stroke="Blue" StrokeThickness="5" Canvas.Top="20">
  <Path.Data>
    <PathGeometry>
      <PathFigure StartPoint="100,150">
        <BezierSegment Point1="20,90" Point2="40,240" Point3="50,50" />
        <BezierSegment Point1="50,50" Point2="40,140" Point3="210,50" />
      </PathFigure>
    </PathGeometry>
  </Path.Data>
</Path>
```

Listado 19

Advierta el lector que el punto 3 de la primera curva es el mismo que el punto 1 de la segunda, haciendo que ésta comience el proceso de dibujo donde termina aquella. La primera curva tiene un perfil suave de tipo paraboloide mientras que la segunda forma un trazo más abrupto debido a la posición de uno de los puntos de control.

El trabajo con estas estructuras es complejo incluso para el diseñador avezado. Lo idóneo es utilizar, herramientas pensadas para ello, capaces de exportar sus datos a formato XAML como Expression Design o Adobe Illustrator.



figura 16 Dos curvas Bézier concatenadas

nota

De hecho, existen ya numerosas herramientas capaces de importar y exportar ficheros desde otros formatos a XAML. Una relación más o menos actualizada, puede encontrarse en la dirección <http://www.realsoftwaredevelopment.com/the-complete-list-of-xaml-tools>.

Precisamente con el propósito de manejar estos elementos cuando la complejidad de los gráficos crece, se creó ese "mini-lenguaje" para los objetos **PathGeometry**, que permite abreviar la descripción de los gráficos pensando en esas herramientas. En el manual de referencia dispone el lector de una explicación detallada de este mini-lenguaje si es de su interés.

Una nota sobre la conversión automática de formatos en ficheros XPS

El formato XPS, es un estándar de Microsoft para generar documentos cerrados, listos para ser impresos, que utilizan XAML como formato para el motor de visualización. Está soportado a partir de Office 2007, como una opción a la hora de guardar un fichero (si no dispone de ese complemento, puede descargarlo de <http://tinyurl.com/y69y7g>) y por Windows Vista y sistemas posteriores. Pues bien, Word (u otro programa de Office 2007 o posteriores) es capaz de guardar en ese formato cualquiera de sus documentos. En el caso de que se haya incluido en el documento un fichero gráfico de carácter vectorial, como por ejemplo, ficheros de la galería de imágenes con formato .WMF (*Windows Metafile*), u otro formato vectorial compatible, el fichero WMF será convertido a XAML y podremos usarlo en nuestras aplicaciones sin más que un mínimo retoque.

Los gráficos vectoriales asociados con el documento se encuentran en la carpeta **Pages** de esa estructura de directorios (algo similar a **Doc1\Documents\1\Pages**). Ahí se encontrará un fichero con la extensión **.fpage**. Ese es el gráfico en formato XAML.

nota

El formato XPS es, en realidad, una forma de encapsular un conjunto de ficheros y recursos asociados a un documento. Su formato comprimido es de tipo ZIP, de forma que si cambiamos su extensión a .zip y lo descomprimos, veremos un montón de carpetas con los recursos y datos que contiene.

Sólo hay que hacer dos pequeños retoques: eliminar el elemento raíz (llamado **Fixed-Page**), que Silverlight no reconoce y eliminar todos los atributos **BidiLine** que se encuentren (si hay alguno). El resto, es un conjunto de objetos **Canvas** y el gráfico en puro formato XAML. Una vez hecho eso funcionará perfectamente.

Incluso ficheros complejos como el de la figura 17, pueden suponer apenas 200 Kb en tamaño, y si el lector hace esta prueba verá la cantidad ingente de código que se produce usando objetos **Geometry** y sintaxis de mini-lenguaje.



figura 17

Fichero *j0149407.wmf* de la Galería de Windows, convertido a XAML y mostrado por Silverlight

Creación de shapes definidas por el usuario

Una vez comprendido el funcionamiento de las geometrías y las formas, estamos en condiciones de extender la funcionalidad predeterminada que ofrecen utilizando los mecanismos clásicos de la herencia.

Y es que, comenzando con esta versión, una de las restricciones asociadas con los objetos **Path** ha desaparecido. En versiones anteriores, la clase **Path** estaba marcada

como **Sealed** (**Non-Inheritable**, en VB.NET). Al desaparecer esta limitación, ahora podemos heredar de ella, y crear nuevas formas sin tener que re-implementar propiedades como **Fill**, **Stroke**, **StrokeFill**, etc.

Gracias a esta nueva capacidad, basta con suministrar los manejadores para los eventos **Measure** y **Arrange** y sub-clasificar **Path.Data** para mostrar lo que se quiera. (Téngase en cuenta que las formas existentes no utilizan la misma lógica para estos dos eventos).

Para ilustrar esta funcionalidad uno de los casos más sencillos de implementar, podría ser la creación de la forma triangular (**Triangle** –listado 20–).

```
using System.Windows.Shapes;

namespace Ejemplo1
{
    public class Triangle : Path
    {
        public Triangle()
        {
            CreatePathData(0, 0);
        }

        private double lastWidth = 0;
        private double lastHeight = 0;
        private PathFigure figure;

        private void AddPoint(double x, double y)
        {
            LineSegment segment = new LineSegment();
            segment.Point = new Point(x + 0.5 * StrokeThickness,
                                     y + 0.5 * StrokeThickness);
            figure.Segments.Add(segment);
        }

        private void CreatePathData(double width, double height)
        {
            // Para ajustar la salida hay que reducir el tamaño del stroke
            height -= this.StrokeThickness;
            width -= this.StrokeThickness;
            // Para evitar un bucle en la fase de rendering
            if (lastWidth == width && lastHeight == height) return;
            lastWidth = width;
            lastHeight = height;

            var geometry = new PathGeometry();
            figure = new PathFigure();
```

```

figure.StartPoint = new Point(0 + 0.5 * StrokeThickness, height + 0.5 * StrokeThickness);
    AddPoint(width, height);
    AddPoint(width / 2, 0);
    AddPoint(0, height);
    figure.IsClosed = true;
    geometry.Figures.Add(figure);
    this.Data = geometry;
}

protected override Size MeasureOverride(Size availableSize)
{
    return availableSize;
}

protected override Size ArrangeOverride(Size finalSize)
{
    CreatePathData(finalSize.Width, finalSize.Height);
    return finalSize;
}
}
}

```

Listado 20

Una vez declarada y compilada la clase **Triangle**, aparecerá en nuestra Caja de herramientas (figura 18-a) una referencia a este nuevo control, y podremos utilizarlo como si se tratase de cualquier otra forma (objeto **Shape**) predeterminado, seleccionándolo y dibujándolo sobre nuestra interfaz de usuario (figura 18-b).

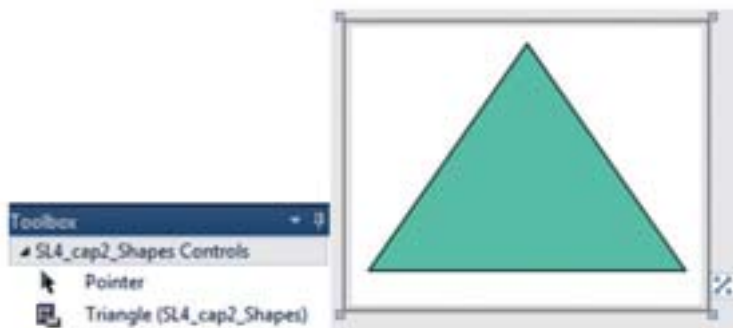


figura 18a-18b

Nuestro objeto Triangle en la caja de herramientas y en el editor visual, una vez modificado con un color de fondo

El código XAML generado por el IDE después de la operación de dibujo y un ajuste del color de fondo del triángulo (atributo **Fill**) es el siguiente (listado 21):

```
<my:Triangle Height="148" HorizontalAlignment="Left" Margin="12,12,0,0" x:Name="triangle1"
Stroke="Black" StrokeThickness="1" VerticalAlignment="Top" Width="206" Fill="#FF83C4B0" />
```

Listado 21

Como vemos, una vez definida la forma en la clase, todas sus propiedades heredadas de **Path** están disponibles desde el editor de XAML, y resulta tan simple como manejar cualquier otro elemento.

Aparte de la construcción de otras formas de dibujo, uno de los usos más interesantes de esta opción sería la creación de logos personalizados, que —una vez definidos como objeto **Shape**— puedan aprovechar todas las posibilidades que ofrecen esta clase de objetos y las herramientas que lo soportan, para crear variantes de esos logos, elementos dinámicos asociados con ellos, y un sinfín de posibilidades en presentación corporativa y publicidad.

Este es un caso claro de cómo un cambio aparentemente nimio en una propiedad del **CoreCLR** puede suponer un gran avance desde nuestro punto de vista.

Dependency Properties y Eventos

Si observamos el diagrama jerárquico de la figura 7, observamos que al principio de toda la jerarquía se encuentra un objeto, **DependencyObject**, del que hereda el resto. Su papel en los entornos visuales basados en XAML es fundamental y vamos a comentar su origen.

Sabemos por otros entornos de desarrollo que un evento es un mecanismo de comunicación entre dos métodos de una clase o de distintas clases, mediante el cual nuestro código responde a un estímulo predefinido. Muchas veces, ese estímulo viene de la interfaz de usuario, otras del propio sistema.

En .NET, esa arquitectura se implementó basada en el concepto de delegado: un intermediario administrado del proceso de comunicación que ha resuelto no pocos problemas respecto a la estabilidad de la plataforma. En Windows Presentation Foundation, el concepto de evento varió, en tanto que variaba el modelo, y Silverlight, como

un subconjunto de aquél, heredó esas características. En ambos sistemas, la estructura jerárquica que impone la sintaxis XML fue determinante, y la separación entre el **Silverlight Rendering Engine** (motor de interpretación visual de Silverlight) y el motor de ejecución .NET, propició la aparición de mecanismos de vinculación entre ambos que permitieran que técnicas como el Late Binding y las notificaciones automáticas de cambios, —entre otras— operen como si tal separación no existiera.

Propiedades de Dependencia y propiedades del CLR

Con ese propósito, Silverlight dispone de un conjunto de servicios que reciben el nombre de Sistema de Propiedades de Silverlight (*Silverlight Property System*), que permiten extender la funcionalidad de una propiedad del CLR así como su acceso a ella. Se establece así una correspondencia entre elementos propios del *CLR* y otros de XAML (esta misma característica es uno de los pilares fundamentales de Windows Presentation Foundation), de forma que se puede trabajar desde el lenguaje XAML y hacer referencia a estos elementos, como si fueran nativos de este lenguaje.

A estas propiedades, se les conoce como *Dependency Properties* (DP) y su función principal es permitir calcular el valor de una propiedad en función del valor de otras entradas, pudiéndose incluso establecer retro-llamadas (*callbacks*) para comprobar los cambios producidos.

En Silverlight 4, las propiedades se exponen, típicamente, como propiedades clásicas del CLR. Podríamos estar utilizando muchas de ellas sin saber que se trata en realidad de DP. De esa forma, se suministra un mecanismo de evaluación del valor de una propiedad basándose en otros datos, como pueden ser propiedades del sistema, datos obtenidos mediante técnicas de Data Binding, recursos (como los estilos, por ejemplo), o valores conocidos, como las relaciones padre-hijo con otros elementos.

Se reconocen por la presencia de un campo cuyo nombre está basado en la propiedad a la que "mapean", seguido de la palabra reservada **Property**, y hay muchos escenarios en Silverlight donde su presencia es fundamental.

En el capítulo dedicado a los controles veremos esta técnica como parte de la definición de un control de usuario. La sintaxis de declaración una **DependencyProperty** es la siguiente (listado 22).

La clase estática **DependencyProperty** registra la propiedad **Text**, anotando su tipo, su clase, la cadena que se utilizará para nombrarla y —eventualmente— una valor para el parámetro **PropertyMetadata**, que aquí pasamos como **null**, pero que puede recibir el nombre de una función por la que pasará el flujo de ejecución cuando el valor de la propiedad se modifique.

```
private static readonly DependencyProperty TextProperty =
    DependencyProperty.Register("Text", typeof(string), typeof(CajaTextoDNI), null);

public string Text
{
    get { return (string)GetValue(TextProperty); }
    set { SetValue(TextProperty, value); }
}
```

Listado 22

Una vez hecho esto, nuestra clase podrá acceder a esa propiedad desde código XAML como un atributo más.

Conclusión

Solo hemos comenzado a entrever la potencia del lenguaje XAML en la construcción de elementos de interfaz de usuario. Los fundamentos, sobre lo que todo lo demás se basa, son estos. Ahora continuaremos profundizando en ellos, y utilizaremos indistintamente Visual Studio o Expresión Blend, según la tarea a realizar.

Contenedores y disposición visual

En el capítulo anterior hemos tratado de las formas básicas del lenguaje XAML y de las características principales de su sintaxis. No obstante, salvo en los casos allí citados, lo usual es reutilizar lo existente. Esto, que en Windows Forms resulta casi obligado, hemos visto que en Silverlight es opcional, ya que contamos con la posibilidad de crear cualquier elemento visual e incorporarlo a la interfaz de usuario casi con la misma facilidad que los controles de Windows Forms.

Pero la costumbre y las buenas prácticas han impuesto la utilización de elementos visuales que forman parte de las interfaces de usuario gráficas (no sólo de Windows) y se repiten en todas las plataformas: botones, cuadros combinados, cajas de texto, etc.



figura 1 Logo del Silverlight Toolkit

Lógicamente, Silverlight dispone de esos elementos, que aparecen relacionados en el Cuadro de herramientas de los IDE, y que han sido contruidos por el equipo de desarrollo de Silverlight desde cero. No obstante, el conjunto de controles que se instala con el SDK predeterminado contiene solo una parte de los que están disponibles. La razón es que como —técnicamente— son extensiones del lenguaje, la actualización y publicación de los controles nuevos la realiza el sitio público Codeplex (<http://codeplex.com>) y se actualiza de forma periódica, pero separada del SDK oficial.

Cuando estos controles han alcanzado los niveles de madurez suficientes, pasan a engrosar el propio SDK. El proyecto que agrupa los controles disponibles actualmente se encuentra en la página <http://silverlight.codeplex.com> y la última versión —en el momento de escribir estas líneas— es la correspondiente a Abril de 2010, si bien, recientemente, apareció un interesante añadido para el desarrollo con la plataforma **Windows Phone 7**, el **Windows Phone Toolkit**, que se encuentra en estado estable e incluye, además de las librerías, ejemplos y abundante código fuente. Trataremos de él en el capítulo 8.

Además, este kit de herramientas no solo incluye controles, sino temas visuales (*themes*), herramientas para depuración y código fuente que merece la pena revisar.

Por tanto, si no nos hemos instalado previamente este paquete de controles, no veremos en el cuadro de herramientas más que los controles básicos. La recomendación oficial indica que no debieran utilizarse para producción más que los controles que han alcanzado la calificación de *stable* o *mature* (estable o maduro), estando el resto disponible como una anticipación de lo venidero.

Así pues, conviene tener presente que todos los controles que utilicemos en Silverlight no son más que grupos de ficheros compilados y almacenados en DLL, de los que podemos obtener el código fuente, o a los que podemos aplicar nuestros propios estilos, estados visuales (luego profundizaremos algo más sobre esto) e incluso sobrescribir funcionalmente para que se adapten mejor a nuestros proyectos.

Controles de terceros

Sin hacer publicidad aquí de otros productos, resulta interesante recordar que para necesidades profesionales, o por simple ahorro de tiempo, también disponemos de una abundantísima colección de controles (tanto de pago, como gratuitos) disponibles con solo hacer referencia a ellos a través de un espacio de nombres y una referencia en un proyecto. El propio sitio de Codeplex alberga cientos de proyectos de diversa envergadura, calidad, y grado de madurez, que resulta recomendable visitar cuando necesitemos de alguna funcionalidad concreta en forma de control que no esté presente en las colecciones anteriores.

Además, el sitio oficial de Silverlight también almacena y cataloga aportaciones de terceros, igual que referencias a soluciones accesibles y publicadas, como forma de que los usuarios que se aventuren en la plataforma, y puedan tener una idea de lo que otras empresas han puesto en circulación.

Y, por supuesto, las empresas más conocidas dedicadas a la construcción de controles disponen de conjuntos funcionales (*suites*) para Silverlight y WPF que el lector descubrirá con una simple búsqueda en la red por términos como "controles +Silverlight".

La arquitectura visual de una aplicación

Una aplicación profesional que quiera dar un aspecto novedoso, presentar un mensaje claro y sin ambigüedad y proporcionar una experiencia de usuario clara y fácil de usar, debe de tener un diseño acorde con esos objetivos. Y eso pasa por la creación de una "arquitectura visual", que son los cimientos sobre los que poner después los distintos elementos que integrarán la solución.

Debemos de tener en cuenta que la aplicación y sus contenidos podrá sufrir cambios de tamaño (tanto por el navegador que lo aloje, como si está ejecutándose fuera de él). E, incluso en el caso de Windows Phone 7, cambios de orientación. Además, la aplicación puede interpretarse visualmente de forma distinta en Windows o en Mac.

El armazón que permitirá ubicar después los distintos elementos XAML creando una interfaz coherente, deberá de estar formado por los elementos contenedores.

Disposición visual de los elementos (Layout)

Ya vimos en el capítulo anterior una primera diferenciación entre los elementos XAML de Silverlight: los que pueden hacer las veces de contenedores y los que no. A su vez, en los primeros, podemos dividirlos en dos partes: los que sólo admiten un elemento contenido (aunque este, a su vez, pueda ser un contenedor) y los que no. Afortunadamente para nosotros, el sistema de Intellisense de Visual Studio 2008 y el analizador sintáctico, reconocerán estas circunstancias, indicando lo que sucede en cada situación.

Los contenedores: una clase especial de controles

Aunque hereden de la clase **Control**, no conviene olvidar que —estrictamente— los elementos contenedores no son controles típicos, lo que significa varias cosas:

- No tienen un aspecto predeterminado (si no los dotamos de propiedades que distingan sus bordes o su color de fondo, aparecen como transparentes).
- No son diseñados mediante plantillas (esto explica lo anterior). Son elementos básicos del *core* de Silverlight, pensados para construir estructuras arquitectónicas visuales de tipo jerárquico (contenedor-contenido), y no para ofrecer aspecto visual alguno.
- Todos los contenedores principales (no los añadidos posteriormente como parte del *Silverlight Toolkit*), heredan de **Panel** (**System.Windows.Controls.Panel**).

Elementos contenedores

A continuación mostramos la jerarquía de elementos que heredan de **Panel**, tal y como la muestra el Explorador de objetos de Visual Studio. Los 4 contenedores disponibles son **Grid**, **Canvas**, **StackPanel** y **VirtualizingPanel**:

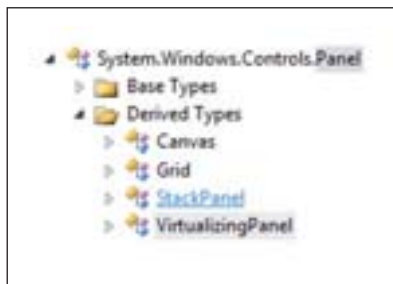


figura 2 Jerarquía de objetos derivados de Panel

La gran diferencia entre los dos primeros es la forma en que ubican sus contenidos. Mientras un **Grid** (rejilla) los sitúa dentro de filas y columnas, el **Canvas** (lienzo) está pensado para la ubicación absoluta, relativa a la posición inicial (coordenadas 0,0). Podríamos decir que es el equivalente al atributo **Position:absolute** de CSS, pero dentro de XAML. Un objeto **Canvas** no divide su superficie, como sucede con **Grid** (donde, al menos, hay una fila —fila 0— y una columna —columna 0—). Esto tiene implicaciones importantes en el comportamiento visual:

- Mientras los elementos de un **Grid** tienden a ocupar posiciones relativas respecto al contenedor —incluso al crecer este—, los de un **Canvas** siempre mantienen su posición respecto al origen.
- Si queremos reproducir una situación en la que los elementos se solapan (las capas típicas de HTML), un objeto **Grid** no nos servirá para ese propósito, mientras un objeto **Canvas** podrá colocar sus elementos a cualquier distancia del origen, y podremos establecer el orden visual en el eje de las **z** utilizando la propiedad **ZIndex**, de forma muy similar a como hacemos en HTML/CSS.

Configuración de contenedores Grid

Por tanto, solemos utilizar rejillas por su similitud con las tablas HTML y la facilidad que nos ofrecen para ubicar elementos en pantalla, manteniendo el conjunto

bien separado gracias a su división en filas y columnas. Es el más flexible de todos los contenedores, y define 3 estrategias de asignación de tamaño para sus filas y columnas:

- **Tamaño absoluto:** donde a cada definición se le indica el tamaño exacto en píxeles (Ej. `<RowDefinition Height="100" />`)
- **Tamaño automático:** el contenedor asigna a cada fila/columna el tamaño que necesita y nada más. (Ej. `<RowDefinition Height="Auto" />`)
- **Tamaño proporcional:** el contenedor asigna a cada fila/columna el tamaño disponible dependiendo del total de espacio utilizable. (Ej. `<RowDefinition Height="*" />`). Este modo tiene otras posibilidades pudiéndose asignar porcentajes del total. Por ejemplo, la siguiente definición asigna a la primera fila, la mitad del espacio que asigna a la segunda pero el doble del que asigna a la tercera (listado 1):

```
<RowDefinition Height="*" />
<RowDefinition Height="2*" />
<RowDefinition Height="0.5*" />

... o para las columnas...

<ColumnDefinition Width="0.5*" />
<ColumnDefinition Width="1.5*" />
```

Listado 1

Además, estos modos siempre pueden mezclarse para conseguir la combinación requerida para un escenario concreto, y, de forma similar a las tablas de HTML, un elemento puede ocupar más de una fila o columna. Para ello se utilizan los atributos **RowSpan** y **ColumnSpan**.

Por último cabe mencionar la capacidad de los **Grids** de funcionar de forma similar a los controles **Splitter**, capaces de dividir una superficie en dos partes, que el usuario puede cambiar de tamaño de forma dinámica. Para ello, se dispone del objeto **GridSplitter**, pensado con esta idea. Su funcionamiento es algo más complejo, y lo mejor para manejarlo es utilizar el diseñador de estados **Visual State Manager**, que veremos en otro capítulo.

Diferencias en la presentación con objetos Canvas y StackPanel

A pesar de que el conjunto de clases que hereda de **Panel** es más amplio, como vimos en la imagen, no todos ellos se utilizan como contenedores principales del control Silverlight. Realmente, solo se utilizan dos: **Grid** y **Canvas**, ya que **StackPanel** se adapta mejor como sub-contenedor por su capacidad de agrupar los contenidos en una sola línea que puede ser orientada vertical u horizontalmente, y **VirtualizingPanel** es una clase abstracta que comentaremos a continuación.

La documentación oficial nos suministra un pequeño ejemplo para ayudarnos a establecer la diferencia operativa de estos dos contenedores. Ese es el caso de una forma rectangular que quisiéramos ubicar en una posición dada dentro de dos objetos **Canvas** anidados (listado 2):

```
<Canvas Width="300" Height="300" Background="White">
  <Canvas Width="250" Height="250" Canvas.Left="30" Canvas.Top="30" Background="blue">
    <Rectangle Canvas.Left="30" Canvas.Top="30" Fill="red" Width="200" Height="200" />
  </Canvas>
</Canvas>
```

Listado 2

El código XAML anterior produce el siguiente resultado (figura 3):

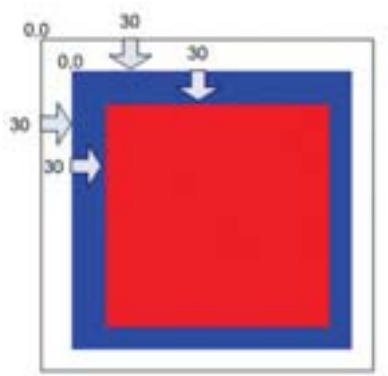


figura 3

Objeto Canvas anidado dentro de otro que contiene un rectángulo (Gráfico de MSDN)

En situaciones de diseños complejos, lo mejor es jugar con ambos mundos, ya que pueden convivir perfectamente (Un **Grid** puede contener objetos **Canvas** en cual-

quiera de sus celdas o en todas ellas, y al revés). A modo de ilustración, el ejemplo siguiente muestra un **Grid** cuadrado (mismo número de filas y columnas) en el que cada celda tiene distintos tipos de contenido: Otro **Grid**, un control **TextBlock**, un **Canvas**, y un control **Image** con su configuración cambiada para que la imagen ocupe toda su superficie. En el caso del **Canvas** (Fila 1, Columna 0) su contenido (el rectángulo) se ubica de forma relativa al contenedor directo (el propio **Canvas**), y no al **Grid** que hace de contenedor general.

El código correspondiente es este (listado 3):

```
<UserControl x:Class="Ejemplo2.Page"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="400" Height="300"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d" Background="#FFDBC8C8">
    <Grid Background="#FFDBC8C8">
        <Grid.RowDefinitions>
            <RowDefinition Height="0.48*"/>
            <RowDefinition Height="0.52*"/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="0.5*"/>
            <ColumnDefinition Width="0.5*"/>
        </Grid.ColumnDefinitions>

        <!-- Fila 0, Columna 0 -->
        <Grid Margin="24,24,24,24" x:Name="Grid_Interior" >
            <Grid.Background>
                <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
                    <GradientStop Color="#FF000000"/>
                    <GradientStop Color="#FFE57676" Offset="1"/>
                </LinearGradientBrush>
            </Grid.Background>
        </Grid>

        <!-- Fila 0, Columna 1 -->
        <TextBlock Margin="0,0,0,0" Text="Hola Mundo" TextWrapping="Wrap"
            Grid.Column="1" Foreground="#FF3512AE" Width="Auto" Height="33"
            HorizontalAlignment="Center"/>

        <!-- Fila 1, Columna 0 -->
        <Canvas Canvas.Left="30" Background="#FF0FA22B"
            Canvas.Top="30" HorizontalAlignment="Stretch" Margin="32,16,40,20"
            Grid.Row="1" Grid.Column="0" >
            <Rectangle Canvas.Left="32" Canvas.Top="30" Fill="#FFFFFF500"
```

```

        Width="74" Height="74" />
    </Canvas>

    <!-- Fila 1, Columna 1 -->
    <Image Margin="32,16,32,20" Grid.Column="1" Grid.Row="1" Source="05.jpg"
        Stretch="Fill"/>
    </Grid>
</UserControl>

```

Listado 3

Como ya indicamos, en caso de no especificar valores para **Grid.Row** y **Grid.Column**, el sistema asume que su valor es 0. Hay un montón de posibilidades, desde las más intuitivas a las más complejas. La salida del código anterior se corresponde con la figura 4.



figura 4

Salida del código anterior mostrando un *Grid* con cuatro celdas y un objeto distinto en cada una de ellas

El **Grid** en la posición (0,0), no tiene ningún contenido. En lugar de eso, hemos escogido dibujar su fondo con una brocha de tipo gradiente, que veremos con más detalle después. En el **Canvas** con el rectángulo incluido se ha disminuido de tamaño y cambiado el color, el **TextBlock** se ha configurado para que esté completamente centrado en la celda que lo contiene y la imagen se ha alargado modificando el atributo **Fill** (modo de relleno de la imagen), de forma que ocupe todo el espacio disponible en el control **Image** que lo carga.

Márgenes y alineación

El control de la ubicación de los elementos que no se sitúan mediante posicionamiento absoluto, se realiza mediante las propiedades **Margin**, **HorizontalAlignment** y **VerticalAlignment**, principalmente. **Margin** permite establecer un valor de margen para los 4 valores de separación: (en este orden) izquierda, superior, derecha e inferior, o bien un valor distinto para cada uno de ellos. Normalmente, la combinación adecuada de ambos ofrece unas excelentes posibilidades de posicionamiento sin tener que recurrir al posicionamiento absoluto (el problema de este último es que, a veces, tenemos que crear interfaces que cambien su tamaño dinámicamente al cambiar la superficie de la página que los contiene).

Respecto al contenido de los textos dentro de los controles que los pueden albergar, además de los anteriores, disponemos de la propiedad **Padding**, que permite establecer la distancia del texto contenido a los bordes de su contenedor con una sintaxis similar a la utilizada para los márgenes (su valor por defecto es 0).



figura 5

Gráfico demostrando las diferencias entre los atributos Margin y Padding

Independientemente de las anteriores, existen multitud de situaciones en las que no deseamos un posicionamiento exacto, sino relativo al contenedor o al contenido. Para estos casos, podremos utilizar las propiedades de alineación (**HorizontalAlignment** y/o **VerticalAlignment**), disponibles para cualquier elemento que herede de **UIElement**, que admiten los siguientes valores de configuración:

- **Left:** ubicación a la izquierda de su elemento contenedor.
- **Center:** ubicación centrada en su elemento contenedor.
- **Right:** ubicación a la derecha de su elemento contenedor.
- **Stretch:** el elemento se distorsiona hasta ocupar todo el espacio disponible en su contenedor.

StackPanel

El contenedor **StackPanel** se utiliza siempre que deseemos organizar un subconjunto visual de forma vertical u horizontal. Normalmente, ahí acaba su función, aunque no hay inconveniente para que sea utilizado como contenedor total de una página si la estructura lo recomienda así. Su propiedad principal (**Orientation**), establece la manera en que dispone sus contenidos y, simplemente, deberemos ubicarlos en el mismo orden que queremos que aparezcan en caso de que el orden sea importante.

VirtualizingPanel

Se trata de una clase abstracta que se planteó para ofrecer una solución en algunos casos en que los elementos contenedores deben de incluir una gran cantidad de contenidos (por ejemplo por que los construyan dinámicamente a partir de una fuente de datos de cualquier tipo), algo que en muchas ocasiones plantea problemas de rendimiento.

Mediante la funcionalidad ofrecida por la clase **VirtualizingPanel**, un elemento contenedor como **StackPanel** puede utilizar una clase derivada como **VirtualizingStackPanel**, para calcular los elementos visibles en pantalla según su definición en un momento dado. Esta clase funciona junto a un **ItemContainerGenerator**, disponible para los elementos que posean una colección **ItemsControl** (como **ListBox** o **ListView**), de forma que solo carga el número de elementos que pueden ser visibles, con la consiguiente ganancia de rendimiento.

Por ejemplo, el código siguiente, es una forma de implementación típica, utilizando la clase derivada **VirtualizingStackPanel** desde un **ListBox** interior a él. El acceso a la propiedad de virtualización es mediante una propiedad adjunta del contenedor (listado 4):

```
<StackPanel DataContext="{Binding Source={StaticResource Ligas}}">
  <ListBox VirtualizingStackPanel.IsVirtualizing="True"
    ItemsSource="{Binding XPath=Equipos}" />
</StackPanel>
```

Listado 4

Como vemos en este código, el enlace a datos se establece mediante la propiedad **DataContext** del elemento **StackPanel** contenedor, leyendo la información de algún recurso (**StaticResource**) llamado *Ligas* (más adelante veremos el enlace a datos). Pero el elemento que carga la información a partir de ese recurso para mostrarla en pantalla, es el **ListBox** contenido en él, que recupera los datos de los *Equipos* disponibles en las *Ligas*.

Lo interesante aquí es la sintaxis utilizada (la propiedad adjunta **VirtualizingStackPanel.IsVirtualizing**), junto al hecho de que los elementos cargados de esta forma se calculen de tal manera que solo se lean aquellos que caben en pantalla, dependiendo de los valores seleccionados (o predeterminados) para la superficie útil del **ListBox** tal y como se haya establecido por diseño, y los tipos de letra que Silverlight asume como predeterminados (en caso de no especificar ninguno).

Las dos grandes diferencias operativas son:

- Solo creamos los controles que van a tener visibilidad una vez presentada la interfaz de usuario
- Esos controles no son constantemente destruidos y vueltos a construir dependiendo del desplazamiento del usuario, sino que son "reciclados" cambiando simplemente la visibilidad de los elementos.

Los otros contenedores

El conjunto de elementos incluidos en el Silverlight Toolkit, incluye un buen número de controles, entre los cuales hay varios que caen en la categoría de contenedores (simplemente, porque su función principal es la de albergar otros elementos), si bien les hay que no están contruidos siguiendo las jerarquías de clases en que se basan los que hemos estudiado aquí (no heredan de **Panel**). Este es el caso de los controles **DockPanel**, **WrapPanel** y **ViewBox**, aunque éste último no es reconocido como un control contenedor "puro" por la documentación oficial, ya que su única función es la de modificar la escala de sus elementos contenidos.

Naturalmente, todos los controles son —a fin de cuentas, eso— contenedores de información, pero la diferencia es que aquí separamos los que están pensados como elementos de mera arquitectura de aquellos que se han definido para contener a otros.

DockPanel

Su función es la de acoplar uno o más controles contenidos a uno de sus lados: superior, inferior, izquierdo o derecho. Se configura mediante la propiedad adjunta

DockPanel.Dock, y sus valores son: **Top**, **Bottom**, **Left** y **Right**. A diferencia de otros contenedores similares (como sucede en Windows Forms) no se permite el acoplamiento simultáneo a todos ellos. El ejemplo siguiente (listado 5) define un contenedor **DockPanel** y dos elementos **TextBox**, uno de ellos acoplado y el otro no:

```
<my:DockPanel Width="300" Height="150" Background="#FFDEEAED">
  <TextBox Text="TextBox Acoplado a la parte superior" Width="150"
    Background="#FFD08888" TextWrapping="Wrap"
    my:DockPanel.Dock="Top" />
  <TextBox Text="Este TextBox está contenido, pero no acoplado"
    Width="150" Height="60" TextWrapping="Wrap" />
</my:DockPanel>
```

Listado 5

Lo que produce una salida como esta en el editor visual de Visual Studio 2010:

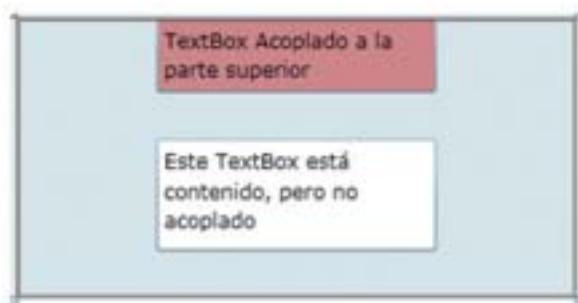


figura 6

El contenedor DockPanel con un elemento acoplado a la parte superior

nota

Si el usuario utiliza Visual Studio 2010 o Expression Blend, habrá observado que el Editor Visual actualiza la posición según cambiamos el código XAML, de la misma forma que éste es cambiado al manipular los elementos de la superficie de diseño.

WrapPanel

Como indica su nombre en inglés, una vez establecidos sus límites (longitud y altura) va ubicando todos sus elementos contenidos de izquierda a derecha según el orden de inclusión hasta que uno de ellos llega a completar todo su espacio disponible, momento en que continúa en la siguiente fila (aunque no existen tales filas, sino que las establece el elemento más alto), y así sucesivamente hasta el final.

El siguiente código (listado 6) implementa un contenedor con 3 cajas de texto y 2 botones. Una vez establecidas las dimensiones del objeto **WrapPanel**, los elementos se van ubicando "según caen", sin más consideraciones:

```
<my:WrapPanel Width="344" Height="175" Background="#FFDEEAED">
  <TextBox Text="Primera Caja de Texto" Width="150"
    Background="#FFD08888" TextWrapping="Wrap"
    my:DockPanel.Dock="Top" />
  <TextBox Text="Segunda Caja de Texto"
    Width="150" Height="60" TextWrapping="Wrap" />
  <TextBox Text="Tercera Caja de Texto"
    Width="250" Height="60" TextWrapping="Wrap"
    Background="#FFFE66FF" />
  <Button Content="Botón 1" Height="23" Name="button1" Width="75" />
  <Button Content="Botón 2" Height="23" Name="button2" Width="75" />
</my:WrapPanel>
```

Listado 6

Y su aspecto visual en el editor, sería el de la figura 7:



figura 7 El contenedor *WrapPanel*

ViewBox

Estrictamente, no se trata de un elemento contenedor para **disposición** visual, sino para **escalado** visual. Su única función es la de escalar sus contenidos en función de sus propiedades de anchura y altura. Y dispone de un funcionamiento automático que permite, según el valor de su propiedad **Stretch**, utilizar un mecanismo de relleno de contenidos similar al que ya hemos visto para esta propiedad en el capítulo anterior.

El gráfico adjunto muestra la diferencia de salida visual de una imagen de tamaño configurado a 300x200 y —en la columna de al lado— la misma imagen dentro de un elemento **ViewBox** de tamaño 200x100 y propiedad **Stretch** asignada a **Fill**.



figura 8

Un elemento *ViewBox* controla el tamaño de la segunda imagen

Lo que corresponde al siguiente al código fuente del listado 7.

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>

  <Image Source="/Ejemplo1;component/W7_Logo.png" Width="300" Height="200"
    Grid.Column="0" />
  <Viewbox Width="200" Height="100" Stretch="Fill" Grid.Column="1">
    <Image Source="/Ejemplo1;component/W7_Logo.png" />
  </Viewbox>
</Grid>
```

Aunque su uso con gráficos es muy habitual, también se ha estado utilizando como alternativa a las soluciones de accesibilidad, dada la capacidad de hacer crecer/decrecer cualquier tipo de contenido XAML, incluyendo controles estándar, **Shapes**, imágenes, vídeo, etc.

Respecto a la calidad obtenida, dependerá del elemento a escalar, como es lógico. Los mapas de bits perderán calidad al agrandarse o deformarse, mientras que los elementos vectoriales la mantendrán.

Conclusión

Los elementos contenedores permiten la construcción del escenario donde el resto de controles se ubicarán, pero también permiten la creación de capas y solapamientos, ubicación absoluta o relativa de elementos, escalado, virtualización y un largo etcétera. Es por ellos por los que todo diseño de IU debiera comenzar.

Expression Blend 4 para desarrolladores

Ya vimos en el capítulo 2 cómo —en XAML— podemos distinguir dos tipos básicos de elementos desde el punto de vista de su forma y funcionamiento: los creados por el usuario como parte de sesiones de dibujo asistido y los controles, que son estructuras ya construidas.

También apuntábamos que para poder crear de forma visual elementos del primer tipo, la solución no es Visual Studio, sino **Expression Blend 4.0**, que sí dispone de capacidades especiales como herramienta de diseño. Por lo demás, el proceso de construcción de una aplicación Silverlight en Blend es muy similar, si bien hay ciertas plantillas que no están disponibles (las de aplicaciones LOB, y las de Navegación), y como contrapartida disponemos de las plantillas para aplicaciones *Sketch-Flow*, una herramienta para crear prototipos de soluciones ricas, que generan ejecutables de WPF con excelentes capacidades en el marco de la colaboración del equipo de desarrollo.

El papel de Blend 4.0 y la suite Microsoft Expression 4.0 en el desarrollo con Silverlight



Consciente de la necesidad de herramientas para diseño y de la creciente importancia del diseñador en las aplicaciones de hoy, Microsoft comenzó a distribuir —casi de forma paralela a la aparición de las primeras versiones de Silverlight— la que ahora es una *suite* (Microsoft Expression), para el diseñador (o para el programador que necesita incluir elementos de diseño en sus aplicaciones). De forma que Expression Blend ha ido evolucionando paralelamente al resto de herramientas de la *suite*, que cuenta en la actualidad con 5 miembros:

- **Expression Blend** para diseño interactivo para XAML (WPF / Silverlight).
- **Expression Design**, para diseño gráfico.
- **Expression Web**, para diseño de sitios Web.
- **Expression Web SuperPreview** (complemento del anterior) para testeo visual de sitios simulando diversas plataformas y navegadores y resoluciones de pantalla.
- **Expression Encoder**, para edición de vídeo (y audio).

La idea es que, técnicamente hablando, se puede decir que Expression aporta todo lo necesario para la construcción de aplicaciones RIA con gran carga de elementos de diseño y multimedia y Expression Web es la contrapartida para HTML/ASP.NET del sitio, mientras Blend resulta especialmente indicado para el diseño en Silverlight.

Primera aplicación en Blend 4.0

Tras abrir Blend, y seleccionar "New Project", lo primero que vemos, es que existen 5 tipos de proyectos Silverlight, y que tenemos la opción de trabajar con las versiones 3.0 y 4.0 de la plataforma. (figura 1).

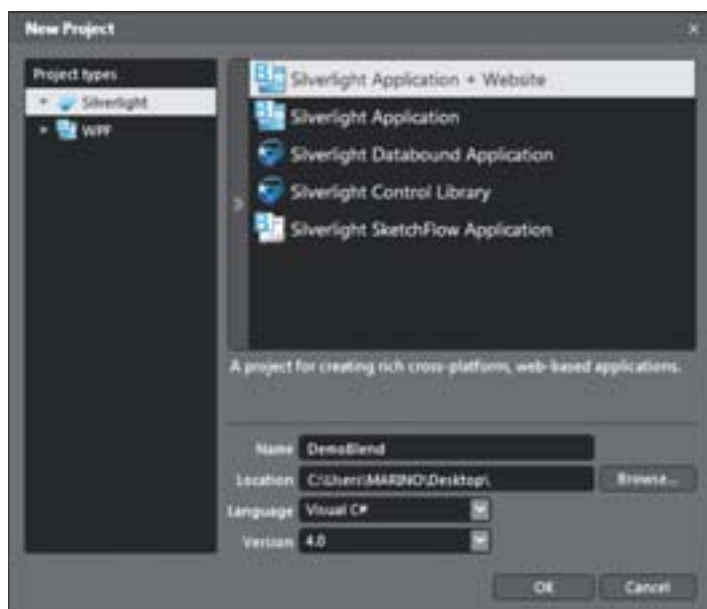


figura 1 Oferta inicial de proyectos en Blend 4 para soluciones Silverlight

Si seleccionamos una aplicación Silverlight + Website, nos creará una solución con dos proyectos, al estilo de Visual Studio, con algunas variantes en la parte Web donde se alojará el complemento (en una página llamada **Default.html**).

En la figura 2, vemos la ventana "Projects": la misma estructura de ficheros y directorios que cabría esperar de una aplicación Silverlight 4 iniciada con Visual Studio, incluyendo las referencias a las librerías (sección "References"), el fichero **Application.manifest**, que establece comportamientos de la aplicación, y, claro está, los ficheros principales **App.xaml** y **Page.xaml**, con sus contrapartidas de clases en C#.

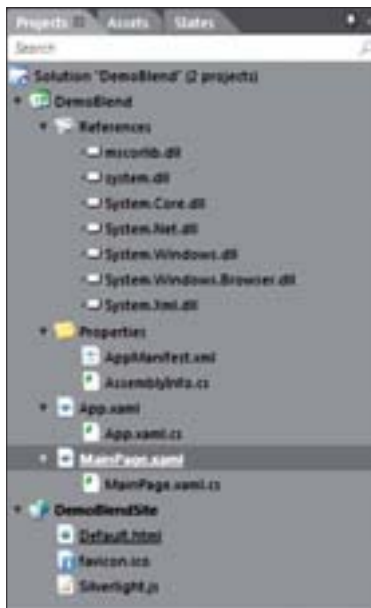


figura 2 Elementos típicos de una aplicación Silverlight creada con Blend 4 (solapa *Projects*)

Una de las mejoras de Blend 4.0 respecto a sus versiones anteriores la podemos apreciar nada más abrir uno de los ficheros C#, ya que mientras antes se nos redirigía a Visual Studio (o se abría una instancia del IDE con el proyecto), ahora tal cosa no es necesaria, disponiendo de un editor de lenguajes C#/VB.NET totalmente operativo, con Intellisense, generación dinámica de recursos de código, información complementaria, etc. La figura 3 ilustra esta capacidad, mediante la edición del fichero **MainPage.cs**, durante el proceso de añadir un manejador para el evento **Loaded** de la página.

Solo por esto, ya merecería la pena una actualización o instalación nueva de Blend 4, pero la herramienta posee muchas otras virtudes interesantes:



```
using System.Windows.Media.Animation;
using System.Windows.Shapes;

namespace DemoBlend
{
    public partial class MainPage : UserControl
    {
        public MainPage()
        {
            // Required to initialize variables
            InitializeComponent();
            this.Loaded += new System.Windows.RoutedEventHandler(MainPage_Loaded); (Press TAB to insert)
        }
    }
}
```

figura 3 El nuevo editor de código fuente de Blend 4.0

- **SketchFlow.** Como hemos anticipado, son plantillas para la creación de proyectos "borrador" con capacidades de edición colectiva, control del flujo de la aplicación, utilización de distintos tipos de controles para la interfaz de usuario, reutilización de elementos, utilización de librerías de recursos y creación de un auténtico ejecutable (en WPF), perfectamente portable para demos fuera del entorno de desarrollo.
- Importación directa de recursos de **Adobe Photoshop** y **Adobe Illustrator**, con conversión automática del formato original a formato XAML.
- Edición y depuración de animaciones y gestión de conjuntos de animaciones mediante **Visual State Manager (VSM)**. Un sencillo editor de animaciones nos permite controlar cada elemento en cada milisegundo del marco de tiempo seleccionado, y asignar esas animaciones y manejar su ejecución y problemas de concurrencia mediante el gestor de estados VSM.
- Edición visual directa de las capacidades de proyección 3D de que ahora dispone Silverlight. Las herramientas de la Ventana de propiedades nos permiten efectuar estas transformaciones en directo y observar el resultado en la Ventana de diseño.
- Creación de **estilos visuales** aplicables a cualquier control o conjunto de controles y capacidad de editar controles existentes (ya sean predeterminados, del Silverlight Toolkit, de terceras partes, o creados por nosotros).
- Posibilidad de vinculación a clases de negocio que expongan datos.
- Crear y editar aplicaciones que incluyan la tecnología **DeepZoom** para crear secuencias visuales concatenadas a partir de conjuntos de imágenes, creando experiencias visuales muy atractivas con una gran sensación de profundidad.
- Programación para **Windows Phone 7**. Poco después del lanzamiento de Blend 4, se puso a disposición de la comunidad el SDK de desarrollo para el

nuevo sistema operativo para móviles de Microsoft, que, una vez instalado, permite crear aplicaciones para este nuevo sistema utilizando Blend 4 como herramienta de desarrollo. Veremos algunos ejemplos al final de esta obra, en el apartado dedicado a esta plataforma.

Pero volvamos a nuestra aplicación inicial, como medio de comprobar los recursos disponibles en esta herramienta. Como vemos, en la parte central del IDE, el entorno nos recuerda mucho al de Visual Studio con ventanas de edición y diseño, que pueden compartirse para mostrar ambos aspectos simultáneamente (opción "Split").

El IDE de Expression Blend 4.0

Ancladas junto a la ventana de proyectos, encontramos dos ventanas adicionales: "Propiedades" y "Recursos". Las propiedades mostradas por la primera, siempre corresponden al objeto seleccionado; en la segunda, aparecen los elementos añadidos como recursos para esta aplicación, que pueden ser casi de cualquier tipo (gráficos, multimedia, ficheros de texto, HTML, DLL, etc.). En la figura 4 se aprecia el aspecto general del entorno con los elementos que indicamos.

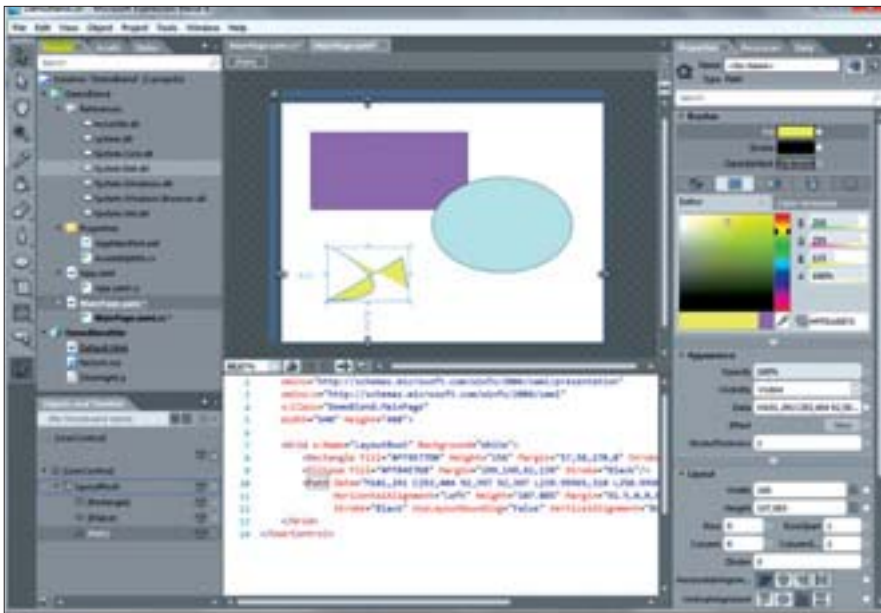


figura 4

Entorno de trabajo de Blend 4.0

A la izquierda, en una versión sintetizada, encontramos la Caja de herramientas. Desde ella tenemos acceso, primero a las herramientas de dibujo (las relacionadas con las sesiones de dibujo de usuario) y después a los controles predeterminados, asociados con iconos de grupo que despliegan menús laterales organizados por acciones comunes. El último de esos iconos de la barra vertical despliega los *Assets*, o recursos disponibles "de fábrica", por decirlo así.

A su derecha, la ventana predeterminada muestra otras dos solapas además de la de "Proyectos": "Assets" y "States". La primera es un mecanismo rápido de acceso al conjunto de recursos: controles, comportamientos (*behaviours*), estilos (*styles*), efectos (*effects*) y algunas divisiones más. La figura 5 muestra algunos de los comportamientos aplicables por defecto, que no son sino formas ya pre-escritas de reaccionar a un evento que se pueden utilizar en multitud de elementos de la IU.

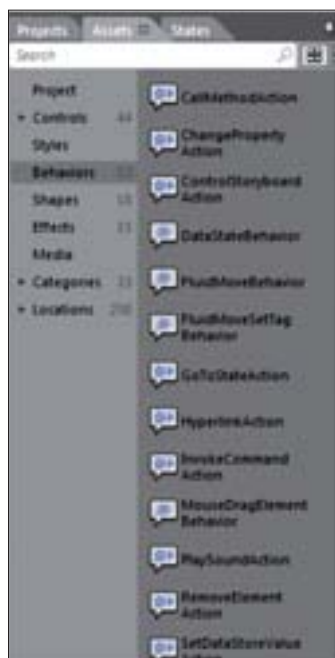


figura 5 *Behaviors* predeterminados en Blend 4.0

Respecto a la solapa "States", se refiere a estados visuales manejados por Visual State Manager y la analizaremos más adelante.

Por debajo de estas solapas encontramos la ventana denominada "Objects & Timeline" ("Objetos y Escala de Tiempo", en la versión en castellano), que nos ayuda a seleccionar los objetos y crear animaciones en el diseño. Además, los iconos  (el

ojo y el candado), nos permiten respectivamente ocultar/mostrar objetos para facilitar la edición y bloquear su posición en pantalla, evitando desplazamientos indeseados. En el capítulo 6, "Más allá de los controles", veremos otros aspectos del funcionamiento de esta ventana en el apartado dedicado a las Transformaciones y Animaciones.

Tras este vistazo inicial, observamos que —aunque el contexto visual es distinto del de Visual Studio—, la estructura del IDE sigue los patrones de sus antecesores, y no se aparta de lo que Microsoft denomina, muy técnicamente, "metáfora de disposición visual en entornos de desarrollo", o dicho llanamente, una forma bien aceptada por los desarrolladores de situar los elementos en pantalla, de manera que se tenga la mayor parte de la información fácilmente accesible.

Predeterminado vs. No predeterminado en el IDE de Blend 4.0.

Para el usuario de Blend 4, hay dos clases de elementos: los que ya tienen una definición (más *shapes*, *behaviours* y *effects*, que se incluyen en esta clase, junto a los controles) y los que no la tienen. Estos últimos son el resultado de una labor de creación del usuario, y para ello cuenta con las dos herramientas que vimos en un capítulo anterior: *Pen* (la pluma, para el dibujo asistido) y *Pencil*, que permite realizar dibujos "a mano alzada" (con todos los inconvenientes de precisión eso que supone). También vimos como su uso genera elementos de tipo **Path**.

Además, el IDE de Blend 4 ofrece geometrías y mecanismos adicionales no presentes en Visual Studio, como las disponibles en el apartado **Shapes**, que se añaden a las geometrías básicas, como **Ellipse**, **Rectangle** y **Line**.

En el manejo de controles, Blend es más flexible, ya que no existe limitación a lo que podemos hacer, incluyendo editar la plantilla predeterminada de un control modificando toda su estructura visual, como veremos más adelante.

La sub-ventana de recursos disponibles

Al final de la lista de herramientas (en el apartado "Assets") encontramos una ventana desplegable con la oferta de los controles disponibles (aquí veríamos también las librerías de terceros que pudiéramos referenciar).

Seleccionando la opción "All controls" de la ventana "Assets", accedemos a la lista completa de recursos visuales. Algunos elementos no pueden ser considerados como controles, sino, más bien, como partes integrantes de otros controles. Esta lista es la que nos muestra la figura 6.

nota

En caso de utilizar Visual Studio 2008, recuerde que no es posible arrastrar un control Silverlight directamente a la ventana de diseño. Esa limitación ha desaparecido en Visual Studio 2010, aunque, en ambos casos, el código XAML generado se interprete visualmente en la ventana de diseño.

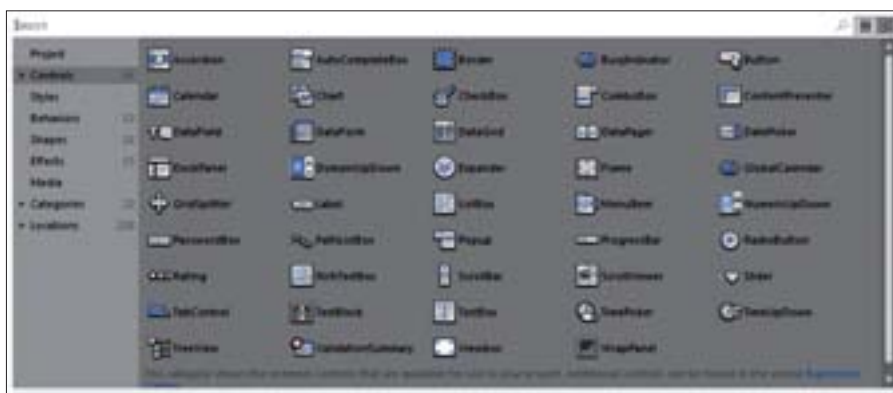


figura 6

Librería de controles (*Asset Library*) disponible en el entorno de Blend 4.0

Creación automática de elementos a partir de recursos gráficos

Otra característica útil de Blend consiste en la capacidad de asociar un elemento visual con el control predeterminado que lo maneja: este es el caso de las imágenes, asociadas al control **Image**, o de los vídeos y su correspondiente **MediaElement**. Cuando arrastramos uno de estos recursos a la Ventana de diseño, Blend crea automáticamente el elemento y le asigna las propiedades predeterminadas, más las propias del recurso (anchura, altura, etc.), de forma que podemos verlo en diseño, listo para funcionamiento. En el caso de que se trate de un vídeo, téngase presente que la pro-

nota

Un truco para conseguir elementos repetidos, consiste en usar la combinación CTL+Drag&Drop (Control + Arrastrar el elemento a otra zona visual). Al terminar el proceso se genera una copia del elemento arrastrado.

piedad de reproducción automática estará activada por defecto, por lo que, si ejecuta la aplicación, el vídeo se comenzará a reproducir nada más concluir la carga.

Categorías de recursos

Dentro de la ventana "Assets", podemos acceder a recursos disponibles por categorías, para que sea más fácil tener en cuenta lo utilizado por la aplicación y lo que está disponible. Si añadimos elementos propios en forma de vídeos o imágenes, estos aparecerán en la solapa "Media", y una lista completa de recursos, incluidos los creados por la aplicación, en el ítem "Project" (por el momento solo aparece la clase **MainPage**, y el vídeo **Wildlife.wmv**), como puede verse en la figura 10. Y cuando dejamos el cursor sobre cualquier elemento, se nos indica en una etiqueta flotante la ubicación física de la DLL o el fichero al que pertenece.

nota

Aunque no tiene que ver directamente con el desarrollo en Silverlight, la herramienta Expression Design 4, permite exportar diseños complejos en un formato utilizable por Blend. Y lo mismo sucede con algunas herramientas de terceros como Adobe Illustrator.

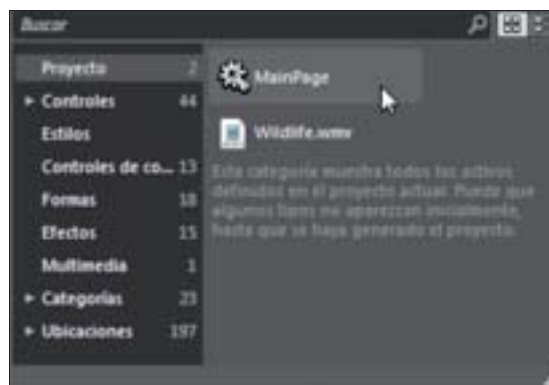


figura 10 los recursos principales de la aplicación, en la ventana *Assets*

División de los controles según Blend

La documentación de Blend, divide los controles en más categorías: los que tienen un texto de cabecera (*Headered Controls*), los que contienen colecciones de otros elementos (*Items Controls*), los que poseen una propiedad **Content** que permite mostrar cualquier contenido incrustado en ella (*Content Controls*), etc.

Una vez elegido un elemento con que trabajar, la ventana de propiedades, como en Visual Studio, muestra los valores asignados: tanto explícitamente (por haberlo dibujado en la ventana de diseño), como los predeterminados. Y esto también es aplicable también a otras opciones, como el enlace a datos mediante atributos **DataContext**.

Con estos sencillos principios ya puede el lector comenzar a hacer sus diseños visuales en Blend 4.0, y observar cómo el IDE siempre responde actualizando automáticamente los cambios hechos en cualquiera de las ventanas de edición (visual o de código XAML).

Ejemplo de inicio con un gráfico, una imagen y un vídeo

Como prueba inicial, hemos creado un par de directorios en la aplicación para albergar gráficos y vídeos. Podemos hacerlo desde el menú "Project" y seleccionar "Add New Folder" o directamente sobre el Explorador de Proyectos con el botón de derecho, tal y como muestra la figura 5.

Una vez creado el directorio, copiamos y pegamos un vídeo y algunas imágenes para hacer referencia a ellas al programar el evento **Click**. Preferimos esta opción porque, si seleccionamos la opción "Add Existing Item", lo que vamos a obtener es una

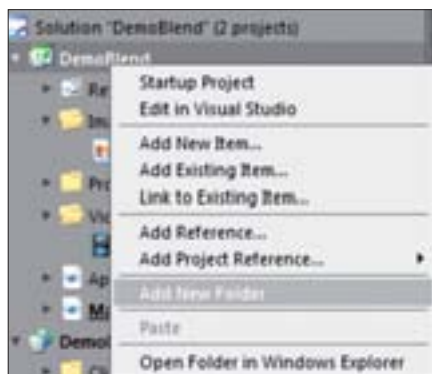


figura 7 Explorador de Proyectos mostrando el menú contextual

referencia al fichero en cuestión, pero éste seguirá estando su ubicación original, y no dispondremos de una copia local para distribuir.

nota

Podemos añadir cualquier clase de fichero al directorio creado, por lo que los formatos soportados son algo muy a tener en cuenta. No todos los formatos gráficos están soportados por Silverlight 4, siendo los formatos JPG y PNG los recomendados, si bien existen conversores por código para otros formatos. Ver el blog de Joe Stegman en el Apéndice I, para el tema de conversores de formato.

Lo más sencillo para tener una imagen en pantalla rápidamente es hacer doble clic sobre una de ellas: el diseñador creará automáticamente un objeto imagen (elemento **<Image>**) con la referencia correspondiente y unos valores de posición y tamaño acordes con las medidas del gráfico (el lector que tenga conocimiento del lenguaje XAML de WPF verá inmediatamente que se trata de un subconjunto funcional de éste).

Si queremos probar con un elemento multimedia de tipo vídeo, podemos hacer lo mismo. Una vez seleccionado uno, comprobaremos como se genera automáticamente un elemento **<MediaElement>**, que apunta al vídeo seleccionado, listo para reproducción automática nada más cargar el control. Es más, con las herramientas de la superficie de diseño tenemos la posibilidad de modificar intuitivamente el aspecto de cualquiera de los estos controles: tamaño de las imágenes, rotar el vídeo, aplicar transformaciones, etc.

Con lo hecho hasta el momento, el código producido (listado 1) tiene este aspecto (hemos añadido un objeto *Path* al final).

Y, si ejecutamos la aplicación, obtendremos esta salida visual del control Silverlight mostrado por la página en Internet Explorer 9 Beta 1 (figura 8).

Vemos, pues, que Blend es muy intuitivo de manejar, y casi siempre nos garantiza una aproximación al problema gráfico a resolver desde el propio diseño. Por otro lado, la fidelidad del resultado en tiempo de ejecución es idéntica a la ofrecida por el IDE.

Trabajo con objetos de dibujo

De la misma forma, podemos trabajar con los elementos de dibujo del panel izquierdo, usando el mismo proceso o secuencia de trabajo: dibujo ← modificación de propiedades en la Ventana de Diseño ← ajuste fino en la Ventana de Propiedades.

```
<UserControl
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:ed="http://schemas.microsoft.com/expression/2010/drawing"
  x:Class="DemoBlend.MainPage"
  Width="640" Height="480">

  <Grid x:Name="LayoutRoot" Background="White">
    <Image HorizontalAlignment="Left" Height="111" Source="Imágenes/W7_Logo.png"
      Stretch="Fill" VerticalAlignment="Top" Width="106" Margin="35,38,0,0"/>
    <MediaElement x:Name="Clip_1080_5sec_VC1_15mbps_wmv" Margin="168,164,68,64"
      Source="/Videos/Clip_1080_5sec_VC1_15mbps.wmv" Stretch="Fill"/>
    <Path Data="M55.5,234 C309,37.5 213,166.5 309,118.5 C405,70.5 432,136.5 456,135
      C480,133.5 553.5,67.500114 553.5,67.500114 L241.5,69.000114 z" Fill="#FFBCBCCA"
      Height="167.5" Margin="73,38,68,0" Stretch="Fill" Stroke="Black"
      UseLayoutRounding="False" VerticalAlignment="Top"/>
  </Grid>
</UserControl>
```

Listado I



figura 8

Resultado de la ejecución del programa anterior en Internet Explorer 9.0 Beta 1

Vamos a seguir con el mismo proyecto, y seleccionamos la opción de añadir nuevo elemento ("Add New Item"), y llamamos al nuevo control "Dibujos.xaml".

Por ejemplo, supongamos que vamos a dibujar un círculo que contenga otra figura (un rectángulo de bordes redondeados), y que éste, a su vez, contenga (visualmente, no mediante código) un control **TextBox**, que permita la edición de texto.

Esto implica el uso de tres controles XAML: un objeto **Ellipse** que, al coincidir los dos radios focales se convierte en una circunferencia, un rectángulo (**Rectangle**) con los bordes redondeados (manipulando las propiedades **RadiusX** y **RadiusY**) y un **TextBox**, al que le hemos aplicado algunas propiedades visuales (en total, dos geometrías y un control).

Por lo demás, la contrapartida XAML de este diseño es simple, como se muestra a continuación (listado 2).

Hay que tener en cuenta que no hemos creado celdas en el **Grid** contenedor. Por tanto, se asume la existencia de una sola fila y una sola columna. Tampoco hemos utilizado otros controles contenedores, por lo que la visibilidad en condiciones de solapamiento es la predeterminada en la propiedad **ZIndex**, esto es, *el primero que se dibu-*

```
<UserControl
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d" x:Class="SilverlightApplication1.Dibujos"
  d:DesignWidth="640" d:DesignHeight="480">

  <Grid x:Name="LayoutRoot" Background="White" >
    <Ellipse HorizontalAlignment="Stretch" VerticalAlignment="Stretch"
      Fill="#FF652DC2" Stroke="#FF000000"
      RenderTransformOrigin="3.20000004768372,2.48000001907349"
      Margin="112,56,144,56"/>
    <Rectangle HorizontalAlignment="Stretch" Margin="176,168,208,168"
      VerticalAlignment="Stretch" Fill="#FFD0C590" Stroke="#FF000000"
      RadiusX="40" RadiusY="40"/>
    <TextBox Margin="207,203,235,208" Text="Caja dentro de un rectángulo dentro de una elipse"
      TextWrapping="Wrap" FontSize="14" FontWeight="Bold"/>
  </Grid>
</UserControl>
```

Listado 2

ja va más al fondo, y los demás se sitúan por encima, y así sucesivamente. Esto puede cambiarse en cualquier momento (a valores positivos mayores de *ZIndex*, más acercamiento al espectador en el eje de las *Z*).

Cambio del punto de entrada de la aplicación

Si el lector prueba estos cambios e intenta ejecutar nuevamente la aplicación, verá que no se obtiene más que la pantalla del ejemplo anterior. Esto es debido a que no hemos tocado el punto de entrada de la aplicación, para indicarle que cargue la página *Dibujos.xaml* en lugar de la original. Recordemos que el código generado para el fichero *Application.xaml.cs* —que sirve como punto de entrada—, crea un objeto *MainPage* para instanciar la propiedad *RootVisual*. Basta con cambiar la instanciación a *Dibujos*, para tener el ejemplo funcionando (listado 3).

```
private void OnStartup(object sender, StartupEventArgs e)
{
    // Carga del control principal
    this.RootVisual = new Dibujos();
    //this.RootVisual = new MainPage(); (comentamos esta entrada)
}
```

Listado 3

Una vez hecho esto deberemos ver la salida correcta en el navegador. Blend 4.0 ya generó una página para probar el control (*Default.html*), que contiene lo básico para la instanciación de un control Silverlight 4.0. Aunque la página predeterminada que alojara el control en ejecución no está visible desde Blend, podemos abrir la carpeta que contiene el proyecto y examinar el HTML del directorio, seleccionando en el menú contextual de la solución la opción "Abrir carpeta en el Explorador de Windows".

En ejecución, como el control central es un *TextBox*, el usuario puede modificar el texto inicial, pero esa es toda la funcionalidad disponible de momento (figura 9). La

nota

Si el lector quiere verificar cuál es exactamente el código HTML que se genera, puede hacerlo lanzando la aplicación y abriendo el código fuente correspondiente a esa página desde el navegador.

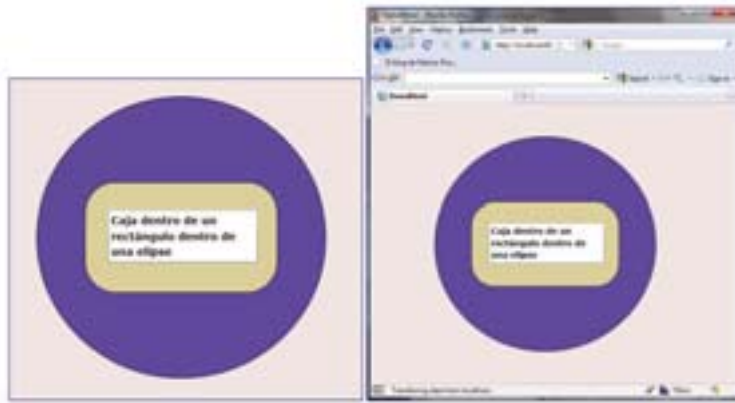


figura 9 Ventana de edición y salida del ejemplo de gráficos
(la salida es idéntica para FireFox, Google Chrome y Safari)

salida es idéntica en el diseñador, y en los navegadores IE9 Beta 1, FireFox, Google Chrome y Safari.

No hemos codificado nada para recoger la entrada del usuario en el **TextBox**. Se trata, simplemente, de demostrar que el aspecto y el funcionamiento son equivalentes a sus contrapartidas en Windows Presentation Foundation cuando se muestra en los navegadores más populares.

Trabajando con la Ventana de Diseño

Recordemos que Blend agrupa los controles y herramientas en categorías. Por ejemplo, bajo el símbolo , se encuentran los controles considerados contenedores de otros controles (a excepción de **VirtualizingPanel**, por razones prácticas). De la misma forma, el resto de símbolos del panel de la izquierda se refiere siempre a un tipo de acción, representada por los elementos de su categoría.



figura 10 Bloque de elementos contenedores

La superficie de diseño

La superficie de diseño, muestra 3 modos de trabajo: XAML, Diseño y *split* (mixto). Por defecto, se crea un elemento **Grid** (dentro de **UserControl**) que presenta 2 modos operativos: modo **Canvas** y modo **Grid**. Pueden alternarse pulsando en el símbolo .

La diferencia estriba en la forma de ubicar los elementos dentro del **Grid**. Por defecto, Blend utiliza los elementos de forma similar a Visual Studio: en modo **Canvas**, se utiliza posicionamiento absoluto, colocando los elementos mediante coordenadas respecto a su contenedor. En modo **Grid** (predeterminado), se utiliza el posicionamiento por fila y columna. Dentro de la superficie de diseño, si queremos seleccionar y desplazar un control interno podemos hacerlo usando la segunda flecha , llamada de "Selección Directa".

nota

Si pulsamos la barra espaciadora, el cursor se convierte en una mano con la que podemos mover todo el control contenedor en cualquier dirección. Además, pulsando la tecla Tab aparecen y desaparecen las ventanas laterales.

Si trabajamos en modo rejilla (**Grid**), lo más probable es que necesitemos definir filas y columnas. Una vez establecido el modo, el proceso es muy sencillo: basta con pasar el cursor por cualquiera de las bandas que delimitan la superficie del **Grid** para que aparezca un cambio en el cursor, indicándonos que podemos señalar una línea divisoria. Hay que observar el candado al lado de la línea: la marca de bloqueo. Cuando pulsamos sobre él para cerrarlo, eso indicará que esa fila (o columna) tiene un alto o ancho fijo, como se comprueba en el código XAML generado. El efecto final, junto al código fuente, puede verse en la figura 11:

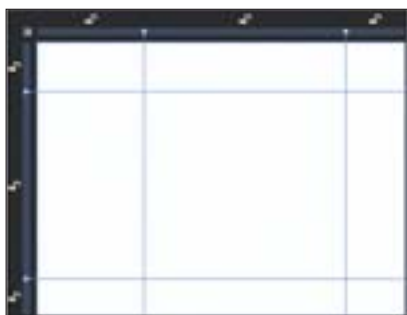


figura 11

Resultado del proceso de edición de filas y columnas

```

<Grid x:Name="LayoutRoot" Background="White">
    <Grid.RowDefinitions>
        <RowDefinition Height="0.181*"/>
        <RowDefinition Height="0.684*"/>
        <RowDefinition Height="0.134*"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="0.291*"/>
        <ColumnDefinition Width="0.548*"/>
        <ColumnDefinition Width="0.161*"/>
    </Grid.ColumnDefinitions>
</Grid>

```

Listado 4

Las marcas anteriores producen como salida el código XAML del listado 4 (recordemos que los números de las filas y columnas en un **Grid** comienzan por 0).

Brushes

En parte, Silverlight sigue los patrones de diseño que establece la especificación de documentos XPS¹, basada en XAML y presente en Windows 7/Vista/2008 (y descargable para Windows XP). La Tabla 1 recoge los tipos de brochas (*brushes*) definidos por esta especificación, todos ellos disponibles para WPF, y, a excepción de *Visual Brush*, también para Silverlight 4.0.

Nombre	Descripción
SolidColorBrush	Rellena una región con un color sólido
LinearGradientBrush	Rellena una región con un gradiente lineal
RadialGradientBrush	Rellena una región con un gradiente radial
ImageBrush	Rellena una región con una imagen
VideoBrush	Rellena una región con un vídeo
WebBrowserBrush	Rellena una región con el contenido de una página Web

Tabla 1. Tipos de *Brushes*

¹ Ver http://en.wikipedia.org/wiki/XML_Paper_Specification

Silverlight 4.0 presenta seis² tipos de *brushes*: **SolidColorBrush**, **LinearGradientBrush**, **RadialGradientBrush**, **ImageBrush**, **VideoBrush** y **WebBrowserBrush**. En ejemplos anteriores hemos visto el funcionamiento de los dos primeros (el tercero es una variante del segundo), así que vamos a ver aquí los más "mediáticos": **ImageBrush**, **VideoBrush** y **WebBrowserBrush**.

ImageBrush

ImageBrush permite dibujar la superficie de un control o geometría utilizando para ello una imagen de la que dispongamos como recurso en la aplicación. Vamos a probarlo con un ejemplo sobre dos posibles destinatarios: un control **Button**, y una geometría en forma de dos triángulos conectados por una de sus aristas, para poder probar las intersecciones con la imagen.

Lo único que puede no resultar muy intuitivo, es que se precisa haber cargado la imagen que vamos a utilizar antes de convertirla en un recurso de la aplicación (o de la página activa). Para ello, basta con un doble clic sobre la imagen, y dejar que Blend nos cree el elemento asociado. A continuación, con la imagen seleccionada (elemento **Image**), desde el menú "Tools", optamos por "Make Brush Resource" -> "Make ImageBrush Resource", y Blend nos creará un recurso asociado al control contenedor principal (o a toda la aplicación, si optamos por ello), añadiendo una entrada XAML similar a la del listado 5:

```
<UserControl.Resources>
  <ImageBrush x:Key="ImageBrush1" ImageSource="Imagenes/W7_Logo.png"/>
</UserControl.Resources>
```

Listado 5

La comprobación visual aparecerá en la solapa "Resources" de la ventana superior derecha, como se ve en la figura 12.

Posteriormente, para hacer que cualquier control adopte esa imagen como fondo (propiedad **Background**) basta con crear un enlace (*binding*), que apunte al recurso como elemento estático disponible (usando **StaticResource**). Veremos todo esto con más detalle, pero el código para vincular uno a otro es intuitivo y tiene esta forma (listado 6).

² En realidad existe un séptimo tipo: *TileBrush*, pero se trata de una clase abstracta que sirve de base a las clases "mediáticas": *ImageBrush*, *VideoBrush* y *WebBrowserBrush*.



figura 12 Recurso de imagen en Blend

```
<Button Height="184" HorizontalAlignment="Stretch" Margin="144,0,144,80"
  VerticalAlignment="Bottom" Content="Button" Background="{StaticResource
  ImageBrush1}"/>
```

Listado 6

Si preferimos que Blend realice la vinculación por nosotros, tenemos dos opciones: arrastrar el recurso sobre el elemento al que queremos aplicarlo (y un menú contextual nos permitirá seleccionar a qué propiedad nos interesa vincular), o bien, con el elemento seleccionado, abrir el menú contextual al lado de cada propiedad que está señalado por un pequeño cuadrado (figura 13), y seleccionar el recurso (aquí, **ImageBrush1**):



figura 13

Acceso al menú contextual de propiedades y propiedades del atributo *Background*

El mecanismo de vinculación utiliza sintaxis de **binding** presentada en WPF e indica que existe un recurso accesible en el propio control que estamos creando, de nombre **ImageBrush1** que es el que vamos a usar.

Para el caso de las geometrías, el proceso es conceptualmente parecido, solo que la propiedad a vincular con el recurso gráfico es la propiedad **Fill** del objeto **Geometry** que usemos, y —en este caso— podemos arrastrar el recurso sobre nuestro dibujo, y utilizar el menú contextual para la selección de la propiedad a vincular, como se muestra en la figura 14:



figura 14 Menú contextual de selección de propiedad a vincular con un recurso.

El código generado para el relleno del objeto **Path** es muy simple (listado 7):

```
<Path Data="M398,83 C198,83 545.99939,234 444.00006,297 L289.00107,234 z"
      Fill="{StaticResource ImageBrush1}" Margin="180.5,69.5,230.334,152.5"
      Stretch="Fill" Stroke="#FF000000" UseLayoutRounding="False"/>
```

Listado 7

Donde **Path** es el objeto tipo **Geometry** a usar, y está compuesto de dos atributos principales: **Path.Fill**, que determina como se rellena su interior, y **Path.Data** que define el dibujo en sí (el contorno).

El código anterior, produce una salida como la de la figura 15.



figura 15 Utilización de *ImageBrush* en geometrías

VideoBrush

Otra extensión de lo anterior consiste en la capacidad de hacer que la brocha sea un vídeo, y por lo tanto el contenido de fondo pase a ser dinámico. En principio, cualquier objeto del que podamos establecer su propiedad de relleno es válido, incluyendo un **TextBlock**:

El único truco está en no mostrar el vídeo a pesar de que lo declaramos como **MediaElement**; para ello, establecemos su valor de **Opacity** a 0. Lo demás, resulta bastante evidente a la vista del código del listado 8:

```
<UserControl
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d"
  x:Class="SilverlightDemo_blend.Demo_VideoBrush"
  d:DesignWidth="640" d:DesignHeight="480">

  <Grid x:Name="LayoutRoot" Background="Transparent" Width="450" Height="310" >
    <Border Background="Transparent" BorderBrush="Maroon" BorderThickness="12"
      CornerRadius="24" />
    <MediaElement Source="Videos/B.wmv" x:Name="Video_Brush" Opacity="0" />
    <TextBlock Text="Aula Vulcan" TextWrapping="Wrap" FontSize="120" FontFamily="Arial"
      Margin="8,8,232,193" FontWeight="Bold" TextAlignment="Center">
      <TextBlock.Foreground>
        <VideoBrush SourceName="Video_Brush" Stretch="Fill" />
      </TextBlock.Foreground>
    </TextBlock>
  </Grid>
</UserControl>
```

Listado 8

En la salida, las letras del texto "Aula Vulcan" son pintadas con un vídeo de demostración. Además, el lector que pruebe el código apreciará un efecto curioso: cada línea completa de texto (aquí hay dos, pero funcionaría con cualquier número de líneas) causa una reproducción individual del vídeo. Y todas sincronizadas (figura 16).



figura 16 Letras cuyo fondo es un vídeo en formato .wmv

WebBrowserBrush

Una de las novedades presentadas en la versión 4. Es similar a **VideoBrush**, pero en lugar de utilizar un vídeo, se usa el contenido de una página web, por lo que no tiene sentido en aplicaciones alojadas en una página, y solo se utiliza en aplicaciones *Out-Of-Browser*, cuando se desea mostrar contenido de páginas HTML. Remitimos al lector al capítulo 7 de esta obra para ver un ejemplo de su funcionamiento.

nota

Existe una aplicación OOB (Out-of-Browser) creada por el conocido divulgador John Papa en la que una página resultado de una búsqueda con Bing, es convertida en un puzzle, como implementación de un juego básico que usa esta característica. (Ver referencias en el Apéndice I, para más información).

Brushes clásicos: SolidColorBrush, LinearGradientBrush y RadialGradientBrush

Recordemos el proceso que seguimos para rellenar un elemento mediante colores, pero en Blend. Es también muy intuitivo. Solo tenemos que escoger el objeto que a colorear, y observar el Panel de propiedades a la derecha. En la figura 17 se ve la sub-ventana "Brushes" (el editor de brochas) y cómo podemos seleccionar la propiedad de destino (**Background**, **BorderBrush**, **Foreground** y **OpacityMask**) y el tipo de brocha que

vamos a usar para ello (el lector apreciará, si ha hecho el proceso anterior, que aparecen las brochas creadas por código XAML como un recurso igualmente utilizable, en la solapa "Resources").

La oferta de propiedades dependerá del tipo de objeto con el que estamos trabajando. En este caso, para no mezclar ideas, hemos limpiado la superficie de diseño. Una vez seleccionado un elemento, elegimos en la banda inferior el tipo de brocha (las flechas en la imagen). Puede ser de un color solido (**SolidColorBrush**) o de tipo gradiente entre dos o más colores (**GradientBrush**).

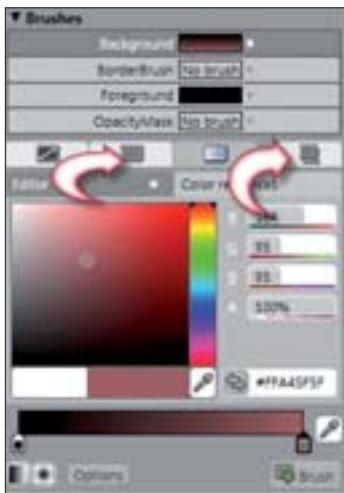


figura 17 Editor de colores basados en objetos Brush

También podemos establecer el valor exacto de un color a través de los valores ARGB que lo componen, o introducir un valor hexadecimal.

En el caso de optar por un gradiente de color, justo a continuación tenemos la banda para señalar los puntos de variación del gradiente. Cada una de las marcas que hagamos en esa banda va a definir en el código un objeto **GradientStop**, que establece el final de un gradiente y el comienzo del siguiente, para los casos en que hay más de dos.

Por último, en la zona más inferior del gráfico, se presentan los botones de selección de tipo de gradiente (puede ser lineal o radial). Y si seleccionamos la marca  con un gradiente seleccionado, una ventana desplegable nos mostrará más opciones adicionales disponibles (figura 18).

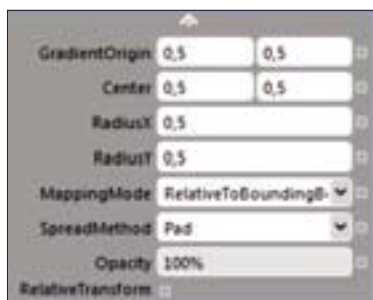


figura 18 Opciones de configuración de un gradiente

Otras sub-ventanas de propiedades

La ventana desplegable "Layout" nos permite configurar la posición de un elemento respecto a su contenedor u otra referencia válida. Un caso fácil de comprobar es el de un botón.

Seleccionemos uno en el conjunto de controles. Pongamos que queremos situarlo en la fila y columna inferior derecha del control. Como se ve en la figura 19, la ventana "Layout" nos presenta todas las propiedades disponibles, reconociendo dónde se encuentra ubicado el control.

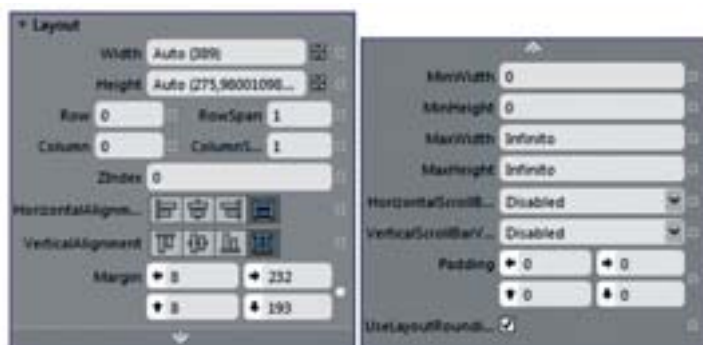


figura 19 Ventanas del editor de posiciones de un elemento (*Layout*), simple y desplegada

Los nombres de las propiedades resultan explicativos y —en muchos casos— idénticos a sus contrapartidas en Windows Forms. No obstante, tenga siempre en

cuenta la flecha que aparece en la parte inferior de esas sub-ventanas, que despliega otra ventana para hacer un ajuste más fino: anchuras y alturas máximas y mínimas, y propiedades del contenido del botón (el texto).

Búsqueda de propiedades

Hasta el momento, no hemos modificado el texto del botón para adaptarlo a la aplicación que estamos haciendo. Aquí entra en juego una propiedad interesante la solapa "Properties": como existen tantas propiedades, podemos utilizar la caja de texto para búsquedas, y según pulsamos el nombre de la propiedad, el entorno irá mostrándonos exclusivamente aquellas cuyo nombre empieza por lo tecleado (figura 20).

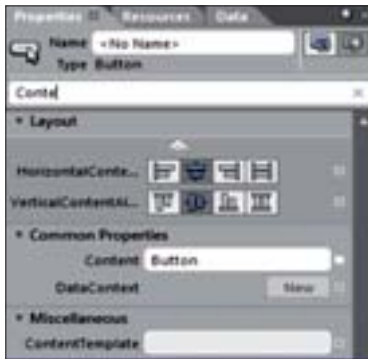


figura 20 Caja de selección de propiedades en la parte superior del Panel y resultado de una búsqueda

Ahí, cambiamos el contenido del botón para que diga simplemente "Pulse". Si queremos modificar las propiedades del texto escrito, (tipo de letra, tamaño, grosor, etc.), la siguiente sub-ventana ("Text"), nos habilita esa posibilidad, y la ventana "Font Manager" (figura 21), permite empotrar un tipo de letra no predeterminado para que se distribuya incrustado en la aplicación.

Resulta igualmente intuitivo establecer el formato del párrafo (¶), y ajustar las características de sangrado (☞). Como norma, todos los elementos visuales que podemos utilizar se encuentran representados en estas ventanas.

Y recordar que las sub-ventanas "Transform" y "Projection" permiten establecer transformaciones o grupos de transformaciones visuales y también configurar el elemento para que se comporte como si estuviera dispuesto en un espacio de 3 dimensiones (pu-



figura 21 Sub-ventana de selección de las propiedades del texto de un elemento y ventana del Gestor de fuentes

diendo girar en los 3 ejes espaciales), y lo que viéramos es la proyección del elemento en el plano bidimensional de la pantalla. Veremos más sobre su uso al hablar de animaciones.

Finalmente, la ventana "Miscelánea", contiene referencias a un conjunto de propiedades varias que pueden ser muy distintas para cada control y que no resultan agrupables en las categorías anteriores.

Los nuevos recursos en Blend 4

Hemos visto cómo en la solapa "Assets" aparecían los recursos disponibles de forma pre-determinada: "Behaviours", "Effects", "Shapes", "Styles" y otros. Hablaremos de los dos primeros nuevamente en el capítulo dedicado a las animaciones y efectos visuales, así como de los estilos (*styles*).

Conviene recordar que en el apartado "Shapes", aparece un conjunto de formas predefinidas que no están presentes por defecto al editar nuestra aplicación desde Visual Studio 2010. Ello es debido a que esos estilos son un "plus" de Blend 4, y pertenecen a la librería **Microsoft.Expression.Drawing.DLL** que solo se distribuye con este producto.

El lector encontrará un conjunto de formas adicionales típicas de los entornos de diseño. Más figuras geométricas, como triángulos, pentágonos, hexágonos, estrellas, anillos —toroide plano—, gráficos de sectores —"tartas"—, gráficos de llamadas

(*callouts*), flechas y arcos. Con ello se completa una oferta muy completa de posibilidades para el trabajo con gráficos.



figura 22

Controles disponibles en Expression Blend 4

La figura 22: muestra los controles que aparecen en la ventana Assets:

Conclusión

Blend es una herramienta pensada para diseñadores, pero ofrece interesantes aspectos de manipulación de código que no estaban presentes en versiones anteriores. Y hay también otros aspectos importantes en los que Blend es de gran ayuda para la creación de interfaces de usuario: transformaciones, animaciones, gestión de estados visuales mediante la herramienta Visual State Manager y enlace a objetos de negocio.

Controles en Silverlight 4

Entendemos por controles aquellos elementos definidos en XAML que heredan de la clase **Control** o sus derivadas, de forma que es muy amplio el abanico de elementos que podemos considerar como controles. No obstante, asumimos que lo son aquellos elementos visuales ya contruidos, que aparecen por defecto en las barras de herramientas de los IDE que utilizamos, o que podemos referenciar y utilizar como si fueran nativas¹.

Con el desarrollo de la plataforma, su número ha ido creciendo considerablemente, por no mencionar los controles adicionales que añade el Silverlight Toolkit, donde también se encuentra ahora el SDK Silverlight for Windows Phone 7 Toolkit, para la programación de dispositivos móviles con este sistema operativo.

Estos componentes son elementos terminados (pero abiertos), que generalmente contienen una estructura visual más elaborada, tanto en lo gráfico como en el funcional: eventos programables, propiedades, comportamientos, efectos, etc.

La creación de interfaces de usuario modernas, asume la existencia de estos controles a los que el usuario está acostumbrado y de los que conocemos su propósito (gráfico y operativo). Cajas de texto (**TextBoxes**), cuadros combinados (**ComboBoxes**), botones de opción (**OptionButtons**), árboles de elementos (**TreeViews**) y rejillas o tablas de datos (**DataGrids**), son recursos típicos de estas interfaces y así sucede en Silverlight.

Controles más usuales en Silverlight 4

En los dos IDE que utilizamos, habrá observado el lector que podemos acceder a las listas de controles disponibles tanto en Visual Studio 2010 (Caja de herramientas),

¹ Existe una amplia oferta de productos, tanto comerciales como de código abierto, que cubren una amplia gama de necesidades. Empresas como *Telerik*, *Infragistics*, *SyncFusion*, *ComponentOne*, *Xceed*, *Dundas*, etc., ofrecen amplias gamas de productos relacionados. Por supuesto, esto es extensible a nuestros propios controles.

como en Expression Blend 4 (ventana "Assets", en la parte inferior izquierda de este IDE), aunque hay presencias y ausencias en cada uno de ellos debidas a sus peculiaridades como herramienta.

Desde el punto de vista de la arquitectura en la jerarquía de clases, un control en Silverlight 4 es un objeto que hereda de **System.Windows.Controls.Control**, al menos, la inmensa mayoría de las veces, si bien hay elementos visuales que Expression Blend agrupa con los anteriores, sin que realmente pertenezcan a esta jerarquía, como los **Glyphs**², o los **PopUp**, que heredan de **FrameworkElement**, pero que aparecen vinculados con la interfaz de usuario, por tener un componente visual innegable.

El número de controles disponible, si contamos con el SDK del Silverlight Toolkit, supera con creces los que están disponibles para las aplicaciones Windows Forms. La figura 1 muestra la lista de controles comunes disponibles en Visual Studio 2010. (Un pequeño conjunto de los más usuales, que aparecen en un apartado propio). Pero la lista completa con todos los SDK instalados, incluyendo los temas (*themes*) y otros, es cercana al centenar de elementos.

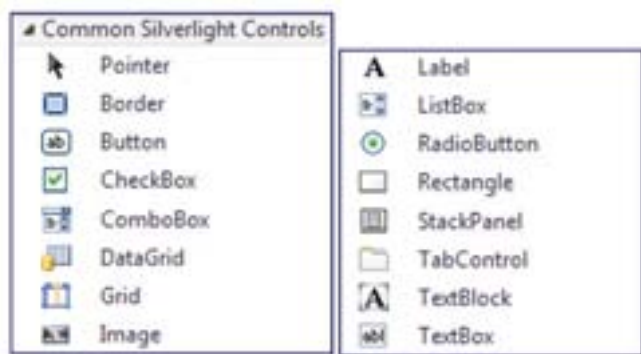


figura 1 Controles comunes de Silverlight 4.0 en la Caja de Herramientas de Visual Studio 2010

² Un **glifo** es un signo grabado o, por extensión, pintado. En tipografía, un **glifo** es una representación gráfica de un carácter, de varios caracteres o de parte de un carácter. Un carácter es una unidad textual mientras que un glifo es una unidad gráfica. En Silverlight 2.0 existe el concepto de *Glyph*, pero no es un control propiamente dicho, ya que no hereda de Control, sino solamente un elemento que se utiliza en la interpretación visual de texto fijo, o para presentar unidades de texto de forma más elaborada. Es más un elemento de diseño que de desarrollo.

Lo positivo de esta aproximación es que todos ellos tienen una sintaxis homogénea y una manera afín de declarar sus atributos y contenidos. El lector estará acostumbrado a utilizarlos en aplicaciones Windows o incluso web, ya que la mayoría tienen sus contrapartidas en esos entornos. Revisaremos el funcionamiento de los más populares en unos ejemplos y después, comentaremos varios aspectos del estado actual de las jerarquías de controles.

Controles que muestran/admiten texto como función primaria

Hay unos cuantos... **TextBox**, **TextBlock**, **CheckBox**, **RadioButton**, **DatePicker**, **DatePickerTextBox**, **HyperLinkButton**, **ListBox/ListBoxItem** (aunque puede mostrar cualquier cosa). Es cierto que otros controles también pueden mostrar texto, pero esa no es su función primaria (botones, **TabControls**, etc.).

También existen otros elementos XAML que no pueden considerarse controles estrictamente hablando, pero que intervienen en el manejo y visualización de texto, como **LineBreak** (salto de línea), **Run** (renglón), **Inline** (e **InlineCollection**), **Glyphs**, y **FontSource**. Todos estos pertenecen al espacio de nombres **System.Windows.Documents**.

nota

Obviamos los específicos de tratamiento de datos, como *DataGrid* y *DataForm*, así como los de representación de gráficos empresariales, ya que se salen del ámbito de esta obra.

Además, estrictamente hablando, solo los dos primeros tienen como función el manejo de texto, aunque estamos acostumbrados a que otros controles como **CheckBox** o **RadioButton**, presenten un texto explicativo que podemos asignar al propio control, o que muestren conjuntos discretos y bien definidos de texto para expresar otros formatos (como **DatePicker**).

Veamos un ejemplo en el que ponemos en funcionamiento un grupo de ellos, creando un formulario de entrada de datos, al que le añadimos dos botones (por ahora sin funcionalidad ninguna). El código completo (listado 1) es el siguiente.

Observe el lector la presencia de un espacio de nombres que se ha añadido al insertar el control **DatePicker: System.Windows.Controls**. Además, se ha incluido un hipervínculo, representado por el elemento **HyperLink** que apunta en su propiedad **Na-**

```
<UserControl x:Class="DemoControles2.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
    mc:Ignorable="d" d:DesignHeight="320" d:DesignWidth="425" >

    <Border BorderThickness="12" BorderBrush="LightBlue" CornerRadius="12"
        Width="420" Height="320">
        <Grid x:Name="LayoutRoot" Background="Azure">
            <Grid.RowDefinitions>
                <RowDefinition Height="50" />
                <RowDefinition Height="30" />
                <RowDefinition Height="35" />
                <RowDefinition Height="90" />
                <RowDefinition Height="30" />
                <RowDefinition Height="40" />
                <RowDefinition Height="21*" />
            </Grid.RowDefinitions>

            <TextBlock x:Name="txbNombre" HorizontalAlignment="Left" VerticalAlignment="Center"
                Height="24" Width="70" Text="Nombre:" TextWrapping="Wrap"
                Margin="20,0,0,0" Grid.Row="0"/>
            <TextBox Height="24" HorizontalAlignment="Right" VerticalAlignment="Center"
                Text="TextBox" TextWrapping="Wrap" Width="244" Margin="0,0,20,0"
                Grid.Row="0"/>
            <CheckBox Height="24" HorizontalAlignment="Left" VerticalAlignment="Center"
                Margin="20,0,0,0" Width="261.695" Content="Coche propio y Carnet de conducir"
                IsThreeState="False" Grid.Row="1"/>
            <sdk:DatePicker Text="Seleccione Fecha de entrada" Grid.Row="2" Height="23"
                HorizontalAlignment="Left" Margin="20,0,0,0" Name="datePicker1"
                VerticalAlignment="Top" Width="356" />

            <ListBox Grid.Row="3" Height="64" Width="170" Margin="-10,-10,0,0">
                <ListBoxItem>
                    <TextBlock>Analista Senior</TextBlock>
                </ListBoxItem>
                <ListBoxItem>
                    <TextBlock>Programador Junior</TextBlock>
                </ListBoxItem>
                <ListBoxItem>
                    <TextBlock>Especialista en IT</TextBlock>
                </ListBoxItem>
            </ListBox>
            <RadioButton Grid.Row="4" Content="Soltero" Margin="50,0,0,0"></RadioButton>
            <RadioButton Grid.Row="4" Content="Casado" Margin="170,0,0,0"></RadioButton>
            <RadioButton Grid.Row="4" Content="Viudo" Margin="280,0,0,0"></RadioButton>
        </Grid>
    </Border>
```

```

<HyperlinkButton Content="Pulse aquí para visitar dotNetMania"
    NavigateUri="http://www.dotnetmania.com" TargetName="_blank"
    Margin="20,15,0,0" Grid.Row="5" Height="12" VerticalAlignment="Center" />
<Button Content="Enviar" Grid.Row="5" Width="100" HorizontalAlignment="Right"
    Margin="0,15,20,0" />
</Grid>
</Border>
</UserControl>

```

Listado I

navigateUri a la dirección del sitio web de la revista. En este caso, para que la navegación se produzca a otra nueva ventana del navegador y no sustituya la existente, hemos asignado la propiedad **TargetName="_blank"**. En el diseñador visual de Visual Studio la salida es como la de la figura 2.



figura 2 Primer ejemplo de controles

ListBox y los elementos colectivos

Otro criterio distintivo que merece la pena reseñar es el de los elementos que albergan colecciones uniformes de objetos. Hemos añadido un control **ListBox** a esta lista, pero, a pesar de que su nombre y sus propiedades básicas recuerdan a las de sus

homólogos de Windows, tanto éste, como otros de corte similar, disponen de funcionalidades que eran muy difíciles de implementar en aquellos. En el caso del **ListBox**, los contenidos son elementos **ListItem**, que, al heredar de **ContentControl**, pueden albergar prácticamente cualquier cosa, así que las posibilidades son innumerables.

nota

Recuerde que la propiedad **Content** de un **ListItem** solo puede albergar un elemento, por lo que, si necesita mostrar varios en uno de los ítems, puede añadir un control de la familia **Panel** e introducir allí lo que sea necesario.

El ejemplo siguiente contiene un **ListBox** cuyos elementos muestran la imagen de conocidos arquitectos de software de Microsoft y también su nombre. Ambos elementos (**Image** y **TextBlock**) se han ubicado en sendos controles **StackPanel** con orientación horizontal (listado 2).

Y, en realidad, podríamos hacer todas las combinaciones que nos interesaran sin más limitaciones que las que impone su arquitectura. Por ejemplo, no hay inconveniente para que cada **ListItem** contenga un control **MediaElement**, que represente un vídeo, y así sucesivamente.

El resultado visual en el editor gráfico de Visual Studio 2010 sería el de la figura 3:



figura 3 Salida del código anterior con un **ListBox** personalizado

```

<Grid x:Name="LayoutRoot" Background="DarkCyan" >
  <Grid.RowDefinitions>
    <RowDefinition Height="50" />
    <RowDefinition Height="190" />
    <RowDefinition Height="40" />
    <RowDefinition Height="20*" />
  </Grid.RowDefinitions>
  <TextBlock Text="Bloggers" HorizontalAlignment="Center" VerticalAlignment="Center"
    FontSize="14" FontWeight="Bold" Foreground="#FFE2E293" />
  <ListBox Grid.Row="1" >
    <ListBoxItem>
      <StackPanel Orientation="Horizontal">
        <Image Source="Fotos/ahajlsberg.jpg" Width="48" />
        <TextBlock VerticalAlignment="Center" Margin="7">A. Hejlsberg</TextBlock>
      </StackPanel>
    </ListBoxItem>
    <ListBoxItem>
      <StackPanel Orientation="Horizontal">
        <Image Source="Fotos/don-box.jpg" Width="48" />
        <TextBlock VerticalAlignment="Center" Margin="7">Don Box</TextBlock>
      </StackPanel>
    </ListBoxItem>
    <ListBoxItem>
      <StackPanel Orientation="Horizontal">
        <Image Source="Fotos/Mark-Russinovich.jpg" Width="48"/>
        <TextBlock VerticalAlignment="Center" Margin="7">Mark Russinovich</TextBlock>
      </StackPanel>
    </ListBoxItem>
  </ListBox>
  <Button Content="Salir" Grid.Row="2" Width="100" HorizontalAlignment="Right"
    Margin="0,15,12,0"/>
</Grid>

```

Listado 2

Programación básica de eventos

La programación de los eventos asociados a los controles sigue el patrón habitual del entorno de trabajo de Visual Studio. Supongamos que añadimos un control **Rating** (visualmente consiste en un número de estrellas cuyo color de relleno indica una valoración entre 0->n). E imaginemos que queremos que cuando un usuario vota —selecciona un valor dinámicamente pulsando en el control— el color cambie indicando un *feedback* al usuario. Por ejemplo, queremos que inicialmente esté en rojo, y al votar se coloree en verde.

El aspecto estático de los controles después de añadir a la derecha del nombre un control de este tipo sería el siguiente:



figura 4 La misma ventana anterior con el control Rating

Lo único que hemos hecho es añadir un control **Rating** a continuación de cada caja de texto con el nombre de un arquitecto de software, usando el siguiente código (listado 3):

nota

El control Rating pertenece a la DLL System.Windows.Controls.Input.Toolkit.dll, del Silverlight Toolkit.

```
<my:Rating ItemCount="5" Value="0.8" VerticalAlignment="Center"
Name="Rating1" Foreground="#FFCB4141"
ValueChanged="Rating1_ValueChanged" />
```

Listado 3

Los controles han recibido los nombres **Rating1**, **2**, **3** y **4**, y se han configurado al azar sus valores significativos: **Value**, que representa el porcentaje de color pintado sobre el total, e **ItemCount**, que indica el número de estrellas que deseamos.

Finalmente, hemos programado el evento **ValueChanged**, que recogerá las acciones del usuario marcando el porcentaje.

Asignación de propiedades tipo DP (DependencyProperty)

Para realizar los cambios pedidos, hemos declarado en la clase un objeto **SolidColorBrush** con el color que queremos (verde). Y, en el manejador del evento, garantizamos que este se ha cargado completamente, y cambiamos su propiedad **ForegroundColor**. Como **ForegroundColor** es una propiedad de dependencia **utilizamos el método SetValue para asignarla**. Pero lo importante aquí es ver el mecanismo de programación básico del evento. El código siguiente de la clase **DemoListbox** (listado 4) produce el resultado deseado:

```
public partial class DemoListbox : UserControl
{
    SolidColorBrush br = new SolidColorBrush(Colors.Green);
    public DemoListbox()
    {
        InitializeComponent();
        Rating1.ValueChanged += Rating1_ValueChanged;
    }

    void Rating1_ValueChanged(object sender,
        System.Windows.RoutedPropertyChangedEventArgs<double?> e)
    {
        //Garantizamos que el elemento ha sido añadido
        if (Rating1 == null) return;

        //La asignación se hace mediante la //DependencyProperty asociada.
        Rating1.SetValue(ForegroundProperty, br);
    }
}
```

Listado 4

Algunos controles especiales

Muchos de los controles disponibles tienen ciertas peculiaridades respecto a los vis-
tos hasta aquí (la gran mayoría siguen estos patrones de comportamiento). O no
poseen un aspecto visual predeterminado (sino que son meros contenedores de
otros), o su tipo de contenido es específico y no general, o son utilizados para for-
mar parte de estructuras más complejas, o su aspecto visual debe ser definido por
otros medios.

En muchos casos, se usan como clases abstractas, pensadas para heredar de ellos
y aprovechar su implementación por esta vía. Vamos a comentar algunos de los más
sencillos, por el juego que pueden dar para construcciones desde cero, y para enten-
der mejor las jerarquías de clases.

ContentControl

Se trata de un control muy especial, porque sirve de base a muchos otros con-
troles y sub-controles. La jerarquía se muestra más detalladamente en el diagrama 1.

```
<Grid x:Name="LayoutRoot" Background="Transparent" ShowGridLines="True">
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="*" />
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>

  <TextBlock Text="Controles contenedores" Margin="0,20,10,20"
    FontFamily="Arial"
    FontSize="21" FontWeight="Bold" Foreground="Navy"
    HorizontalAlignment="Center" Grid.Row="0"/>

  <ContentControl Margin="3" Grid.Row="1" Background="Black"
    BorderBrush="Red"
    VerticalAlignment="Center"
    Content="Control contenedor con texto simple."
    HorizontalAlignment="Center" />

  <ContentControl Margin="3" Grid.Row="2" VerticalAlignment="Center"
    HorizontalAlignment="Center">
    <CheckBox Content="Este contiene un control CheckBox" />
  </ContentControl>
</Grid>
```

Listado 5

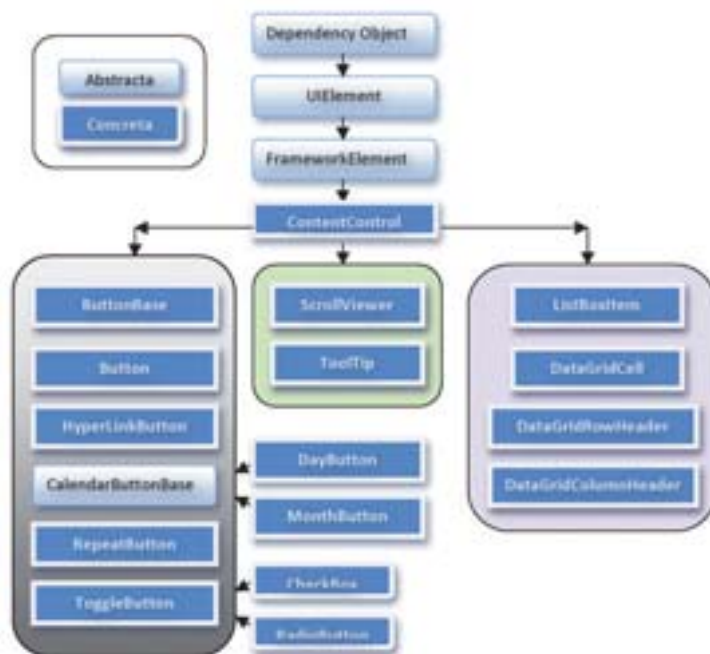


diagrama 1 Jerarquía de clases vinculada a ContentControl

ContentControl hereda de **FrameworkElement**, y —aunque no es lo más usual—, llega a convertirse en origen de clases abstractas, como **CalendarButtonBase**. Entre la jerarquía dependiente, hay una buena cantidad de versiones de la idea de **Button**, que a su vez da origen a otros elementos, etc.

No se suele utilizar directamente en una interfaz, pero no hay inconveniente para ello. El ejemplo siguiente (listado 5) muestra dos objetos **ContentControl** individuales en funcionamiento.

En el primero, su contenido se asigna mediante el atributo **Content**, ya que se trata de texto solamente, mientras que en el segundo, es un **CheckBox**. La salida visual sigue los patrones esperados (figura 5).

La propiedad **Content** de un **ContentControl** puede ser literalmente cualquier cosa, pero tiene algunas limitaciones:

- Debe ser un elemento que herede de **UIElement**
- Puede incluir otros objetos (pero, al visualizarlo, se llamará al método **ToString()** de cada objeto).



figura 5

Dos ContenControl con diferentes contenidos, tal y como se ven en la Ventana del Editor

- c) Puede incluir otros objetos con plantillas, pero si la propiedad **ContentTemplate** del control se asigna a una plantilla de datos, se usará esa plantilla, junto a las expresiones que contenga. Muy útil para colecciones de objetos.

Etiquetas Flotantes (ToolTip y ToolTipService)

No aparecen como control en la Caja de herramientas del IDE de Visual Studio por las razones comentadas antes: en realidad su funcionamiento es suministrado mediante un servicio (al estilo de otros de Windows Forms), llamado **ToolTipService**; este servicio, suministra métodos estáticos para mostrar una etiqueta flotante.

Su manejo predeterminado es a partir del elemento **ToolTipService** (que dispone de un único miembro, **ToolTip**, que hereda de **ContentControl**), y se declara incluyéndole en el elemento sobre el que queremos que aparezca la etiqueta. Por ejemplo, hemos modificado el **TextBlock** del elemento anterior para que muestre una simple etiqueta flotante con un texto, mediante al siguiente código (listado 6):

```
<TextBlock Text="Controles contenedores" Margin="0,20,10,20" FontFamily="Arial"
FontSize="21" FontWeight="Bold" Foreground="Navy"
HorizontalAlignment="Center" Grid.Row="0">
  <ToolTipService.ToolTip>
    <ToolTip BorderBrush="Navy" BorderThickness="5" FontFamily="Arial"
      FontSize="21" Background="Black" Foreground="Yellow">
      <TextBlock Text="Etiqueta flotante nº 1" />
    </ToolTip>
  </ToolTipService.ToolTip>
</TextBlock>
```

Listado 6

Este código crea una etiqueta flotante a la que se le aplica un estilo en la configuración del elemento **ToolTip** de **ToolTipService**.

Y la versión más simple del anterior (listado 7), solamente incluye el elemento a mostrar:

```
<ContentControl Margin="3" Grid.Row="2">
  <Border HorizontalAlignment="Center" BorderThickness="4" BorderBrush="Brown"
    CornerRadius="15">
    <TextBlock Margin="12,0,12,0" VerticalAlignment="Center" >
      Este contiene un elemento Border</TextBlock>
    <ToolTipService.ToolTip>
      <StackPanel Orientation="Horizontal">
        <TextBlock Text="Etiqueta flotante estilo predeterminado"/>
      </StackPanel>
    </ToolTipService.ToolTip>
  </Border>
</ContentControl>
```

Listado 7

Existen dos formas de modificar el aspecto de la etiqueta flotante: una es con plantillas de estilo que pueden ser aplicadas como recurso estático. La otra, tal y como hemos hecho aquí, se basa en la inclusión de un elemento **ToolTip** dentro de la declaración de **ToolTipService**. Si se hace así, dispondremos de un conjunto de elementos configurables.



figura 6 Distintos formatos de etiquetas flotantes

Los "Presenters"

Aunque parece un grupo musical, nos referimos al conjunto de controles que lleva ese sufijo, y que comparten algunos elementos comunes. Son cuatro: **ContentPresenter**, **InkPresenter**, **ItemsPresenter** y **ScrollContentPresenter** (que hereda directamente del primero). Todos pertenecen al espacio de nombres **System.Windows.Controls**.

Están pensados para servir de contenedores de otros, y especialmente, para suministrar una infraestructura común sobre la que aplicar estilos a los controles. Añadimos unos comentarios sobre uno un tanto especial: **InkPresenter**.

InkPresenter

Como sabrá probablemente el lector, **Ink** (literalmente, tinta) es un tipo de dato en .NET. Permite programar entradas de información por parte usuarios de dispositivos táctiles, como PDA, Pockets PC, Tablets PC, Windows Phone 7, etc. El objeto **InkPresenter**, dispone de una propiedad colectiva llamada **StrokeCollection**, que se compone de elementos **Stroke** (trazo). Cuando se añade un elemento a la colección, el objeto lo interpreta visualmente de forma inmediata.

InkPresenter hereda de **Canvas** y dispone de una de las colecciones de miembros públicos más grandes entre los objetos de la jerarquía a la que pertenece. Del grupo de los "Presenters", es quizá el que más se utiliza sin vinculación a ningún otro y sin jugar el papel de mero "ladrillo" constructor de otros controles más complejos.

Thumb, ToggleButton y RepeatButton

El control **Thumb**, está especialmente preparado para permitir el manejo de procesos de usuario del tipo Arrastrar-y-Soltar (*Drag&Drop*). Se utiliza como parte de

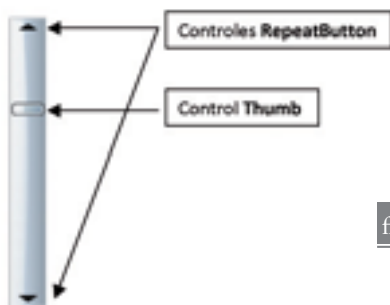


figura 5 Controles *Thumb* y *RepeatButton* como constructores de un *ScrollBar*

otros controles como **ScrollBar**, donde adopta la forma de un rectángulo desplazable a lo largo de la superficie limitada en ambos extremos por otros dos controles **RepeatButton**, pero puede utilizarse independientemente, sobre todo, en la construcción de controles de usuario.

La gestión del proceso *Drag&Drop* en un **ScrollBar** se realiza mediante 3 eventos: **DragStarted**, **DragCompleted** y **DragDelta**, que se lanzan cuando el usuario comienza el proceso, lo termina o utiliza la rueda del ratón teniendo el foco.

Al igual que los anteriores, su utilización real es mediante el concurso de estilos y plantillas, como también lo suelen ser los dos controles que nos quedan por comentar: **ToggleButton** y **RepeatButton**.

El primero sirve para implementar un tipo especial de botón multi-estado (presionado, no presionado). El segundo, para recoger entradas de usuario de tipo continuo. Para ello, lanza una secuencia de eventos **Click** desde el momento que se pulsa hasta que se libera, y el ejemplo típico es el de la estructura de un **Scrollbar**.

Más sobre el sistema de eventos en Silverlight

Todo documento XML es un árbol de nodos, y gran parte de ellos, contienen a su vez otros nodos, y así sucesivamente. En Silverlight existe un tipo especial de eventos llamados **Routed Events**: eventos "encaminados" o dirigidos desde un origen hasta un posible destino, que no puede ir más allá del nodo raíz del árbol XAML, en un fenómeno denominado *bubbling*, de manera similar a como sucede en el modelo de objetos de DOM con elementos HTML³. Silverlight, por el momento, soporta unos cuantos eventos de este tipo, relacionados con el teclado (**KeyDown** y **KeyUp**) y el ratón (varios), además del evento **Loaded**. Este último, está vinculado a la idea de desencadenador (**trigger**).

Bubbling

En los eventos que utilizan un mecanismo de *bubbling*, su segundo argumento de evento (**e**), posee siempre una propiedad **Handled** que permite que se detenga el proceso de "burbujeo" hacia nodos superiores, cuando el valor de la propiedad se asigna a **True**.

³ En WPF esto no tiene por qué suceder siempre de abajo-arriba en la jerarquía, sino que puede ser igualmente de arriba-abajo, mediante el mecanismo de "*tunnelling*".

Pero los nodos XAML, pueden lanzar otros eventos que pueden ser manejados al puro estilo .NET por procedimientos de evento adecuados. Incluso se puede provocar más de una respuesta, o permitir que varios eventos tengan un único manejador común. Veámos el funcionamiento de un solo manejador común en el capítulo anterior, dentro de ejemplo de los **sliders**.

También hemos comprobado que crear un manejador resulta inmediato en el entorno del IDE de Visual Studio (y de Expression Blend), ya que los nombres de los eventos disponibles para un objeto (o elemento), aparecen como atributos en el código XAML (o se presentan en un listado separado en la ventana de propiedades en Blend). Resumiendo, podemos declarar los manejadores de eventos de dos formas: como siempre hemos hecho, desde el código C#, o desde el propio XAML, como ya vimos antes.

```
<Grid x:Name="LayoutRoot" Background="Beige">
    <Border Background="BurlyWood" Width="400" Height="250">
        <StackPanel Orientation="Horizontal" Margin="8,8,8,8">
            <Rectangle x:Name="Thumb1" Fill="Blue" Width="110" Margin="8,8,8,8"
                HorizontalAlignment="Left" MouseEnter="Thumb1_MouseEnter" />
            <Rectangle x:Name="Thumb2" Fill="Blue" Width="110" Margin="8,8,8,8"
                HorizontalAlignment="Right" MouseEnter="Thumb1_MouseEnter" />
        </StackPanel>
    </Border>
</Grid>

public Pagina2()
{
    InitializeComponent();
    Thumb1.MouseLeave += Thumb1_MouseLeave;
    Thumb2.MouseLeave += Thumb1_MouseLeave;
}

void Thumb1_MouseLeave(object sender, MouseEventArgs e)
{
    Brush Fondo = new SolidColorBrush(Colors.Red);
    Thumb1.Fill = Thumb2.Fill = Fondo;
}

private void Thumb1_MouseEnter(object sender, MouseEventArgs e)
{
    Brush Fondo = new SolidColorBrush(Colors.Blue);
    Thumb1.Fill = Thumb2.Fill = Fondo;
}
```

En el ejemplo siguiente, declaramos dos procedimientos de evento para dos rectángulos, dentro de un **StackPanel**. Ambos cambiarán de color al entrar el ratón en las áreas visuales de cualquiera de ellos, y ambos volverán a asumir el que tenían, cuando el cursor salga. Las declaraciones de los eventos **MouseEnter** y **MouseLeave** son equivalentes, aunque una está en XAML y la otra en C# (listado 8).

Para indagar un poco más en profundidad lo que sucede, un vistazo con .NET Reflector⁴ a la clase **Rectangle**, nos muestra algunas cosas interesantes (listado 9).

```
public sealed class Rectangle : Shape
{
    // Campos
    public static readonly DependencyProperty RadiusXProperty;
    public static readonly DependencyProperty RadiusYProperty;

    // Métodos
    static Rectangle();
    public Rectangle();

    // Propiedades
    public double RadiusX { get; set; }
    public double RadiusY { get; set; }
}
```

Listado 9

La clase dispone de dos constructores: uno estático y otro dinámico, (algo muy común en .NET), pero —sobre todo— sus dos miembros públicos modificables (**RadiusX** y **RadiusY**), se corresponden con sendas variables estáticas (**RadiusXProperty** y **RadiusYProperty**) que se declaran como **DependencyProperty** (si seguimos profundizando, veremos que toda la jerarquía de **UIElement** se basa precisamente en ellos).

Por otro lado, la plataforma reconocerá automáticamente el tipo de evento que estamos declarando y construirá —igual que en .NET clásico— dos argumentos para el procedimiento de evento que nos indiquen quién lo ha disparado y qué argumentos se le pasan. Pongamos el caso de las entradas de teclado. Si queremos controlar el comportamiento de dos cajas de texto desde un control de nivel superior que contenga ambas cajas, podemos codificarlo de la siguiente manera (listado 10).

Con éste código averiguamos cuál de los **TextBox** se encuentra activo. El truco reside en que no utilizamos el argumento **sender**, sino la propiedad **OriginalSource** del

⁴ El producto es distribuido por Red Gate que continúa con su desarrollo. No obstante, sigue siendo gratuito (www.RedGate.com).

```

Grid x:Name="LayoutRoot" Background="White">
    <Canvas Width="400" Height="150" Background="BlanchedAlmond"
        KeyUp="ControlTeclado">
        <TextBox x:Name="T1" Width="100" Height="40" Margin="15,20"/>
        <TextBox x:Name="T2" Width="100" Height="40" Margin="15,70"/>
        <TextBlock Name="txtInfo" Canvas.Top="120" Canvas.Left="15"
            Text="Información:"/>
    </Canvas>
</Grid>

// y en el código de la clase gestionamos el
// evento KeyUp
void ControlTeclado(object sender, KeyEventArgs e)
{
    if (e.Key != Key.Unknown)
    {
        String msg = "La tecla " + e.Key;
        msg += " se pulsó cuando el foco estaba en " +
            (e.OriginalSource as TextBox).Name;
        txtInfo.Text = msg;
    }
}

```

Listado 10

segundo argumento (e). La misma técnica puede extenderse para todos los casos en los que sea conveniente manejar lo que sucede dentro de un colectivo de controles anidado en otro, a través de su control contenedor (figura 8).

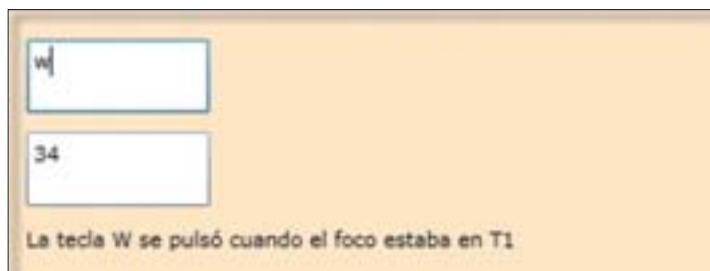


figura 8

Salida del código anterior

Estilos y Plantillas

Una de las herramientas más poderosas para diseñar el aspecto visual de un elemento en Silverlight 4, es la definición de estilos y/o plantillas propias. Esta técnica permite cambiar la apariencia y la estructura gráfica de los elementos XAML en sus dos tipos de comportamiento: estático y dinámico.

Podemos definir un estilo (elemento **Style**) como un recurso de toda la aplicación o de forma individual para una sola página o un simple elemento. El ámbito dependerá del lugar donde se declare el estilo, y de algunos atributos opcionales como **TargetType**, que indicará el tipo de objeto al que es aplicable, o **TargetName**, si queremos aplicarlo a un solo elemento con nombre. Más adelante, podremos utilizar ese mismo estilo en todos los elementos similares o que cumplan una condición establecida por nosotros.

Y si lo que se desea es cambiar la apariencia de un control más allá de sus propiedades directas, la técnica consiste en crear un **ControlTemplate**, que defina la apariencia y el comportamiento para todos sus estados posibles. Siempre que se trate de un elemento que herede de **FrameworkElement**, es posible asociarle un estilo.

Uso de estilos para cambiar la apariencia de un conjunto de controles

Por ejemplo, para dar una apariencia unificada a un conjunto de controles, lo más sencillo es crear un estilo personalizado y utilizar la propiedad **TargetType**, y así indicar a qué controles debe aplicarse. No obstante, para aquellos lectores que conozcan WPF, hay ciertas diferencias en la forma de utilizar estilos con Silverlight 4:

- En Silverlight, se deben usar atributos **x:Key** en los estilos personalizados y referenciarlos como recursos estáticos.
- A diferencia de los que sucedía en versiones anteriores, si un estilo no dispone de propiedad **x:Key**, pero se le asigna una propiedad **TargetType**, se asume que todos los controles de ese tipo usarán ese estilo (esta característica es nueva en la versión 4). Sin embargo, el estilo no se aplicará a elementos que deriven del valor asignado en **TargetType**.

En el listado siguiente, se declaran dos estilos aplicables a los elementos de un objeto **StackPanel**. Ambos disponen de una propiedad **TargetType**, pero no de una asignación **x:Key**, por lo que todos los elementos contenidos, asumen los valores por defecto correspondientes al tipo sin necesidad de asignarlo (listado 11).

```

<StackPanel Margin="0,0,6,0" Height="90">
  <StackPanel.Resources>
    <Style TargetType="TextBox">
      <Setter Property="Foreground" Value="Maroon" />
      <Setter Property="FontSize" Value="21" />
    </Style>

    <Style TargetType="Button">
      <Setter Property="Foreground" Value="Tomato" />
      <Setter Property="Background" Value="Red" />
    </Style>
  </StackPanel.Resources>

  <TextBox Height="40" Width="301"
    Text="TextBox con estilo implícito"/>
  <Button Height="30" Width="162" Margin="2"
    Content="Botón con estilo implícito" />
</StackPanel>

```

Listado 11



figura 9

Aplicación de estilos implícitos

Desde la versión 3.0, también es posible crear estilos basados en otros, con el consiguiente ahorro de código. Para ello podemos utilizar el atributo **BasedOn**.

Por ejemplo, podemos añadir un nuevo estilo a los anteriores, basándonos en el estilo de los **TextBox**, pero añadiéndole un atributo específico para esta definición, y aplicarlo después a otro **TextBox** para cambiar solo la apariencia de éste. Para esta operación, el estilo sobre el que nos basamos debe de tener un nombre (**x:Key**). Esto queda reflejado en el listado 12.

Y la salida generada por el código anterior la vemos en la figura 10.

Podemos definir estilos para sobrescribir otros predeterminados, pero si se intenta aplicar el mismo estilo otra vez provocaremos una excepción. Sin embargo, sí

```

<StackPanel Margin="0,0,6,0" Height="90">
  <StackPanel.Resources>
    <Style TargetType="TextBox" >
      <Setter Property="Foreground" Value="Blue" />
      <Setter Property="FontSize" Value="21" />
    </Style>
    <Style TargetType="TextBox" x:Key="EstiloTextBox">
      <Setter Property="Foreground" Value="Maroon" />
      <Setter Property="FontSize" Value="21" />
    </Style>

    <!--Crear un Style basado en otro-->
    <Style x:Key="EstiloHeredado" TargetType="TextBox"
      BasedOn="{StaticResource EstiloTextBox}">
      <Setter Property="Foreground" Value="Red" />
      <Setter Property="FontSize" Value="14" />
    </Style>

  </StackPanel.Resources>

  <TextBox Height="40" Width="301"
    Text="TextBox con estilo implícito"/>
  <TextBox Height="40" Width="301"
    Style="{StaticResource EstiloHeredado}"
    Text="TextBox con estilo heredado"/>
</StackPanel>

```

Listado 12

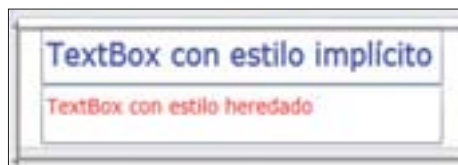


figura 10 Ejemplo de estilo heredado (basado en otro)

podemos cambiar los valores de propiedades individuales de un control que se hayan establecido utilizando estilos. La ventaja es que esta asignación dinámica llega hasta la posibilidad de establecer la propiedad **ControlTemplate** en tiempo de ejecución incluso si se ha asignado a un estilo.

Veamos cómo cambiar el estilo de un elemento simple (listado 13).

```

<Grid x:Name="LayoutRoot" Background="White">
    <StackPanel>
        <StackPanel.Resources>
            <Style x:Key="EstiloBoton" TargetType="Button">
                <Setter Property="Height" Value="20"/>
                <Setter Property="Margin" Value="12"/>
                <Setter Property="Background" Value="Magenta"/>
                <Setter Property="FontFamily" Value="Verdana" />
                <Setter Property="FontSize" Value="14"/>
                <Setter Property="FontWeight" Value="Bold"/>
            </Style>
        </StackPanel.Resources>

        <Button Content="Uno" Style="{StaticResource EstiloBoton}" />
        <Button Content="Dos" Style="{StaticResource EstiloBoton}" />
        <Button Content="Tres" Style="{StaticResource EstiloBoton}"
            Background="Blue"/>
        <Button Content="Cuatro" Background="Blue"/>
        <Button Content="Cinco" x:Name="BotonCinco" Loaded="BotonCinco_Loaded" />
        <Button Content="Seis" />
    </StackPanel>
</Grid>

```

Listado 13

En este ejemplo, se utilizan 5 formas de presentación de los botones: los dos primeros usan el estilo declarado en su elemento contenedor (**StackPanel**) mediante la asignación de la propiedad **Style** al valor "*EstiloBoton*" declarado como recurso del contenedor. El tercero usa una técnica mixta, aplicando el estilo como los dos anteriores, pero cambiando uno de sus valores: el color de fondo. El cuarto, no utiliza estilos, y simplemente cambia el valor de su color de fondo directamente. El quinto, usa una forma equivalente al cuarto, pero la asignación del valor se hace mediante código, creando un valor aleatorio cada vez que se carga el control (basta con refrescar la página del navegador para comprobarlo). Finalmente, el sexto botón no tiene ningún estilo ni atributo asociado, por lo que presenta su apariencia por defecto, para servir de comparación.

Este es el código asociado al evento **Loaded** del quinto botón (listado 14).

No obstante, el lector puede ver en la salida del código anterior, que, los botones, con o sin estilo aplicado, heredan una cierta *cualidad visual* predeterminada (incluyendo cambios visuales al pasar el cursor por encima): son parte de la plantilla que define ese control.

Muchas veces, las plantillas no sólo definen la apariencia (estática), sino el comportamiento visual (dinámico) de un control en sus distintos estados posibles. Pode-

```
private void BotonCinco_Loaded(object sender, RoutedEventArgs e)
{
    Random aleatorio = new Random(DateTime.Now.Millisecond);
    int R = aleatorio.Next(0, 255);
    int G = aleatorio.Next(0, 255);
    int B = aleatorio.Next(0, 255);
    BotonCinco.Background = new SolidColorBrush(Color.FromArgb(255, (byte)R,
                                                                (byte)G, (byte)B));
}
```

Listado 14

mos crear un estilo que reescriba el contenido de la propiedad **ControlTemplate**, y, al aplicarlo a un control, habremos cambiado totalmente su aspecto gráfico (figura 11).



figura 11 Salida del código anterior

Vamos a continuar con el caso anterior. Si ahora queremos hacer un botón, con los bordes redondeados y apariencia plana, podemos añadir lo siguiente a la definición del estilo anterior (listado 15).

Esto redefine la plantilla del botón, y por tanto, anula su plantilla predeterminada, pasando a tener una apariencia totalmente dependiente de la nueva (hemos cambiado la orientación del **StackPanel** contenedor de vertical a horizontal, pero esto no afecta a la apariencia gráfica de cada botón –figura 12–).

En la imagen, se muestran los 4 últimos botones. El cambio tan drástico se produce porque toda la apariencia está empotrada dentro del **ControlTemplate**. Si queremos permitir que se modifique manualmente un solo atributo, dejando que la plan-

```
Setter Property="Template">
  <Setter.Value>
    <ControlTemplate TargetType="Button">
      <Border Width="95" Height="35" BorderThickness="3" BorderBrush="Maroon"
        CornerRadius="16" Background="Beige" >
        <TextBlock Text="Botón" Foreground="Navy" HorizontalAlignment="Center"
          VerticalAlignment="Center"/>
      </Border>
    </ControlTemplate>
  </Setter.Value>
</Setter>
```

Listado 15



figura 12

Modificación en el botón anterior comparada con el aspecto de los otros botones

tilla se haga cargo del resto, y ese atributo está definido y recibe un valor en la plantilla, nuestra asignación manual no funcionará: seguirá tomando los valores de la plantilla.

Para resolver este problema se utiliza la asignación tardía mediante la palabra reservada **TemplateBinding**.

Utilización de TemplateBinding en plantillas

La técnica del enlace tardío permite contar con dos niveles de personalización (general y específico). Se puede utilizar la extensión de marcado **TemplateBinding** (solo dentro de la plantilla de un control), asignando como valor una propiedad accesible. Por ejemplo, al cambiar el código anterior, añadiendo **Background="{TemplateBinding Background}"** a los atributos del objeto **Border**, la salida gráfica sí que reconocería el cambio de fondo que habíamos aplicado al tercer botón de la serie, como podemos ver en la figura 13.

Ahora, el tercer botón tiene un fondo individualizado, aunque toma el resto de sus propiedades de la plantilla. Además, **ControlTemplate** podría ser mucho más complejo y contener, a su vez, otro control: un **Image** para añadir un gráfico o cualquier otro diseño que consideremos conveniente.



figura 13 La nueva plantilla, utilizando la extensión *TemplateBinding*

ContentPresenter

Relegamos hasta aquí los comentarios sobre el elemento **ContentPresenter**, que permite establecer la propiedad **Content**, en las plantillas de aquellos controles que la poseen. **ContentPresenter** debe utilizar un atributo **TemplateBinding** para asociar la propiedad **Content** de **ContentControl** con su equivalente de **ContentPresenter**. Mejor, verlo con un ejemplo:

```
<Grid x:Name="LayoutRoot" Background="White">
  <Grid.RowDefinitions>
    <RowDefinition Height="30"/>
    <RowDefinition Height="170"/>
  </Grid.RowDefinitions>

  <TextBlock Text="Demo de ContentPresenter" FontFamily="Verdana" FontSize="18"
    FontWeight="Bold" Foreground="#FF5C9AC9" Grid.Row="0"
    HorizontalAlignment="Center" />
  <Button Content="Botón" FontFamily="Verdana" FontSize="18" Grid.Row="1"
    Background="BurlyWood" Width="230" Height="90"
    HorizontalAlignment="Center" VerticalAlignment="Center">
    <Button.Template>
      <ControlTemplate TargetType="Button" x:Name="ButtonTemplate">
        <Grid>
          <Border BorderThickness="5" BorderBrush="Navy" CornerRadius="16"
            Background="{TemplateBinding Background}" />
          <ContentPresenter x:Name="ButtonPresenter"
            Content="{TemplateBinding Content}"
            HorizontalAlignment="Center"
            VerticalAlignment="Center" />
          <Image Source="Graficos/serv.png" HorizontalAlignment="Left"
            VerticalAlignment="Center" Margin="15,12,0,12"/>
        </Grid>
      </ControlTemplate>
    </Button.Template>
  </Button>
</Grid>
```

El código siguiente muestra un botón que define su **ControlTemplate** y su **ContentPresenter**. Además, incluye una imagen, y un fondo personalizados. El texto del botón está vinculado mediante la técnica que acabamos de comentar (listado 16).

Este código produce la salida de la figura 14, y es claro que el contenido de **ContentPresenter** puede ser cualquier combinación que el usuario desee (hemos escogido mantener el fondo de la imagen para que se aprecie bien la posición). **ContentPresenter** hereda de **FrameworkElement**.



figura 14 Salida del código del listado del elemento *ContentPresenter*

Por tanto, todo control puede ser modificado a partir de su plantilla original de presentación visual, además de los cambios que hagamos en su comportamiento mediante código. Eso significa que tenemos la capacidad de alterar el aspecto gráfico de cualquier control de una aplicación (predeterminado o no), o crear diseños que se conviertan en controles personalizados.

Expression Blend se presta especialmente bien para trabajar con las plantillas originales de los controles. Nos permite cambiarlos, crear una copia nueva solo para la página que estemos haciendo o generar todo un diseño desde cero, pudiendo manejar sus comportamientos en la interacción con el usuario, si le añadimos algo de trabajo con la herramienta **Visual State Manager**.

nota

El reducido espacio de este libro no nos permite adentrarnos más en otros aspectos avanzados del tratamiento de estilos, como son los *Themes* (Temas), disponibles en el Silverlight Toolkit. Para más información ver el sitio correspondiente de Codeplex.

Edición de estilos y plantillas desde Expression Blend

Basta con dibujar el control, y en el menú contextual, elegir la opción "*Edit Control Parts*" -> "*Edit a Copy*", y la estructura interna del control se despliega en la ven-

tana de código, lo que —además— resulta un ejercicio muy esclarecedor (por ejemplo, ver de que está hecho un **TextBox** puede resultar sorprendente). En el caso de un botón, al editar una copia, vemos que consta, en realidad, de todo un paquete de elementos, como se aprecia en la figura 15:



figura 15 Estructura interna de un botón

La estructura muestra que, internamente, un botón está formado por un **Grid** (que, a su vez, contiene otros) más una serie de rectángulos (**Background** y otros), **Brushes**, etc., la mayor parte de ellos para definir comportamientos visuales con el objeto Visual State Manager. Un vistazo más atento al código desglosado, nos presentará junto a cada elemento una serie de vínculos a esos recursos donde vemos la utilización de elementos que hemos estado estudiando en ejemplos anteriores: **ContentTemplates**, **TemplateBindings**, etc.

Omitimos reproducir el código XAML correspondiente por las restricciones de espacio, pero esos son los "ladrillos" constituyentes de que hablábamos antes. Como el mismo proceso puede repetirse para cualquier control predeterminado, tenemos una libertad sin precedentes a la hora de construir controles. Vamos a continuar con el sistema de animaciones y, al terminar, estaremos en disposición de crear nuestros propios controles (o versiones de los existentes) que incorporen todo esto.

Resumen

Operando de esta forma, Microsoft permite una aproximación mucho más personalizada a la creación y modificación de controles de la que teníamos con Windows Forms, y esa es una gran aportación del lenguaje XAML.

Disponemos de los elementos constituyentes y de los mecanismos programáticos para idear la creación de nuevos tipos de controles, para cualquier contexto, por muy específico que sea y con escaso esfuerzo, ya que, mediante herencia, accedemos a las funcionalidades básicas proporcionadas por sus "ladrillos" constructores.

Más allá de los controles

La nueva oferta para interfaces de usuario

El uso de controles y dibujos no determina todo lo que puede hacerse para enriquecer y potenciar la interfaz de usuario, tanto si admite interacción con el cliente, como si no.

Por una parte disponemos de un modelo de objetos vinculado a los elementos de la interfaz. Un modelo que permite cambiar lo que inicialmente vemos o hemos diseñado, y que puede enlazarse con otro sub-modelo (el de las animaciones), para crear lo que las interfaces Windows Forms no eran capaces de hacer: proporcionar movimiento, transformación y cambio. Y hacerlo tanto en el plano como en el espacio.

Por otro lado, las versiones 3 y 4 de Silverlight, han visto la aparición de elementos de apoyo que potencian sobremanera la construcción de objetos más sofisticados y también de su funcionalidad vinculada, su comportamiento. Los objetos **Effect** y **Behavior** (que hemos visto en Blend) son un ejemplo de ello. De igual manera, lo es el soporte de *Triggers o desencadenadores*, que establecen las condiciones para que algo suceda, y se vinculan en cierta forma con el gestor Visual State Manager, que nos permite un control adecuado de los distintos estados posibles de un elemento de la interfaz (con foco, sin él, deshabilitado, etc.).

Transformaciones

Dos de estas características avanzadas disponibles Silverlight son las transformaciones y las proyecciones en el espacio, también llamadas 2,5D¹.

Una transformación establece cómo corresponder puntos de una coordenada del espacio a otra utilizando una matriz de transformación (que provee un mecanismo de

¹ Ya que no se trata de 3D real, sino de la proyección en 2D de una estructura 3D.

conversión simple). En el lenguaje XAML de Silverlight 4.0, existen 4 clases predefinidas de transformaciones: **rotaciones**, **escalado**, **elongaciones** y **traslación** (movimiento). Además, es posible definir nuestras propias matrices de transformación, que pueden usarse, por ejemplo para combinar varias transformaciones². De forma adicional, podemos utilizar la nueva característica **Effects**, para crear efectos visuales de sombreado y desenfoque.

Transformaciones

Las transformaciones se crean aplicando *propiedades de transformación*, a distintos tipos de objetos. Dependiendo del elemento, las transformaciones soportadas serán, también, distintas (por ejemplo, los objetos **Brush** admiten varias formas, los objetos **Geometry** utilizan su propiedad **Geometry.Transform** y los que pertenezcan a la categoría **UIElement**, utilizarán la propiedad **RenderTransform**). Son objetos declarados en XAML, y por tanto, si las identificamos con un nombre, podremos modificarlos posteriormente mediante código C# o VB.NET. Veamos algunos ejemplos.

Propiedades RotateTransform y ScaleTransform

La primera, permite girar un elemento un ángulo especificado alrededor de un punto dado en el sentido de las agujas del reloj. Por ejemplo, el código siguiente produce una rotación de un texto tal y como muestra el listado 1 y aparece en la figura 1.

Como puede verse en el código de la figura adjunta, añadimos el atributo **RenderTransform** al cuerpo del control (en este caso, el **TextBlock**), y especificamos una transformación por rotación, con un ángulo de -30° . Una imagen visual para darse cuenta exactamente del proceso, consiste en "clavar un alfiler" (mentalmente) en las coordenadas indicadas, y rotar toda la superficie del control afectado los grados que se indiquen.

En el caso de la transformación de escala (**ScaleTransform**), se sigue un patrón similar al anterior, pudiendo indicarse la dirección del escalado y el punto a partir del cual deseamos que se haga. El código siguiente, produciría un escalado en el eje de las

² Aunque el concepto de transformación es similar, no debe confundirse con las conocidas Transformadas de Fourier (para más datos al respecto de este aparato matemático ver http://es.wikipedia.org/wiki/Transformada_de_Fourier y también (en inglés) "An Intuitive Explanation of Fourier Theory", en <http://entos2.area.trieste.it/russo/LabInfoMM2005-2006/ProgrammaEMaterialeDidattico/daStudiare/009-FourierInSpace.html>

```
Grid x:Name="LayoutRoot" Background="White">
  <TextBlock Margin="38.312,129.195,153.266,0" TextWrapping="Wrap"
    Text="dotNetMania, la revista del programador de .NET"
    FontSize="26.667" FontWeight="Bold" Foreground="#FF6A3434"
    RenderTransformOrigin="0.5,0.5" UseLayoutRounding="False"
    d:LayoutRounding="Auto" Height="82.117" VerticalAlignment="Top">
    <TextBlock.RenderTransform>
      <CompositeTransform Rotation="-30"/>
    </TextBlock.RenderTransform>
  </TextBlock>
</Grid>
```

Listado 1



figura 1 Texto rotado mediante una transformación

X de un rectángulo (tenga en cuenta que **RenderTransform** solo se puede establecer una vez –listado 2–).

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>
    <Rectangle Width="150" Height="50" Canvas.Left="150" Fill="Blue">
      <Rectangle.RenderTransform>
        <ScaleTransform ScaleX="1.5" CenterX="150" />
      </Rectangle.RenderTransform>
    </Rectangle>
  </Grid>
</Page>
```

Listado 2

De la misma forma, para aplicar las Transformaciones de elongación y Traducción la propiedad **RenderTransform** aplicada al mismo rectángulo podría tomar los siguientes valores (listado 3):

```
<Rectangle.RenderTransform>
  <SkewTransform AngleX="45" />
</Rectangle.RenderTransform>
```

Listado 3

```
<Rectangle.RenderTransform>
  <TranslateTransform X="-30" Y="-30" />
</Rectangle.RenderTransform>
```

Listado 4

Para una elongación o estiramiento de 45 grados, y (listado 4) para un desplazamiento de 30 píxeles a la izquierda y arriba respecto a su posición inicial.

Creación de transformaciones mediante código

También es posible crear transformaciones de forma dinámica, sin ningún tipo de diseño previo. Para ello, solo necesitamos un elemento que sirva de contenedor, e ir creando los objetos de la transformación con sus propiedades, para incluirlos finalmente en dicho contenedor. La técnica es muy similar a la utilizada para la creación dinámica de animaciones, por lo que remito al lector a ese aspecto que tratamos más adelante con un ejemplo completo.

Matrices de Transformación y TransformGroups

Para los casos en los que deseamos realizar una transformación múltiple, disponemos de dos opciones: una basada en la matemática matricial, que se fundamenta en de-

finir una Matriz de transformación, mediante el elemento **MatrixTransform** y la otra, mucho más sencilla, consistente en utilizar el elemento **TransformGroup**, que permite exactamente eso: agrupar en un solo proceso un conjunto de transformaciones diversas.

En el siguiente ejemplo, tenemos un grupo múltiple de transformación que produce simultáneamente las transformaciones de rotación, escalado y elongación (listado 5):

```
Rectangle.RenderTransform>
<TransformGroup>
  <RotateTransform Angle="45" CenterX="150" CenterY="30" />
  <ScaleTransform ScaleX="1.5" CenterX="150" />
  <SkewTransform AngleX="45" />
</TransformGroup>
</Rectangle.RenderTransform>
```

 Listado 5

3D en Silverlight: proyecciones

A partir de la versión 3.0, Silverlight presentó un nuevo tipo de transformación llamado transformación de perspectiva, o proyección 3D, consistente en la proyección en 2D de un objeto 3D (de ahí que algunos la denominen 2.5D, en el sentido de que no se trata de 3D real). Estas transformaciones pueden aplicarse a cualquier objeto que herede de **UIElement**.

El sistema permite realizar giros en los 3 ejes espaciales. La propiedad que permite establecer la definición es **Projection**. Y dentro de ella, podemos establecer las propiedades vinculadas a **PlaneProjection** ajustando los atributos **RotationX**, **RotationY** y **RotationZ** para establecer las propiedades del giro en el espacio indicando va-



figura 2

El *TextBlock* anterior, sometido a una transformación 3D

lores tipo **double** que oscilen entre 0 y 360. Volvemos sobre nuestro ejemplo inicial de este capítulo, para generar una rotación del elemento **TextBlock**, y generar la salida de la figura 2.

La sintaxis vinculada con la transformación de la figura 2 puede verla en el listado 6.

```
<TextBlock Margin="38.312,129.195,153.266,0" TextWrapping="Wrap"
  Text="dotNetMania, la revista del programador de .NET" FontSize="26.667"
  FontWeight="Bold" Foreground="#FF6A3434" RenderTransformOrigin="0.5,0.5"
  UseLayoutRounding="False" d:LayoutRounding="Auto" Height="82.117" VerticalAlignment="Top">
  <TextBlock.Projection>
    <PlaneProjection RotationX="38.398" RotationY="-57.525" RotationZ="-36.816"/>
  </TextBlock.Projection>

  <TextBlock.RenderTransform>
    <CompositeTransform Rotation="-29.166"/>
  </TextBlock.RenderTransform>
</TextBlock>
```

Listado 6

Adicionalmente, podemos añadir propiedades de translación desde la posición original (**LocalOffset**), o desde el origen de coordenadas (**GlobalOffset**). También es posible crear efectos vistosos combinando estas propiedades con la capacidad de modificar el centro de giro que ofrecen las propiedades **CenterOfRotationX**, **CenterOfRotationY** y **CenterOfRotationZ**. De forma predeterminada, los ejes de giro atraviesan directamente el centro del objeto, haciendo que éste gire alrededor de su centro; pero si se mueve el centro de giro al borde exterior del objeto, girará alrededor de ese borde. Los valores predeterminados de **CenterOfRotationX** y **CenterOfRotationY** son 0,5, y el valor predeterminado de **CenterOfRotationZ** es 0.

Con una adecuada modificación de los centros de rotación pueden conseguirse fácilmente efectos visuales como el paso de páginas, carruseles, etc.

Por último, en el caso de tener que realizar transformaciones espaciales más complejas, podemos recurrir a los elementos **Matrix3D** y **Matrix3DProjection**. Este último proporciona una matriz de transformaciones 3D completa para aplicar a cualquier **UIElement**, haciendo así posible aplicar matrices de transformación arbitrarias y matrices de perspectiva a los elementos de la interfaz. No obstante, el trabajo con estos elementos conlleva más trabajo adicional, ya que se trata de API básicas.

Efectos

Otro aspecto novedoso a partir de la versión 3 de Silverlight, consiste en la disponibilidad de efectos visuales predefinidos que pueden aplicarse igualmente a cualquier elemento (**UIElement**), utilizando simplemente un atributo de propiedad.

Por el momento, los dos únicos efectos disponibles de manera predeterminada son **DropShadowEffect** y **BlurEffect**, que producen, respectivamente, un efecto de sombra configurable y un efecto de distorsión o desenfoque.

La sintaxis es sencilla (listado 7), y puede aplicarse de la siguiente forma (por ejemplo, al botón utilizado en el ejemplo anterior):

```
<Button Content="Button" Height="99" Width="268" >
  <Button.Effect>
    <DropShadowEffect Color="Blue" Direction="320"
      ShadowDepth="15" BlurRadius="7"
      Opacity="0.5" />
  </Button.Effect>
</Button>
```

Listado 7



figura 3

Botón con un efecto de sombra aplicado

Lo que produce una salida como la de la figura 3.

Aunque no entra en el ámbito de esta obra, conviene indicar que es posible crear nuestros propios efectos personalizados. Para ello, se utiliza un lenguaje específico, propio del SDK de DirectX, llamado *High-Level Shader Language* (HLSL). El lector puede encontrar más información relacionada en los sitios de referencia indicados al final de esta obra.

Otros elementos vinculados con Blend

Ya hemos visto en anteriores capítulos que, desde el entorno de desarrollo de Microsoft Blend, se dispone de ciertos elementos que no se encuentran al trabajar con Silverlight desde Visual Studio. En el caso de los efectos que acabamos de comentar, el sitio de Microsoft Expression dispone de recursos gratuitos en forma de DLL o ficheros de código fuente que definen nuevos efectos que podemos descargar sin costo alguno. En cualquier caso, estos elementos forman parte de lo que se denomina el SDK de Blend, y se vincula con librerías como **Microsoft.Expression.Media** y **Microsoft.Expression.Interactivity**. En otros casos, las librerías son totalmente independientes, y debemos hacer referencia a ellas, para poder usarlas en cualquiera de los dos entornos de desarrollo.

Behaviors y Triggers

Al igual que sucede con los estilos (*Styles*) que "sacan factor común" de elementos de presentación y los ubican separadamente para poder usarlos después, desde la aparición de Silverlight 3 se nos presenta una opción similar, pero con la funcionalidad: se separa el comportamiento de los elementos en clases y luego puede aplicarse esa funcionalidad en donde se desee. Aunque generalmente se habla de *Behaviors* (comportamientos, en inglés), en realidad hay al menos dos elementos vinculados con este proceder: **Behaviors** y **Triggers**.

Behaviors

Como hemos indicado antes, tienen por objeto añadir funcionalidad a un elemento. La mayor parte de las veces responden a múltiples eventos. Existen varios predefinidos, que podemos ver desde la ventana "Assets" de Expression Blend, como aparece en la figura 4.

Basta con arrastrar uno de estos comportamientos sobre un elemento de la interfaz de usuario, para que automáticamente pase a formar parte de este (en su definición XAML) y podamos utilizarlo.

Desde el punto de vista del código, el resultado de arrastrar y soltar un *behavior* como "*PlaySoundAction*", sobre un botón y configurar un origen para la reproducción del sonido (previamente copiado al directorio de la aplicación), es el siguiente (listado 8).

Y hay que observar que, en las definiciones de espacios de nombres, aparecerán las dos referencias:

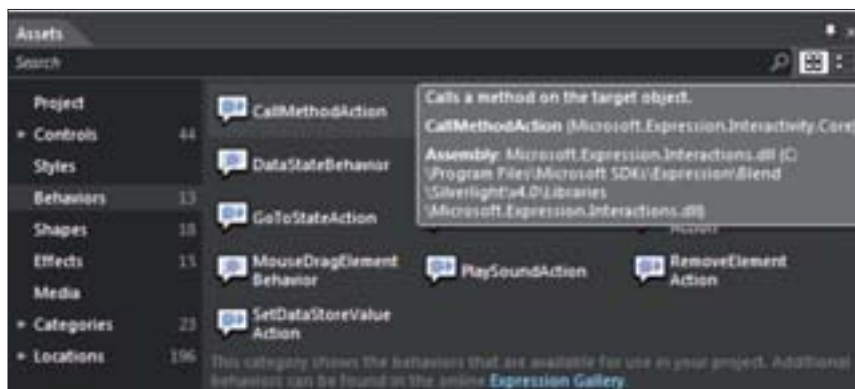


figura 4 Behaviors disponibles desde la ventana Assets de Blend 4.0

```
<Button Content="Button" Height="99" Width="268">
  <i:Interaction.Triggers>
    <i:EventTrigger EventName="Click">
      <ei:PlaySoundAction Source="/Supernatural-Smooth.mp3"/>
    </i:EventTrigger>
  </i:Interaction.Triggers>
</Button>
```

Listado 8

```
xmlns:ei="http://schemas.microsoft.com/expression/2010/interactions"
xmlns:i="http://schemas.microsoft.com/expression/2010/interactivity"
```

Como estos elementos se pueden seleccionar en la ventana "*Objects & Time Line*", al hacerlo, vemos que sus propiedades muestran un conjunto de ítems configurable. La ventana de configuración del **behavior** en Blend, nos ofrece las opciones que aparecen en la figura 5.

En este caso, se enlaza un objeto desencadenador (**Trigger**), del tipo **EventTrigger** (ver apartado siguiente), de forma que ponga en marcha el comportamiento al pulsar el botón, pero podríamos seleccionar cualquier evento de los disponibles. Este desencadenador, a su vez, puede ser de varias clases (figura 6), que muestra la ventana de selección de **TriggerType**, donde vemos otros tipos de desencadenadores pertenecien-

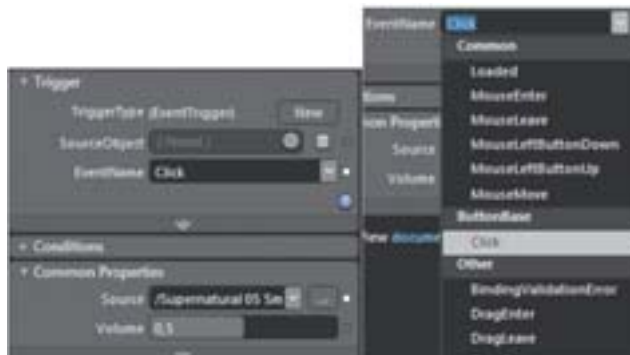


figura 5 Configuración del *behavior PlaySoundAction*, y *ComboBox* de selección del evento asociado

tes a la librería `Microsoft.Expression.Interactions`, como los `DataTriggers`, `PropertyChangedTriggers`, `TimeTriggers`, etc.

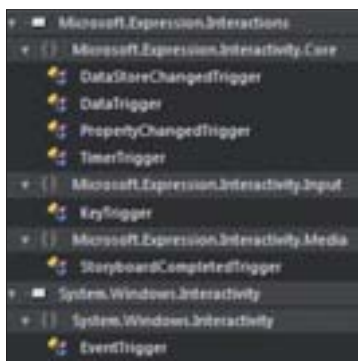


figura 6 Tipos de Triggers utilizables desde Blend 4

Gracias a este mecanismo, no se precisa código alguno para poner en marcha la acción, ya que el disparador es el encargado de esa función. Con el conjunto actual de **Triggers** disponible, como el **DataTrigger**, podemos forzar a que el disparador lance el proceso solamente cuando una propiedad alcance un cierto valor, o hacer que la ejecución dependa del momento en que una animación termine (**StoryboardCompletedTrigger**).

Triggers

Las animaciones son procesos que tienen lugar siempre como *respuesta a un evento*. Hemos visto que este evento, se puede expresar en XAML usando un **trigger**, que es representado como un atributo más del elemento.

Pero como un elemento puede responder a más de una acción, se dispone de una colección de **Triggers** vinculados con dispositivos (de ratón, de teclado, etc.) e incluso de **triggers** relacionados con otros aspectos, como los indicados en el apartado anterior.

El trigger más simple sólo en código XAML

Vamos a ver un ejemplo sencillo de esto partiendo de un rectángulo, aunque nos adelantemos al apartado siguiente referente a las animaciones. Supongamos que queremos que se produzca una animación consistente en desplazar un rectángulo por el eje de las X hasta una posición dada y volver a su posición inicial. Esto es posible sin escribir más que código XAML, si utilizamos un **trigger**. Si lo que queremos es que

```
<UserControl x:Class="Animacioni2.Page"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="400" Height="300">
    <Canvas x:Name="LayoutRoot" Background="White">
        <Rectangle x:Name="rect" Fill="BlueViolet" Canvas.Top="100" Canvas.Left="10"
            Width="100" Height="100">
            <Rectangle.Triggers>
                <EventTrigger RoutedEvent="Rectangle.Loaded">
                    <BeginStoryboard>
                        <Storyboard x:Name="Animacion2">
                            <DoubleAnimation Storyboard.TargetName="rect"
                                Storyboard.TargetProperty="(Canvas.Left)"
                                AutoReverse="True" Duration="0:0:5" To="300"/>
                        </Storyboard>
                    </BeginStoryboard>
                </EventTrigger>
            </Rectangle.Triggers>
        </Rectangle>
    </Canvas>
</UserControl>
```

se produzca como respuesta a un evento, deberemos de incluir la animación dentro de un recurso con nombre (veremos esto con otro ejemplo), y llamarla desde código no-XAML.

El código siguiente genera un rectángulo, y lo desplaza hasta el punto 300 del eje de las X, volviendo nuevamente a su posición original. La duración de cada desplazamiento se ha establecido en 5 segundos (listado 9).

Lo único que hemos hecho es añadir un **Trigger** a la colección de **Triggers** del elemento **Rectangle**, asociar el **trigger** con el evento **Loaded** y establecer sus propiedades. Aquí hemos incluido el objeto **DoubleAnimation** dentro de un elemento **Storyboard**, y éste, dentro de un **BeginStoryboard**, que solo puede contener un **Storyboard**, y es el encargado de distribuir las animaciones que contiene a los objetos y propiedades adecuadas.

A continuación veremos cómo funciona el sistema de animaciones.

El sistema de animaciones de Silverlight (Silverlight 4 Animation System)

Lo que hemos hecho hasta este punto, trata con elementos estáticos. Como mucho, los transforma a partir de una posición inicial. Pero, si queremos incluir aspectos dinámicos, la tecnología asociada será **Silverlight Animation System**, continuamente mejorada de versión a versión, que dispone de potentes herramientas para manejar cambios de estado, como Visual State Manager, la característica *Fluid UI* (implementada de forma especial)³, y otras tecnologías que resumiremos en este capítulo.

Animaciones

La interfaz de usuario del futuro se caracteriza por su dinamismo. Y ese dinamismo está asociado al concepto de animación. El dinamismo ha sido uno de los elementos diferenciadores de **Adobe Flash**, y es una de las grandes virtudes de **Silverlight**. La ventaja es que podemos programarlo utilizando los mismos elementos que usamos para otros desarrollos: XAML en la definición y un lenguaje .NET en el comportamiento.

³ Ver el excelente artículo "The Fluid UI in Silverlight 4" de Charles Petzold en la dirección <http://msdn.microsoft.com/en-us/magazine/ff798276.aspx>

Comencemos por la definición. Entendemos por animación una *ilusión visual*, creada mediante la proyección rápida y continua de varias imágenes o gráficos, cada uno de ellos ligeramente distinto del anterior. Las técnicas para la creación de vídeos no son más que un sistema de animaciones de fotogramas. Pero no vamos a ocuparnos aquí de los vídeos, sino de las animaciones generadas por nosotros.

Animaciones en Silverlight

En Silverlight, las animaciones siempre se relacionan con elementos visuales (XAML) y se dice que *animamos un objeto XAML aplicando elementos de animación a sus propiedades individuales*. Por ejemplo, para que un elemento de la interfaz de usuario crezca, procedemos a aumentar progresivamente sus propiedades **Width** y/o **Height**. Si queremos que se oculte progresivamente, vamos reduciendo el valor de su propiedad **Opacity**, y así sucesivamente. La versión de XAML soportada por Silverlight contiene muchos objetos cuyas propiedades admiten animaciones.

Ahora bien, solo pueden animarse aquellas cuyos valores sean del tipo **double**, **color** o **point**. La excepción es que también pueden animarse valores de propiedades mediante la denominada *interpolación discreta* (el salto de un valor a otro sin pasar por estados intermedios, que realmente, no es una forma de interpolación). Veremos después algo más sobre los dos tipos de interpolaciones soportadas (lineales y no-lineales) y la forma de programarlos.

Así pues, una animación se basa en *transiciones de estado*. Es decir, el valor de una propiedad varía desde un estado inicial hasta otro estado o valor final, produciendo la sensación visual de movimiento, o más generalmente, de cambio (los cambios de color o visibilidad, no implican movimiento). Esas transiciones de estado tienen lugar a lo largo de un período de tiempo, y existen dos conceptos vinculados a toda animación que son fundamentales: la *línea de tiempo* (*timeline*) y la *duración* del proceso de animación (*duration*). La duración es simplemente un valor escalar que indica al sistema cuando debe concluir el proceso después de iniciado. La línea de tiempo, es algo más compleja y permite otras posibilidades tales como establecer un comportamiento para la transición entre estados, en caso de que no deseemos que ésta sea uniforme (por ejemplo, puede ir rápido al principio, más lento en el medio y nuevamente rápido al final), combinar animaciones, y mucho más. Al trabajar con animaciones, definimos todo el proceso a través de la línea de tiempo, indicando la duración y características de cada uno de los estados intermedios. Estos estados intermedios reciben el nombre de **KeyFrames**.

Lanzando animaciones mediante código

Si la animación no es lanzada mediante un **trigger**, sino mediante código, podemos recurrir a los **Resources** asociados con el elemento. El código fuente para manejar al evento lo escribiremos en C#, y, en lugar de situar el código XAML de la animación dentro de un **EventTrigger**, lo ubicamos dentro de los recursos del elemento. Del mismo modo, identificamos con un nombre a cada animación para referenciarlas después desde el código C#. Por ejemplo, para modificar el proceso anterior de forma que se dispare al pulsar sobre el rectángulo, el código XAML se convertiría en (listado 10):

```
<Canvas x:Name="LayoutRoot" Background="White">
  <Rectangle x:Name="rect" Fill="BlueViolet" Canvas.Top="100" Canvas.Left="10"
    Width="100" Height="100"
    MouseLeftButtonDown="rect_MouseLeftButtonDown">
    <Rectangle.Resources>
      <Storyboard x:Name="Animacion2" AutoReverse="True" >
        <DoubleAnimation Storyboard.TargetName="rect"
          Storyboard.TargetProperty="(Canvas.Left)"
          AutoReverse="True" Duration="0:0:1" To="300" />
      </Storyboard>
    </Rectangle.Resources>
  </Rectangle>
</Canvas>
```

Listado 10

Vinculamos directamente el evento a su manejador en el código XAML, e incluimos toda la animación dentro los recursos de **<Rectangle>**. Ya hemos visto que es la forma en que Visual Studio crea automáticamente un manejador de evento en el que programar el comienzo de la animación (listado 11).

Si el lector prueba el ejemplo anterior, observará un efecto adicional que hemos añadido a propósito para ilustrar mejor los conceptos de *Animation* y *Storyboard*: la ida y vuel-

```
private void rect_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    Animacion2.Begin();
}
```

Listado 11

ta del rectángulo se repite dos veces: una por cada aparición del atributo **AutoReverse="True"**, ya que éste puede aparecer como parte de todo el *Storyboard* (contenga las animaciones que contenga), y también de cada una de las animaciones individuales.

nota

Esto mismo puede hacerse con cualquier otro objeto *UIElement*. El evento correspondiente variará en función de los que tenga definidos el elemento.

Creación de animaciones mediante código

Para crear una animación con código .NET, nos basta con crear un proyecto inicial y realizar todas las operaciones de puesta en marcha de la animación en un método llamado a continuación de **InitializeComponent()**, que, en este caso, hemos llamado **CrearEjecutaAnimation()**. El código sería el siguiente (listado 12).

El efecto es similar al anterior e idéntico funcionalmente. Solo falta añadir la llamada a este método en el manejador de evento que nos interese (o en **InitializeComponent()**, si deseamos que se ejecute al cargar la página).

Líneas de tiempo y KeyFrames

Anticipábamos antes que es posible controlar la forma en que se producen las transiciones de estado, recurriendo al concepto de *KeyFrame*, que establece la duración individual de cada uno de los segmentos en que se puede dividir una animación. Hay dos valores relacionados con el tiempo: la duración (**Duration**), y el momento en que debe de concluir cada segmento a contar desde su inicio (**KeyTime**). También es posible observar gráficamente la forma en que evoluciona el valor de la propiedad en el tiempo. Con ello, tendremos una idea visual del mecanismo de interpolación lineal⁴ utilizado internamente para la construcción de la animación.

El esquema del siguiente ejemplo queda ilustrado en la figura 7.

⁴ En análisis numérico, se denomina interpolación a la construcción de nuevos puntos partiendo del conocimiento de un conjunto discreto de ellos. Es de gran utilidad en el manejo de imágenes digitales, especialmente para el cambio de tamaño, donde existen algoritmos que expresan varias formas de "calcular" los puntos restantes.

```
public void CreaYEjecutaAnimation(object sender, EventArgs e)
{
    // Creamos el rectángulo a animar
    Rectangle Rect1 = new Rectangle();
    Rect1.Width = 180;
    Rect1.Height = 30;
    Rect1.SetValue(Canvas.LeftProperty, 50.0);
    Rect1.SetValue(Canvas.TopProperty, 50.0);

    // .. y sus propiedades
    Color myColor = Color.FromArgb(255, 255, 125, 0);
    SolidColorBrush myBrush = new SolidColorBrush();
    myBrush.Color = myColor;
    Rect1.Fill = myBrush;

    // Añamos el rectángulo al contenedor.
    LayoutRoot.Children.Add(Rect1);

    // establecemos la duración
    Duration duration = new Duration(TimeSpan.FromSeconds(0.5));
    DoubleAnimation Animation1 = new DoubleAnimation();
    Animation1.Duration = duration;
    Animation1.To = 200;

    // Y creamos la animación
    Storyboard sb = new Storyboard();
    sb.Duration = duration;
    sb.Children.Add(Animation1);
    Storyboard.SetTarget(Animation1, Rect1);
    Storyboard.SetTargetProperty(Animation1, new PropertyPath("(Canvas.Left)"));

    // Añadir el Storyboard al recurso, para vincular animación y rectángulo
    LayoutRoot.Resources.Add("Key1", sb);

    sb.Begin(); //Lanzar la animación
}
```

Listado 12

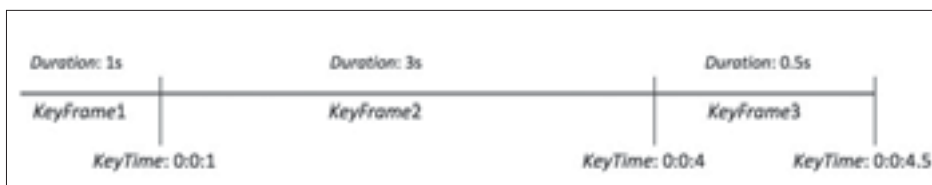


figura 7

La línea de tiempo abarca un total de 4,5 segundos, y está dividida en 3 segmentos de duración variable

Supongamos que en el caso del dibujo se trata de animar la propiedad **Canvas.Left**, al igual que hicimos antes, pero queremos que se realice en 3 fases: en la primera se recorren 100 píxeles en 1 segundo, en la segunda 200 en 3 segundos (va más despacio), y en la tercera 100 píxeles en 0,5 segundos (la más rápida de todas).

Para ello, en el Explorador de proyectos, seleccionamos la opción "*Abrir con Expression Blend*" en el menú contextual del fichero **Page.xaml**, y una vez allí, seleccionamos el control que deseamos animar (de nombre **rect** en el ejemplo). Después, sobre el panel izquierdo, en la sub-ventana "*Objects & Timeline*", creamos la animación —aquí con el nombre **sbMovimiento**—, y para editarla, movemos la línea vertical amarilla hasta el punto final de la duración del primer segmento (hay un punto rojo en la parte superior de la pantalla, indicando que "*Timeline recording is on*", que significa que se están almacenando los cambios en las propiedades para generar después el código XAML correspondiente). Hecho esto, asignamos el nuevo valor que deseamos para la propiedad. Si modificamos más de una propiedad, Blend creará un elemento **Storyboard** por cada una (La figura 8 ilustra esta situación).



figura 8

Editor de líneas de tiempo en Expression Blend mostrando los 3 puntos de segmentación de la animación. A la derecha, sub-ventana de edición de posiciones, mostrando el valor de la propiedad *Left* al final de la animación.

Este proceso lo repetimos para cada segmento.

El resultado es un código algo distinto (listado 13), que utiliza un elemento llamado **DoubleAnimationUsingKeyFrames**, que contiene elementos **SplineDoubleKeyFrame**, cada uno de ellos indicando el tiempo transcurrido para cada **KeyFrame** y el valor de la propiedad a cambiar que debe obtenerse en ese momento. El código de la animación es el siguiente:

```
<UserControl.Resources>
  <Storyboard x:Name="sbMovimiento">
    <DoubleAnimationUsingKeyFrames BeginTime="00:00:00"
      Storyboard.TargetName="rect" Storyboard.TargetProperty="(Canvas.Left)">
      <SplineDoubleKeyFrame KeyTime="00:00:01" Value="100"/>
      <SplineDoubleKeyFrame KeyTime="00:00:04" Value="300"/>
      <SplineDoubleKeyFrame KeyTime="00:00:04.5" Value="400"/>
    </DoubleAnimationUsingKeyFrames>
  </Storyboard>
</UserControl.Resources>
```

Listado 13

Si el lector ejecuta este código verá, que, si bien el desplazamiento se produce en tres tramos y —en cada uno— la velocidad es distinta (se recorre distinta distancia por unidad de tiempo), dentro de cada tramo la velocidad es constante, como no podía ser de otra forma, de acuerdo con la definición. Se ha producido aquí un proceso de interpolación lineal, para cada una de las tres fases. En la figura 9-a vemos el gráfico correspondiente para una interpolación lineal, que se obtiene en Blend pulsando en cada uno de los símbolos indicados por las flechas verdes de la figura 9.

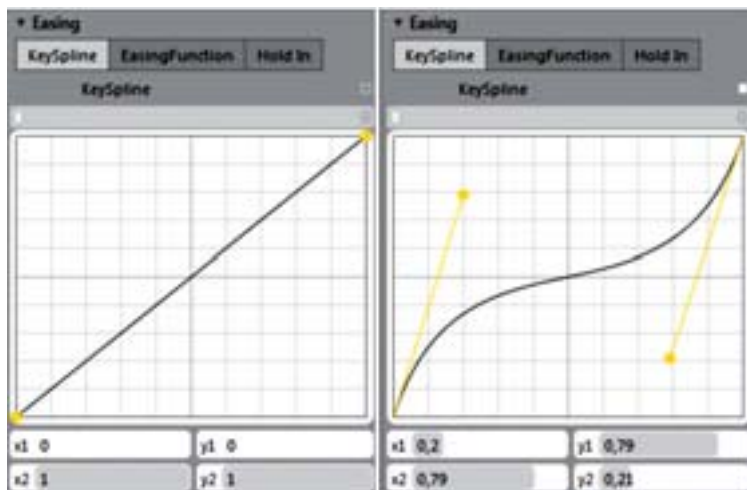


figura 9a y 9b Representación gráfica de la interpolación lineal y curva propia del uso de *KeyFrames*

La curva que describe la interpolación (el cálculo de valores intermedios entre dos datos previamente), es —en realidad— una recta. Eso nos está indicando que ésta se ha realizado en saltos *discretos y uniformes*, sin cambios bruscos dentro de cada fase, y generando, de ese modo, velocidades constantes en el desplazamiento.

Interpolación no-lineal

Por tanto, la cuestión es: ¿se pueden utilizar interpolaciones no lineales que permitan expresar conceptos como la aceleración (entendiendo esto por el cambio brusco entre valores en el camino hacia el valor final)? La respuesta, naturalmente, es que sí. Pero, en este caso, lo que haremos es modificar el trazado para convertirlo en una curva, y de una manera totalmente visual. El eje de las X representa el tiempo necesario para llegar a un valor dado, mientras que los valores intermedios hacia el valor final se representan en el eje de las Y. Por tanto, cada cambio en la curva, provocará resultados distintos.

Así pues, si volvemos al modo de edición y —en la ventana "Easing", que nos describe la curva de interpolación— desplazamos los puntos amarillos hacia otro lugar de la cuadrícula, modificaremos el trazado, creando una curva de interpolación que dará como resultado aceleraciones y/o deceleraciones (el gráfico 9-b muestra el resultado del cambio).

El nuevo patrón obtenido en nuestro ejemplo, es una curva con un punto de inflexión que pasa directamente por el centro. En el primer tramo, se produce una aceleración que va descendiendo mientras se adoptan valores intermedios, volviendo a acelerarse en el tramo final. De hecho, como puede intuir el lector, existe una relación directa entre los valores intermedios obtenidos y los valores de la función representada aquí.

En el código XAML, no se han producido muchos cambios. Lo único que ha cambiado es que el segundo elemento **SplineDoubleKeyFrame**, contiene ahora un subelemento **KeySpline**, donde se definen esas fluctuaciones de acuerdo con la sintaxis siguiente (listado 14):

```
<SplineDoubleKeyFrame KeyTime="00:00:04" Value="300">
  <SplineDoubleKeyFrame.KeySpline>
    <KeySpline ControlPoint1="0.1019,0.899"
      ControlPoint2="0.893999993801117,0.094"/>
  </SplineDoubleKeyFrame.KeySpline>
</SplineDoubleKeyFrame>
```

Se indican 2 puntos de control a partir de los cuales el motor de Silverlight realizará la interpolación de acuerdo con la curva definida. Pueden crearse tantas curvas distintas como permitan las combinaciones posibles donde situar los dos modificadores resaltados con un punto amarillo.

Funciones Easing

De hecho, desde la versión 3 de Silverlight, contamos con algunos de estos mecanismos ya creados y listos para ser añadidos a cualquier objeto **Animation**, produciendo efectos adicionales como el rebote, caída libre, vibración, y muchísimos más.

En la ventana de la figura 10 podemos ver algunas de las funciones **Easing** disponibles desde la ventana de propiedades de una animación.

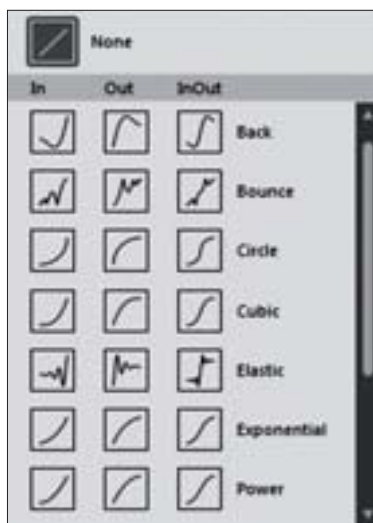


figura 10 Funciones Easing disponibles para las animaciones desde Blend 4

Visual State Manager

La codificación de cada posible estado de un control puede alcanzar grados de complejidad muy altos, y, a menudo, resulta complicado sin una herramienta preparada para ello.

Por otro lado, el modelo propuesto por Silverlight respecto a los controles (llamado *Parts-and-States Model*), aboga por una separación de la presentación visual res-

pecto a la lógica del control, que no tiene por qué variar debido a esto (y viceversa, podemos dejar intacta la parte visual y manipular su lógica como estábamos acostumbrados en ASP.NET).

El modelo Parts-And-States

Trabajando con este modelo, muchos de los controles suministrados con esta versión encapsulan su apariencia en una plantilla administrada por Visual State Manager, que maneja sus 3 partes fundamentales:

- **Parts** (las Partes: los elementos internos constituyentes del control).
- **States & StateGroups** (los Estados y Grupos de estado). El código XAML asociado con las transiciones entre estados para otorgar a cada uno su *personalidad visual*: qué es lo que sucede cuando el control pasa de tener al foco a no tenerlo, a estar pulsado, o deshabilitado, etc. Cada estado tiene su definición, y Visual State Manager se encarga de la transición entre estados.
- **Transitions** (las Transiciones). Cambios visuales, codificados normalmente como recursos, que se vincularán a un cambio de estado.

Partes

Una parte es un elemento con nombre dentro de un control. Es importante para referirnos a él con posterioridad, tanto por código como desde Blend, y también resulta fundamental a la hora de la creación de plantillas de controles personalizados.

Estados y Grupos de Estado

Cada control tiene —visualmente— un conjunto de estados denominados "Estados Comunes", "Estados de Foco" y "Estados de comprobación". Son los siguientes:

- Estados Comunes (*Common States*)
 - Normal** (ninguna acción sobre el elemento)
 - MouseOver** (ratón por encima del área útil del elemento)
 - Pressed** (botón pulsado sobre el elemento)
 - Disabled** (deshabilitado, no responde a eventos de usuario)

- Estados de Foco (*Focus States*)
 - Focused** (con el foco —si es posible adquirirlo—).
 - Unfocused** (sin el foco).
 - FocusedDropDown** (con foco y desplegado, solo para el **ComboBox**).
- Estados de comprobación (*Checked States*). Para controles **CheckBox** y **RadioButton**
 - Checked** (marcado).
 - Unchecked** (desmarcado).
 - Indeterminate** (indeterminado, solo para **CheckBox**)

Esta división en grupos de estado es muy importante en la codificación de la plantilla, porque permite la agrupación de acciones comunes aplicables después, evitando así la multiplicación combinatoria de estados que podría producirse de otra forma.

Como veremos a continuación, tras editar la plantilla de un botón, la ventana de interacciones ("Interaction"), nos muestra todos los estados posibles de un control y cuáles son las acciones vinculadas con las transiciones entre estados. Ahí podemos modificarlas para crear nuestros propios efectos, o crear transiciones entre estados especiales definidos por el usuario.

Deconstruyendo a Button

Para observar cómo están hechos los controles predeterminados, nada mejor que seleccionar uno en la ventana de diseño de Blend y, en su menú contextual, escoger el ítem "Edit Control Parts" -> "Edit a Copy". Veremos que en la ventana superior izquierda del IDE de Blend aparecen definidos el conjunto de estados posibles del control y cuáles son las acciones a tomar en las transiciones entre un estado y otro, como aparece en la figura 11, tras haber hecho esto con un control **Button**.

Al igual que sucedía con las animaciones, cuando marcamos uno de los estados en la ventana, aparece una notificación en la superficie de diseño para avisarnos de que se están grabando las acciones en formato XAML (*State Recording is on*). Podemos hacer pruebas desactivando ese botón y volviendo a activarlo cuando queramos que se genere el código XAML correspondiente.

No obstante, la literatura digital relacionada no suele recoger un aspecto del funcionamiento de VSM como es la creación de estados personalizados, dejando entrever, en ocasiones, que los citados antes son los únicos posibles (o que tienen sentido) y eso dista mucho de ser así.



figura 11

Ventana de interacción para definir transiciones de estado

Utilización de Visual State Manager para la personalización de una interfaz de usuario

Supongamos que queremos crear nuestra propia versión de los botones que incluye Silverlight 4, utilizando Blend y el VSM. Podemos aprovechar nuestro proyecto abierto en Blend con un botón en pantalla, para usar este mismo ejemplo mediante los siguientes pasos:

- Indicamos que queremos crear una plantilla nueva para aplicarla al botón, lo que despoja al control de todo contenido visual añadido, quedándose en un mero **Grid** vacío.
- En ese momento, aparecerán los estados en la ventana **States**, y solo tenemos que asignar a cada uno una imagen distinta (pero de colores diferentes, por ejemplo) para que —automáticamente— al ejecutar, tengamos un botón con apariencia diferente al cambiar de estado.
- Comprobamos desde la Ventana de estados que el cambio se realiza seleccionando cada estado (no es necesaria la ejecución).
- Para el estado **Disabled** nos basta con cambiar la opacidad del gráfico a un 33% aproximadamente.
- Para dar la apariencia de respuesta asociamos un movimiento cuando el usuario pulse el botón: aquí le aplicamos una transformación por rotación. Por ejemplo, 45° cuando se encuentre en su estado **Pressed**. No hay que codificar nada.

- Y todo esto mediante diseño. Dibujamos 3 controles de este tipo y uno de ellos lo deshabilitamos en su ventana de propiedades. El resultado obtenido será algo similar a la figura 12.



figura 12

Botón personalizado con 3 estados distintos manejados por VSM

Obsérvese como se produce la rotación cuando pasamos el cursor por encima del botón inferior. De eso se encarga VSM sin ningún código en C# y optimizando todo el proceso. De la misma forma, al declarar el botón superior derecho como deshabilitado, el gestor de estado se encarga de aplicarle la opacidad disminuida creando el efecto oportuno.

Creación de estados y transiciones propios con VSM

En realidad, hay muchos escenarios donde lo que tiene más sentido es la creación de estados y transiciones propias, aunque no tengan que ver directamente con el foco, el ratón o no sigan el patrón estándar.

Imaginemos la siguiente situación: disponemos de una página en la que aparece una estructura inicial con un rótulo que indica que no nos hemos identificado en el sitio (situación predeterminada). En la parte inferior, un texto explica las ventajas de hacerlo, etc. Y queremos probar el control de esta situación mediante VSM, de forma que las transiciones tengan lugar cuando se dé una situación de negocio (como el hecho de que un usuario se haya identificado o no). En ese caso, los estados no tendrían nada que ver con la interacción visual, sino con la gestión lógica de la página.

Podemos programar una cosa así simplemente disponiendo de 2 alternativas, una de las cuales se encuentra oculta por defecto (por ejemplo, una zona que indica la iden-

tificación y un texto de agradecimiento, en coordenadas negativas, a la izquierda del borde visible). En diseño, (donde veríamos ambos escenarios posibles), tendríamos algo como lo que aparece en la figura 13:



figura 13 Situación inicial de partida. Los dos elementos de la izquierda, están ocultos al haberse definido con una coordenada negativa sobre el eje de las X

Este diseño define por lo tanto, un estado **lógico**, no visual. El usuario puede o no haberse identificado. En este caso, el patrón *Parts-and-States* no tiene sentido, porque solo es aplicable a estados visuales. Lo que hacemos, pues, es —sobre la Ventana de Estados—, definir un nuevo **StateGroup** que tiene que ver con esta lógica, y que llamamos **EstadoRegistro**. Dentro de él añadiremos dos estados posibles: **EstadoInicial** y **Registrado**.

No tocamos el estado inicial que coincidirá con el del diseño. Pero en el estado **Registrado**, (después de que el usuario haya pulsado el botón Entrar), definimos una transformación por desplazamiento, e intercambiamos el valor de la propiedad **x**. Así, los dos objetos **Border** que contienen los rótulos asumen valores opuestos a los que tenían (positivo por negativo, etc.) La consecuencia es que el rótulo del usuario se desplazará a la zona invisible y aparecerá súbitamente el rótulo del saludo. Y lo mismo hacemos con el bloque de texto que muestra la información explicativa. Podemos comprobarlo, como siempre, pulsando en los dos estados y viendo los cambios.

Ahora bien, tenemos que intervenir en el código fuente para llamar al cambio de estado, cuando se haya producido la autenticación. Bastará con programar el botón **Entrar** para que indique a Visual State Manager que el estado ha cambiado. A su vez, en el rótulo que corresponde al usuario registrado, utilizaremos el icono de usuario identificado para volver a la situación inicial. El código C# será similar a esto (listado 15):

```
private void btnEntrar_Click(object sender, RoutedEventArgs e)
{
    VisualStateManager.GoToState(this, "Registrado", true);
}

private void ImgRegistrado_MouseLeftButtonUp(object sender, MouseButtonEventArgs e)
{
    VisualStateManager.GoToState(this, "EstadoInicial", true);
}
```

Listado 15

Si además, modificamos el tiempo de la transición y asignamos **0,3** segundos en lugar del valor **0** predeterminado, obtendremos un cambio dinámico mucho más agradable en la interfaz de usuario, cuando pasemos de un estado a otro (ver el cambio final en la figura 14).

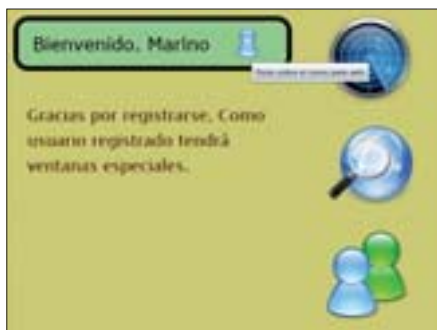


figura 14

Resultado de la transición mostrando una etiqueta flotante

Y hemos añadido una etiqueta flotante para que el usuario sepa que el icono le lleva nuevamente al estado inicial.

El código XAML completo es el siguiente (listado 16).

Construcción de un control personalizado

Con lo visto hasta el momento, ya tenemos muchas características de lo que puede ser un control de usuario para reutilización en otros programas. Vamos a construir uno que

```

<UserControl
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Class="EstadosYTransiciones.Page"
  Width="640" Height="480"
  xmlns:vsm="clr-namespace:System.Windows;assembly=System.Windows">

  <Grid x:Name="LayoutRoot" Background="#FFD3CD6D" >
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="432"/>
      <ColumnDefinition Width="208"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition Height="104.16"/>
      <RowDefinition Height="375.84"/>
    </Grid.RowDefinitions>
    <vsm:VisualStateManager.VisualStateGroups>
      <vsm:VisualStateGroup x:Name="EstadoRegistro">
        <vsm:VisualStateGroup.Transitions>
          <vsm:VisualTransition GeneratedDuration="00:00:00.3000000"/>
        </vsm:VisualStateGroup.Transitions>
        <vsm:VisualState x:Name="EstadoInicial">
          <Storyboard/>
        </vsm:VisualState>
        <vsm:VisualState x:Name="Registrado">
          <Storyboard>
            <DoubleAnimationUsingKeyFrames BeginTime="00:00:00"
              Duration="00:00:00.0010000" Storyboard.TargetName="BordeRotulo2"
              Storyboard.TargetProperty="(UIElement.RenderTransform).
                (TransformGroup.Children)[3].(TranslateTransform.X)">
              <SplineDoubleKeyFrame KeyTime="00:00:00" Value="0"/>
            </DoubleAnimationUsingKeyFrames>
            <DoubleAnimationUsingKeyFrames BeginTime="00:00:00"
              Duration="00:00:00.0010000" Storyboard.TargetName="BordeRotulo1"
              Storyboard.TargetProperty="(UIElement.RenderTransform).
                (TransformGroup.Children)[3].(TranslateTransform.X)">
              <SplineDoubleKeyFrame KeyTime="00:00:00" Value="-500"/>
            </DoubleAnimationUsingKeyFrames>
            <DoubleAnimationUsingKeyFrames BeginTime="00:00:00"
              Duration="00:00:00.0010000" Storyboard.TargetName="textBlock"
              Storyboard.TargetProperty="(UIElement.RenderTransform).
                (TransformGroup.Children)[3].(TranslateTransform.X)">
              <SplineDoubleKeyFrame KeyTime="00:00:00" Value="0"/>
            </DoubleAnimationUsingKeyFrames>
            <DoubleAnimationUsingKeyFrames BeginTime="00:00:00" Duration="00:00:00.0010000"
              Storyboard.TargetName="textBlock1"

```

```

        Storyboard.TargetProperty="(UIElement.RenderTransform).
            (TransformGroup.Children)[3].(TranslateTransform.X)">
        <SplineDoubleKeyFrame KeyTime="00:00:00" Value="-500"/>
    </DoubleAnimationUsingKeyFrames>
</Storyboard>
</vsm:VisualState>
</vsm:VisualStateGroup>
</vsm:VisualStateManager.VisualStateGroups>
<Border HorizontalAlignment="Stretch" Margin="12,12,12,12" BorderThickness="7,7,7,7"
    CornerRadius="18,18,18,18" BorderBrush="#FF000000" x:Name="BordeRotulo1"
    Background="#FFA370B9" RenderTransformOrigin="0.5,0.5">
    <Border.RenderTransform>
        <TransformGroup>
            <ScaleTransform/>
            <SkewTransform/>
            <RotateTransform/>
            <TranslateTransform/>
        </TransformGroup>
    </Border.RenderTransform>
    <StackPanel Orientation="Horizontal" >
        <TextBlock Height="Auto" Width="123" TextWrapping="Wrap" Margin="21,21,21,21"
            HorizontalAlignment="Left">
            <Run FontFamily="Verdana" FontSize="24" FontWeight="Bold"
                Text="Usuario:"/>
        </TextBlock>
        <TextBox HorizontalAlignment="Left" VerticalAlignment="Center" Text="" TextWrapping="Wrap"
            BorderThickness="3,3,3,3" Width="135" Height="Auto" FontFamily="Arial"
            FontSize="24"/>
        <Button Height="35" HorizontalAlignment="Right"
            Margin="12,12,12,12" x:Name="btnEntrar"
            FontFamily="Arial" FontSize="24" Content="Entrar" Click="btnEntrar_Click" />
    </StackPanel>
</Border>
<Image HorizontalAlignment="Stretch" Margin="40,12,40,-35.8400001525879"
    VerticalAlignment="Stretch" Grid.Column="1" Source="Graficos/find.png" Stretch="Fill"/>
<Image Margin="40,63.8400001525879,40,184" Grid.Column="1" Grid.Row="1"
    Source="Graficos/search-web.png" Stretch="Fill"/>
<Image Height="128" Margin="40,0,40,24" VerticalAlignment="Bottom"
    Grid.Column="1" Grid.Row="1" Source="Graficos/kdmconfig.png" Stretch="Fill"/>
<Border HorizontalAlignment="Stretch" Margin="12,12,12,12"
    BorderThickness="7,7,7,7" CornerRadius="18,18,18,18" BorderBrush="#FF000000"
    x:Name="BordeRotulo2" Background="#FF9CC889" RenderTransformOrigin="0.5,0.5">
    <Border.RenderTransform>
        <TransformGroup>
            <ScaleTransform/>
            <SkewTransform/>

```

```

        <RotateTransform/>
        <TranslateTransform X="-450"/>
    </TransformGroup>
</Border.RenderTransform>
<StackPanel Orientation="Horizontal" >
    <TextBlock Height="Auto" Width="312" TextWrapping="Wrap" Margin="21,21,21,21"
        HorizontalAlignment="Left" VerticalAlignment="Center" FontFamily="Verdana"
        FontSize="27" Text="Bienvenido, Marino"/>
    <Image Height="Auto" HorizontalAlignment="Stretch"
        Margin="12,12,122,12" x:Name="ImgRegistrado"
        VerticalAlignment="Center" Width="Auto"
        RenderTransformOrigin="0.5,0.5"
        Source="Graficos/personal.png"
        Stretch="Uniform"
        MouseLeftButtonUp="ImgRegistrado_MouseLeftButtonUp">
        <Image.RenderTransform>
            <TransformGroup>
                <ScaleTransform/>
                <SkewTransform/>
                <RotateTransform/>
                <TranslateTransform X="-50"/>
            </TransformGroup>
        </Image.RenderTransform>
    </Image>
    <ToolTipService.ToolTip>
        <TextBlock>Pulse sobre el icono para salir</TextBlock>
    </ToolTipService.ToolTip>
</StackPanel>
</Border>
<TextBlock Margin="32,35.8400001525879,40,48" Grid.Row="1"
    TextWrapping="Wrap" RenderTransformOrigin="0.5,0.5" x:Name="textBlock1">
    <TextBlock.RenderTransform>
        <TransformGroup>
            <ScaleTransform/>
            <SkewTransform/>
            <RotateTransform/>
            <TranslateTransform/>
        </TransformGroup>
    </TextBlock.RenderTransform>
    <Run FontSize="20" FontWeight="Bold"
        Text="El veloz murciélago hindú comía feliz cardillo y kiwi. "/>
    <LineBreak/>
    <Run FontSize="20" FontWeight="Bold"
        Text="¡¡ Y Ud. también comerá feliz si entra en nuestro sitio y disfruta
```

```

        de todas sus ventajas!!"/>
    <LineBreak/>
    <Run FontSize="20" FontWeight="Bold" Text="Saludos"/>
    <LineBreak/>
    <Run FontSize="20" FontWeight="Bold" Text="El WebMaster"/>
</TextBlock>
<TextBlock Margin="32,35.8400001525879,40,48" TextWrapping="Wrap"
    OpacityMask="#FFCEDF18" RenderTransformOrigin="0.5,0.5"
    Grid.Row="1" FontSize="24" Foreground="#FF3E0404"
    Text="Gracias por registrarse. Como usuario registrado tendrá ventanas especiales."
    x:Name="textBlock">
    <TextBlock.RenderTransform>
        <TransformGroup>
            <ScaleTransform/>
            <SkewTransform/>
            <RotateTransform/>
            <TranslateTransform X="-450"/>
        </TransformGroup>
    </TextBlock.RenderTransform>
</TextBlock>
</Grid>
</UserControl>

```

Listado 16

tenga una apariencia visual muy simple —por razones didácticas—, pero que permita su uso y programación por terceros y que presente alguna funcionalidad adicional, como podríamos necesitar en objetos de negocio. Para ello, tenemos que decidir esa funcionalidad, y —dependiendo de eso— la arquitectura de base. Por ejemplo, no es lo mismo un control que albergue colecciones de objetos que uno de contenido simple, etc.

En general, podemos construir 3 tipos de controles:

- Los que combinan dos o más controles básicos en uno
- Controles que extienden la funcionalidad de uno existente
- Controles totalmente nuevos contruidos desde cero.

Ya que el tercer caso aporta el mayor grado de libertad, y da paso para continuar su extensión con características de los otros dos, hemos optado por construir uno desde cero, manteniendo funcionalidad y presentación en niveles muy sencillos, de forma que pueda entenderse fácilmente. Los pasos a dar serán los siguientes.

Pasos típicos en la creación de un control de usuario

Cuando no basta con la funcionalidad predeterminada, o preferimos escribir un control de usuario desde cero, hay varios aspectos importantes a tener en cuenta y que no son evidentes. El más significativo, es que la definición de la plantilla que definirá la estructura visual del control, debe de ser creada en un fichero de nombre **Generic.xaml** (no vale otro nombre), que ubicaremos en un subdirectorio de nombre **Themes** en la aplicación y ahí concretaremos los estilos dentro de un elemento **ResourceDictionary**. Ese fichero, deberá establecer la opción "Build Action" a **Resource** en lugar de la predeterminada (**Page**), aunque podría funcionar con ésta última según y qué circunstancias.

Los pasos son los siguientes:

- 1) Definir funcionalidad básica y arquitectura (para la jerarquía de herencia).
- 2) Crear una clase que herede de **Control** o **ContentControl**.
- 3) Establecer la IU inicial del control, con valores predeterminados, dependiendo de (1) y (2).
- 4) Definir la plantilla del control. El mecanismo consiste en crear un fichero de nombre **Generic.xaml** en el subdirectorio **themes** de la aplicación, e incluir allí un elemento **ResourceDictionary**, donde establezcamos el aspecto inicial.
- 5) En el constructor de la clase que representa al control, asignamos la propiedad **DefaultStyleKey** al tipo de elemento que estamos creando:

```
this.DefaultStyleKey = typeof(Nombre_DeL_Control);
```

Si es preciso, por ejemplo, por incluir algún control predeterminado, añadimos las **DependencyProperties** requeridas para almacenar valores del control.

- 6) Añadir **TemplateBindings** en las propiedades definidas en el **ResourceDictionary** para permitir la personalización de otros usuarios (esto puede hacerse como continuación del paso 4, igualmente).

- 7) En caso de necesitar sustituir al contenido heredado, crear su intérprete de contenidos (**ContentPresenter**). Dependerá de las acciones a realizar por el control.
- 8) Añadir los eventos necesarios para gestionar la lógica interna (visual o de negocio).
- 9) (Opcionalmente) Añadir los estados visuales mediante Visual State Manager y retocar el punto 8 si es preciso.
- 10) Probar y publicar.

El control **TextoDNI**

Vamos a construir un control muy sencillo, llamado **TextoDNI**, partiendo de cero, al que añadiremos la capacidad de validar la presencia de números y calcular la letra del DNI automáticamente al mostrarse en pantalla.

nota

Un control para producción, llevaría mucho más código que el que vamos a incluir aquí. Por propósitos didácticos, hemos mantenido el código fuente lo mínimo imprescindible para que el control funcione y pueda ser probado.

Seguimos paso a paso las indicaciones apuntadas más arriba, personalizándolas para este control.

Paso 1) Queremos que sea capaz de validar —al cargarse o al perder el foco, tras haber sido modificado— que los contenidos sean de tipo numérico y calcular la letra asociada a ese número suponiendo que se trata de un DNI, generando un mensaje estándar de error en caso contrario.

Paso 2) Crear una aplicación Silverlight con sitio web de pruebas, y añadir una librería de clases Silverlight (que llamamos **CajaDNI**) para albergar nuestro control. Éste, deberá heredar de **Control** o **ContentControl** (al final nos decantamos por **ContentControl**, por si el usuario decide modificar a su vez, toda la propiedad **Content**). Cambiamos el nombre de la clase a **TextoDNI** y escribimos un constructor vacío de momento. Compilamos, para poder hacer referencia a la DLL del proyecto desde el código XAML.

Paso 2-b) Antes de definir la interfaz, creamos un contexto vacío. Esto es, creamos la arquitectura y las referencias.

- Compilamos la librería **CajaDNI**, para generar la DLL
- Hacemos referencia a ella desde el control Silverlight (**Page.xaml**). El código añadido podría ser algo como esto:

```
xmlns:DNI="clr-namespace:CajaDNI;assembly=CajaDNI"
```

- En la aplicación Silverlight, añadimos el control, que no tiene representación gráfica, —pero tendrá que ser reconocido por el editor— con una línea de este tipo: **<DNI:TextoDNI />**.
- Generamos la solución.

Paso 3) Definimos un estilo para el control, que, para propósitos de prueba, puede residir en el propio control, dentro de una etiqueta **<Style>**. Establecemos su **TargetType**. Deberíamos de ver el aspecto del control en pantalla y el contenido si añadimos un elemento **ContentPresenter**. El código podría ser algo así (listado 17):

```
<DNI:TextoDNI Content="123123123">
  <DNI:TextoDNI.Style>
    <Style TargetType="DNI:TextoDNI">
      ...Definiciones del estilo...incluyendo una para Content
    </Style>
  </DNI:TextoDNI.Style>
</DNI:TextoDNI Content="123123123">
```

Listado 17

Paso 4) Trasladamos el código de la plantilla a un nuevo fichero **Generic.xaml**, que ubicamos en un subdirectorio de la librería de clases **CajaDNI** llamado **Themes**. Eliminamos el estilo de la interfaz de usuario, dejándolo como estaba al principio. Comprobamos que funciona exactamente igual con la plantilla de **Generic.xaml** (recuérdese la propiedad **Build Action** asignada a **Resource**).

El código de **Generic.xaml** podría ser algo como esto (listado 18).

Paso 5-a) En la clase **TextoDNI**, establecemos su asignación de plantilla mediante la propiedad **DefaultStyleKey** (ver sintaxis en punto 5 de los pasos a dar). Hemos usado **TemplateBindings** para las propiedades que queremos que sean modificables, por ejemplo,

```

<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:DNI="clr-namespace:CajaDNI;assembly=CajaDNI">
  <Style TargetType="DNI:TextoDNI">
    <Setter Property="Width" Value="180" />
    <Setter Property="Height" Value="35" />
    <Setter Property="Background" Value="Lavender" />
    <Setter Property="FontSize" Value="14" />
    <Setter Property="Margin" Value="10" />
    <Setter Property="Template">
      <Setter.Value>
        <ControlTemplate TargetType="DNI:TextoDNI">
          <Grid x:Name="Contenedor">
            <Rectangle Width="{TemplateBinding Width}"
              Height="{TemplateBinding Height}"
              Fill="{TemplateBinding Background}"
              StrokeThickness="3"
              Stroke="Purple" RadiusX="12" RadiusY="12" />
            <ContentPresenter Content="{TemplateBinding Content}"
              HorizontalAlignment="Center"
              VerticalAlignment="Center"/>
          </Grid>
        </ControlTemplate>
      </Setter.Value>
    </Setter>
  </Style>
</ResourceDictionary>

```

Listado 18

el fondo del rectángulo, la altura y la anchura. Como no nos estamos basando en ningún control anterior, el contenido estará vinculado al elemento **ContentPresenter**, cuya propiedad **Content**, también asignamos a un **TemplateBinding**.

Paso 5-b) Aquí no es necesario, pero si el usuario probara la construcción de un control basándose en otro ya terminado, como un **TextBox**, para que el **TemplateBinding** sobre la propiedad **Text** del control **TextBox** sea reconocido por la clase **TextoDNI**, podemos definir en ésta una propiedad **Text** como **DependencyProperty**, con la idea de mantener un mecanismo de almacenamiento del valor de la propiedad interna del **TextBox** en el control que lo encapsula. El código para la definición de esta propiedad en la clase **TextoDNI** sería el siguiente (listado 19).

```
private static readonly DependencyProperty TextProperty =
    DependencyProperty.Register("Text", typeof(string), typeof(CajaTextoDNI), null);

public string Text
{
    get { return (string)GetValue(TextProperty); }
    set { SetValue(TextProperty, value); }
}
```

Listado 19

A partir de este momento, se podría asignar el valor de la propiedad **Text** de **TexttoDNI** desde el código XAML igual que otra propiedad cualquiera que ofrezca el sistema de Intellisense.

```
using System;
using System.Text.RegularExpressions;
using System.Windows;
using System.Windows.Controls;

namespace CajaDNI
{
    public class TextoDNI : ContentControl
    {
        string Letras = "TRWAGMYFPDXBNJZSQVHLCKET";
        Regex reNum = new Regex(@"^\d+$");

        public TextoDNI()
        {
            this.DefaultStyleKey = typeof(CajaDNI.TextoDNI);
            this.Loaded += new RoutedEventHandler(TextoDNI_Loaded);
        }

        void TextoDNI_Loaded(object sender, RoutedEventArgs e)
        {
            string Texto = this.Content.ToString();
            bool esNumero = reNum.Match(Texto).Success;
            if (Texto != "" && esNumero)
            {
                this.Content += " - " + Letras[Int32.Parse(Texto) % 23];
            }
            else { this.Content = "Error en el DNI"; }
        }
    }
}
```

Listado 20

Paso 6) Para este ejemplo, ya hemos añadido antes los **TemplateBindings** de las propiedades personalizables.

Paso 7) También hemos definido ya un elemento **ContentPresenter**, así que no se necesita hacer nada.

Paso 8) Programación de la validación y el cálculo de la letra del DNI. Para ello, creamos el manejador del evento **Loaded**, que realizará la validación numérica y el cálculo nada más cargarse el control, justo antes de mostrarlo en pantalla (el algoritmo de cálculo es trivial y el lector puede comprobarlo en el código siguiente, que representa el estado final de la clase –listado 20–).

Y eso es todo. Hemos preferido mantener el código de la clase lo más compacto y reducido posible para mostrar la funcionalidad y la estructura sin ningún adorno, ni más comprobaciones que la que hacemos utilizando expresiones regulares, que acredita que el valor introducido solo contiene números. El cálculo de la letra del DNI, es muy sencillo, como puede verse.

Paso 9) Compilar, probar, etc. Como prueba final, podemos escribir un contenido de **Page.xaml** para tener un par de controles **TextoDNI** independientes, con validación y cálculo.

Y este es el código de pruebas... (listado 21).

```
<UserControl x:Class="ctlDNI_RC0.Page"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:DNI="clr-namespace:CajaDNI;assembly=CajaDNI"
    Width="300" Height="150">
    <Grid x:Name="LayoutRoot" Background="Beige">
        <StackPanel Orientation="Vertical" Grid.Row="0" VerticalAlignment="Center">
            <DNI:TextoDNI Content="87654321" Width="150" />
            <DNI:TextoDNI Content="12345678" Background="Gainsboro" />
        </StackPanel>
    </Grid>
</UserControl>
```

Listado 21

Para obtener una salida como la de la figura 15.

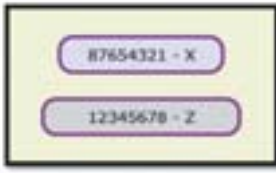


figura 15

El control TextoDNI en funcionamiento

A partir de aquí son muchas las modificaciones que podríamos hacer, tanto en la clase (control de otros comportamientos, más propiedades de dependencia, etc.), como en la plantilla de **Generic.xaml**: por ejemplo, añadiendo estilos visuales para el control de los distintos estados del control, etc.

Conclusión

Silverlight va mucho más allá de la mera representación estática de elementos vectoriales. El sistema de animaciones, permite infinitas posibilidades para la construcción de interfaces dinámicas y la generación de elementos definidos por el usuario. El próximo capítulo nos centraremos en la interacción con el entorno.

Interacción con el entorno

Las opciones de interacción

Silverlight ha ido potenciando sus capacidades de interacción con el contexto de ejecución en muchos aspectos: tanto en la comunicación de aplicaciones individuales entre sí, como en el acceso a recursos del navegador y del sistema operativo. Un conjunto significativo de mejoras que llega hasta el punto de poder ejecutar las aplicaciones Silverlight fuera del navegador.

Podemos dividir este tipo de interacciones en varias categorías.

- Navegación entre páginas de una aplicación.
- Acceso a recursos del sistema.
- Interacción dúplex (en los dos sentidos) con el HTML DOM.
- Interacciones con otras aplicaciones Silverlight.
- Ejecución directa fuera del navegador (*Out-of-Browser*, o interacción con el sistema).
- Acceso a otros recursos (Cámara de vídeo, captura de audio, etc.)

Navegación entre páginas de una aplicación

El lector habrá notado que una de las plantillas predeterminadas de Silverlight 4, se denomina "Silverlight 4 Navigation Application". Si creamos un proyecto de este tipo, y con un par de cambios en los rótulos que designan el nombre de la aplicación y los elementos de menú, tendremos, tras la ejecución inicial, una ventana como la de la figura 1.

Por un lado, la aplicación define unos cuantos elementos **Page** predeterminados en la carpeta **Views**, incluyendo la página inicial (**Home.xaml**), la página "Acerca de..." (**About.xaml**) y una página de error para mostrar en caso necesario (**ErrorWindow.xaml**).



figura 1 Aplicación de navegación. Los elementos de menú se mantienen sincronizados con la página activa

Además, notaremos la presencia de la clase **Styles**, dentro de la carpeta "Assets", donde se definen los estilos predeterminados que se utilizan en el menú y el resto de la aplicación. La clase principal (**MainPage.xaml**), sigue siendo del tipo **UserControl**.

Un vistazo al código XAML nos revela los funcionamientos básicos de la navegación. Los elementos de menú son objetos **HyperLink** que apuntan a la página correspondiente en su propiedad **NavigateUri**. Para ello se utiliza la sintaxis que vemos en el listado 1.

```
<HyperlinkButton x:Name="Link1" Style="{StaticResource LinkStyle}"
    NavigateUri="/Home" TargetName="ContentFrame" Content="Inicio"/>
```

Listado 1

El mecanismo que se encuentra detrás del sistema de navegación se basa en la clase **NavigationService**. Esta clase posee varios métodos fundamentales, que podemos ver en la tabla 1.

NavigationService suministra, además, propiedades y eventos para hacer posible el mecanismo de la navegación. Por otro lado, la clase *Frame* carga los elementos utilizando URI's y soporta características como **URIMapping** (correspondencia entre las URI y las ubicaciones). Y tanto páginas como **Frames** disponen de mecanismo de *caching* para optimizar el tiempo de carga.

El otro elemento fundamental implicado en la navegación es la clase **Page**. Como puede verse en el código fuente XAML de un ejemplo de este tipo, el elemento **User-**

Método	Acción	Detalles
GoBack	Navega a la entrada anterior del historial	Lanza una excepción si no existe dicha entrada
GoForward	Navega a la siguiente entrada del historial	Lanza una excepción si no existe dicha entrada
Navigate	Navega a una URI arbitraria	
Refresh	Recarga la página actual	Solo funciona si se implementa INavigationContentLoader , para la propiedad ContentLoader del Frame
StopLoading	Cancela cualquier operación asíncrona que no haya sido procesada.	

Tabla 1. Opciones principales de la clase *NavigationService*

Control es sustituido por una instancia de esta clase. **Page** suministra, además, eventos y propiedades de navegación, además del *caching* antes citado.

La tabla 2 muestra los eventos más importantes que aporta esta clase respecto a la navegación, implementados como funciones virtuales, para evitar la necesidad de la asignación del manejador correspondiente:

Evento	Acción	Detalles
OnFragmentNavigation	Llamado cuando se navega a una parte de una aplicación Silverlight.	Es similar a la navegación a elementos anchor de HTML. Por ejemplo: /Views/Cientes.xaml#Item3
OnNavigatedFrom	Sucede cuando la página ya no es la página activa	Se suele utilizar para operaciones de limpieza final.
OnNavigatingFrom	Llamado justo antes de que la página se cambie por otra.	EventArgs permite cancelar el proceso.
OnNavigatedTo	Se llama cuando la página se convierte en la página actual en el Frame activo.	Suele usarse en forma similar al evento Loaded en situaciones sin navegación.

Tabla 2. Eventos principales de la clase *Page*

NavigationService se implementa como una propiedad de la clase **Page**. **NavigationService** aporta propiedades útiles para la navegación que se relacionan directamente con el estado de ésta, como **CanGoBack**, **CanGoForward** (ambas tipo **Boolean**), **Source** (que indica la página a mostrar) y **CurrentSource**, que almacena la página que se está mostrando en un momento dado.

En caso de implementar nuestro propio sistema de navegación mediante código, lo correcto sería utilizar comprobaciones de esas propiedades, como se muestra en el prototipo de código que aparece en el listado 2, que asume la existencia de dos botones **NavigateBack** y **NavigateForward**:

```
private void NavigateBack_Click(object sender, RoutedEventArgs e)
{
    if (NavigationService.CanGoBack)
        NavigationService.GoBack();
}
private void NavigateForward_Click(object sender, RoutedEventArgs e)
{
    if (NavigationService.CanGoForward)
        NavigationService.GoForward();
}
```

Listado 2

Por su parte, la clase **Frame**, permite la definición de los elementos de navegación a través de las propiedades **URIMappings** y **MapURI**. La primera, podemos verla en la clase principal del ejemplo genérico y es una colección que almacena todos enlaces navegables vinculándolos al nombre de la página correspondiente, como vemos en el listado 3:

```
<navigation:Frame x:Name="ContentFrame"
    Style="{StaticResource ContentFrameStyle}"
    Source="/Home" Navigated="ContentFrame_Navigated"
    NavigationFailed="ContentFrame_NavigationFailed">
    <navigation:Frame.UriMapper>
        <uriMapper:UriMapper>
            <uriMapper:UriMapping Uri="" MappedUri="/Views/Home.xaml"/>
            <uriMapper:UriMapping Uri="{pageName}"
                MappedUri="/Views/{pageName}.xaml"/>
        </uriMapper:UriMapper>
    </navigation:Frame.UriMapper>
</navigation:Frame>
```

Listado 3

Como vemos, excepto en el caso de la página inicial (**Home.xaml**), la sintaxis permite establecer un enlace entre el nombre de la clase (**pageName**) y la ubicación relativa en la subcarpeta **Views**.

Utilización de ventanas modales

Dentro del contexto de una aplicación es muy habitual que necesitemos utilizar ventanas modales, que no permiten seguir utilizando la aplicación hasta que el usuario no cierra la ventana. Silverlight proporciona la clase **ChildWindow**, que proporciona esa funcionalidad.

Un caso clásico de uso de ventanas modales es la autenticación del usuario. En nuestro ejemplo, nos bastaría con añadir un nuevo elemento del tipo *Child Window* como vemos en la figura 2:

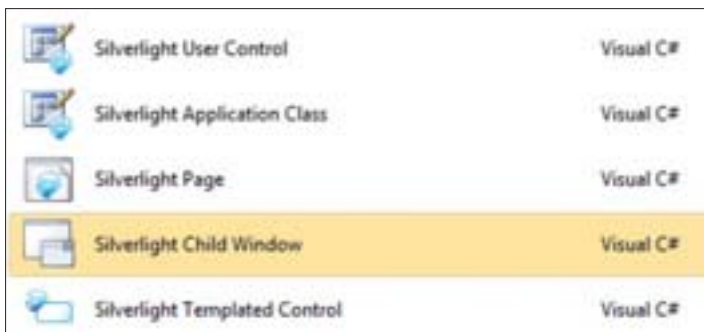


figura 2

Selección de un elemento *Child Window* para crear la ventana de Login de la aplicación

Hemos llamado a este nuevo elemento **VentanaLogin**, y hemos añadido un par de etiquetas con sus correspondientes cajas de texto, para almacenar la entrada del usuario.

Obsérvese cómo el código XAML indica que se trata de una clase de este tipo:

```
<controls:ChildWindow
```

ChildWindow, pertenece al ensamblado **System.Windows.Controls**, y los dos botones predeterminados ("OK" y "Cancel"), simplemente, asignan a la propiedad **DialogResult**

el valor **boolean** que distingue el botón con el que el usuario cierra la ventana (listado 4).

```
private void OKButton_Click(object sender, RoutedEventArgs e)
{
    this.DialogResult = true;
}

private void CancelButton_Click(object sender, RoutedEventArgs e)
{
    this.DialogResult = false;
}
```

Listado 4

Naturalmente, aquí situaríamos la lógica de la gestión de credenciales de la aplicación. En cualquier caso, la ventana se cerrará al pulsar cualquiera de ellos.

Para simplificar la puesta en marcha y, de paso, simular algunas situaciones en las que la página lanza la petición de credenciales al inicio, nos bastaría con añadir el siguiente código en la página **Home**, enlazado al evento **OnNavigatedTo**:

```
// Executes when the user navigates to this page.
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    VentanaLogin vl = new VentanaLogin();
    vl.Show();
}
```

Listado 5



figura 3 Ventana de Login en primer plano al iniciarse la aplicación

En tiempo de ejecución la pantalla de inicio será mostrada con un color más apagado que indica que el contexto que se muestra está deshabilitado, colocándose la ventana de inicio en primer plano y en el centro de la pantalla (ver figura 3).

Acceso a recursos del sistema

A medida que evoluciona la plataforma, el acceso (controlado) a recursos del sistema aumenta. Es el caso de las cajas de diálogo de abrir y guardar fichero, el acceso al portapapeles, el control del botón derecho del ratón, la gestión de videocámaras, la posibilidad de enviar contenidos a la impresora, etc.

Cajas de diálogo *OpenFileDialog* y *SaveFileDialog*

La primera, nos da acceso al sistema de ficheros local, si bien con ciertas restricciones (al igual que la segunda). En ambos casos, el proceso debe hacerse desde código, y no puede definirse en XAML.

Por lo demás, y de forma parecida a como podemos hacer con WPF, se puede configurar la clase **OpenFileDialog**, de manera que se ajuste más a las necesidades de la aplicación. En el ejemplo, implementaremos una llamada que permita seleccionar archivos con formato texto (**.txt**) o XML (**.xml**), y la asignaremos a un botón de la página de inicio de nombre **btnAbrirArchivo**.

El código necesario para la llamada sería algo similar a lo que vemos en el listado 6. Lo que produce una salida como la de la figura 4.

```
private void btnAbrirArchivo_Click(object sender, RoutedEventArgs e)
{
    OpenFileDialog openFileDialog = new OpenFileDialog();
    openFileDialog.Filter = "Ficheros de texto (*.txt)|*.txt|Ficheros Xml (*.xml)|*.xml";
    bool? FicheroSeleccionado = openFileDialog.ShowDialog();
}
```

Listado 6



figura 4 Caja de diálogo Abrir Fichero lanzada desde la página inicial

En el caso de tratarse de la caja **SaveFileDialog**, solo se puede seleccionar un fichero (a diferencia de la anterior que admite la propiedad **MultiSelect**), y la propiedad **SafeFileName** permite el acceso al nombre del fichero.

Continuando con el ejemplo, antes de acceder al fichero, deberíamos comprobar la propiedad **FicheroSeleccionado**, y utilizar un objeto **StreamReader** para acceder al contenido y asignarlo a algún elemento de la interfaz de usuario, como un **TextBlock** de nombre **txtContenido**, como vemos en el listado 7.

```
private void btnAbrirArchivo_Click(object sender, RoutedEventArgs e)
{
    OpenFileDialog openFileDialog = new OpenFileDialog();
    openFileDialog.Filter = "Ficheros de texto (*.txt)|*.txt|Ficheros Xml (*.xml)|*.xml";
    bool? FicheroSeleccionado = openFileDialog.ShowDialog();

    if (FicheroSeleccionado == true)
    {
        FileInfo fileInfo = openFileDialog.File;
        StreamReader reader = fileInfo.OpenText();
        txtContenido.Text = reader.ReadToEnd();
        reader.Close();
    }
}
```

Listado 7

Acceso al portapapeles

Otra de las opciones de acceso a recursos del sistema. Esta soportada por la clase **Clipboard**, que pertenece al espacio de nombres **System.Windows**, y como vemos en la figura 5, solamente soporta 3 métodos básicos de acceso. Todos ellos se refieren a texto, y no a gráficos, que, de momento, no están soportados.



figura 5

los 3 métodos de la clase *Clipboard*, tal y como aparecen en el Explorador de Objetos

Por lo demás, y como puede suponer el lector la asignación de texto al portapapeles a partir de un elemento de la interfaz de usuario como un **TextBlock** tendría la forma:

```
Clipboard.SetText = txtContenido.Text;
```

Y para recuperarlo, comprobaríamos primero su existencia antes de leer, mediante este par de instrucciones:

```
if (Clipboard.ContainsText) txtContenido.Text = Clipboard.GetText();
```

O cualquier forma equivalente.

Menús contextuales

El Silverlight Toolkit, incluye un elemento llamado **ContextMenu** que se ejecuta automáticamente al pulsar botón derecho sobre el elemento de texto que se incluya en el mismo contenedor que lo aloja (en código XAML).

Un **ContextMenu**, puede incluir elementos **MenuItem**, que responden a eventos y pueden, por tanto, ser tratados de forma individual o colectiva. En el listado 8, implementamos un menú contextual en lenguaje XAML, y asignamos un único manejador de evento para 3 posibles opciones (copiar, pegar e imprimir). En el código C#, se hace el tratamiento correspondiente, controlando —a partir de la propiedad

OriginalSource del argumento **e**—, cuál es el menú que origino la pulsación, para responder adecuadamente.

```
<TextBox x:Name="txtTexto" Text="Seleccione texto y pulse botón derecho" />

<toolkit:ContextMenuService.ContextMenu>
  <toolkit:ContextMenu Height="100" Name="contextMenu1" Width="200">
    <toolkit:MenuItem Height="30" Name="menuCopiar" Width="190"
      Header="Copiar"
      Click="menu_Click"/>
    <toolkit:MenuItem Height="30" Name="menuPegar" Width="190"
      Header="Pegar"
      Click="menu_Click"/>
    <toolkit:MenuItem Height="30" Name="menuImprimir" Width="190"
      Header="Imprimir"
      Click="menu_Click"/>
  </toolkit:ContextMenu>
</toolkit:ContextMenuService.ContextMenu>

//En el código C#:

public override void OnMouseRightButtonDown(object sender,
    MouseButtonEventArgs e)
{
    e.Handled = true;
}

public override void OnMouseRightButtonUp(object sender, MouseButtonEventArgs e)
{
    // En el caso de que queramos controlar la posición del menú.
    Point coordenadas = e.GetPosition(this);
    MostrarMenu(coordenadas);
    e.Handled = true;
}

private void menu_Click(object sender, RoutedEventArgs e)
{
    MenuItem menuItem = (MenuItem)sender;
    switch (menuItem.Header.ToString())
    {
        case "Copiar":
            Clipboard.SetText(txtTexto.SelectedText);
            txtTexto.Focus();
            break;
        case "Pegar":
            txtTexto.Text += Clipboard.GetText();
            txtTexto.Focus();
            break;
    }
}
```

```

    case "Imprimir":
        //código para imprimir el texto
        break;

    default:
        break;

}
}

```

Listado 8

Además, manejamos el evento **MouseRightButtonDown** a nivel de página para evitar el comportamiento predeterminado que permite abrir la configuración del complemento Silverlight y otras opciones asociadas, y el evento **MouseRightButtonUp** para afinar la presentación del menú contextual detectando el punto de la pantalla donde se pulsó el botón derecho, y ubicar la salida a partir de ese punto.

Igualmente, los eventos propios del objeto **ContextMenu**, manejables mediante las funciones virtuales **OnLoaded**, **OnClose** y **OnOpened**, nos permitirían hacer un ajuste más fino del comportamiento del menú.

Además, como la propiedad **TabNavigation** está asignada de forma predeterminada al valor "**Cycle**", el menú volverá nuevamente al principio cuando —utilizando el teclado— lleguemos al último ítem y viceversa.

La salida final es la que vemos en la figura 6, con el menú contextual desplegado.

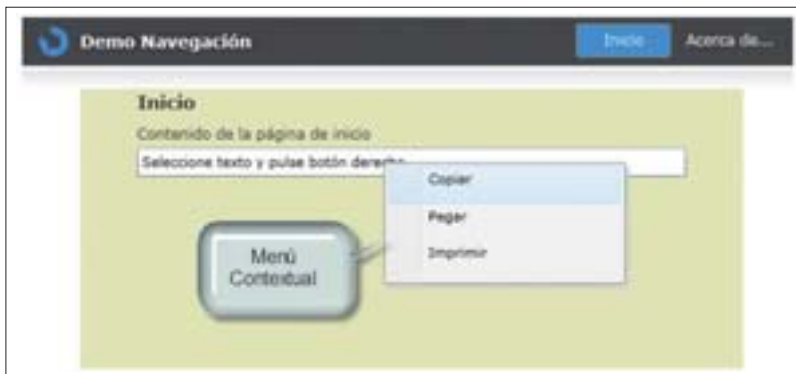


figura 6

Salida en el navegador del código correspondiente al listado 8

Manejo de la impresora

Otra de las novedades aparecidas en la versión 4.0 del complemento, se refiere a la posibilidad de manejar la impresora mediante programación. El objeto principal relacionado con esta nueva capacidad es **PrintDocument**. La forma de controlar el proceso de impresión es mediante los métodos y eventos de esa clase, donde podemos asignar el contenido a imprimir, averiguar las características del área de impresión, etc.

Como puede ver el lector si reproduce el ejemplo, el objeto **PrintDocument** dispone de un método **Print** (que solicita una cadena con un nombre para el documento a imprimir, a efectos de identificación), y también un evento **PrintPage**, donde se asigna el contenido que se quiere imprimir al objeto **PageVisual** de los argumentos de evento.

En una primera aproximación al problema, hemos construido una aplicación Silverlight estándar, y hemos añadido una interfaz de usuario básica con algunos controles para ser rellenados en tiempo de ejecución. Esos controles están insertados dentro del elemento **LayoutRoot**, y son 3 elementos **Label** y 3 **TextBox** donde el usuario introducirá la información.

Si nuestro propósito inicial es, simplemente, imprimir el formulario que hemos definido, conocemos a priori el área de impresión (el elemento **LayoutRoot**) por lo que resulta muy sencillo mandar ese contenido a la impresora, definiendo un objeto **PrintDocument** y un manejador anónimo para su evento **PrintPage**, como podemos ver en el listado 9.

```
private void btnImprimir_Click(object sender, RoutedEventArgs e)
{
    PrintDocument pd = new PrintDocument();
    pd.PrintPage += (s, argImpresion) =>
    {
        argImpresion.PageVisual = LayoutRoot;

    };
    pd.Print("Documento");
}
```

Listado 9

Observe el lector, que tenemos que renombrar la variable que representa a los argumentos de evento (**argImpresion**), ya que nos encontramos dentro un manejador de evento, y el uso de la *e* produciría una coincidencia.

En tiempo de ejecución, si no queremos malgastar papel, podemos configurar la salida hacia un archivo, de forma que nos muestre exactamente cómo quedaría en im-

presión, como hemos hecho aquí, generando el archivo **Documento.xps**. Para ello, seleccionamos como impresora la marcada como "Microsoft XPS Document Writer", y —opcionalmente— marcamos la salida "Imprimir a Archivo" en la misma caja de diálogo de selección de impresora.

La llamada a **Print**, provocará la aparición de esa caja de diálogo, por lo que siempre podemos configurar la salida a nuestro gusto. La figuras 7 y 8 muestran, respectivamente, la entrada de datos del programa y su posterior vista del archivo impreso (abierto por el visor de documentos XPS):

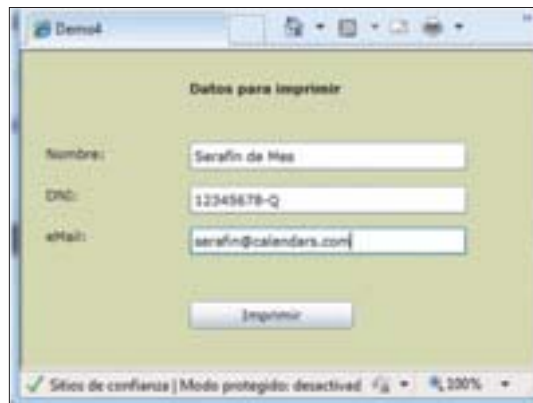


figura 7 Interfaz de usuario del código XAML anterior para ser mandado a impresión



figura 8 Visor de documentos XPS mostrando la salida generada

Como podemos ver, la salida es idéntica, y lo mismo sucede si existe algún elemento gráfico vectorial.

Interacción dúplex (en los dos sentidos) con el HTML DOM

Otra de las opciones interesantes para las aplicaciones Silverlight, es la capacidad de interactuar con el código HTML de la página que lo aloja. Esto puede suceder en ambos sentidos: que el código .NET de una aplicación Silverlight se comunique con el código fuente HTML y al revés, que, mediante código en JavaScript, podamos acceder a elementos de la aplicación Silverlight.

```
using System.IO;
using System.Runtime.Serialization.Json;
using System.Runtime.Serialization;

namespace SeriacionJSON
{
    public class Mensaje
    {
        public string Usuario { get; set; }
        public string Hora_Com { get; set; }
        public string CuerpoMensaje { get; set; }

        /// <summary>
        /// Método que serializa cualquier objeto en JSON
        /// </summary>
        public string SeriacionJSON(object obj)
        {
            using (MemoryStream ms = new MemoryStream())
            {
                DataContractJsonSerializer sj =
                    new DataContractJsonSerializer( obj.GetType() );
                sj.WriteObject(ms, obj);
                ms.Position = 0;
                using (StreamReader reader = new StreamReader(ms))
                {
                    return reader.ReadToEnd();
                }
            }
        }
    }
}
```

En algunos contextos, este mecanismo se denomina *Browser Bridge*. Para poder utilizarlo en un ejemplo que muestre el funcionamiento de ambas opciones, hemos creado una nueva aplicación Silverlight, y hemos añadido una clase Mensaje para que represente los datos, con 3 propiedades públicas, como vemos en el listado 10.

En el código de la aplicación, comenzamos por añadir una referencia al espacio de nombres **System.Windows.Browser**. Con ello, obtenemos acceso a los valores específicos del navegador que aloje la página ASPX/HTML que contenga el complemento Silverlight. Dentro de este espacio de nombres, el objeto **HtmlPage** dispone de la propiedad estática **BrowserInformation**, que contiene información del navegador, accesible directamente por nuestro código. Por ejemplo, si queremos mostrar información relativa al navegador desde nuestro control Silverlight, podemos crear un elemento **TextBlock**, y poblarlo con la información adecuada simplemente usando el código fuente del listado 11:

```
// Recupera información propia del navegador que aloja
// la página contenedora del complemento Silverlight
BrowserInformation b = HtmlPage.BrowserInformation;
BloqueTexto.Text = "Nombre: " + b.Name;
BloqueTexto.Text += "\nVersión del navegador: " +
    b.BrowserVersion.ToString();
BloqueTexto.Text += "\nPlataforma: " + b.Platform;
BloqueTexto.Text += "\nCookies HAbilitadas: " + b.CookiesEnabled;
BloqueTexto.Text += "\nAgente (de usuario): " + b.UserAgent;
```

Listado 11

Ahora, imaginemos que queremos llamar a una función de JavaScript desde Silverlight. Para empezar, puede ser una función muy sencilla que reciba una cadena a mostrar y lo haga mediante una llamada al método **alert()**. Lógicamente, esto implica a dos funciones, cada una escrita en un lenguaje, como vemos en el código del listado 12.

La forma de efectuar la llamada es utilizando **HtmlPage** nuevamente, pero, esta vez, su propiedad estática **Window**, que posee un método **Invoke** que busca dentro del código JavaScript de su página una función con el nombre que se le pasa como primer argumento, y le pasa la lista de argumentos que siguen a este. (El código asume que hemos dispuesto un control **TextBox** en la parte de Silverlight con el nombre **CajaTexto**).

De la misma forma, es posible manipular los elementos HTML de la página contenedora. Para ello, debemos acceder al documento HTML-DOM generado por el navegador y accesible por la clase **HtmlPage**, en su propiedad **Document**. Una vez obten-

```
// Código JavaScript de la página de prueba (.ASPX)
<script type="text/javascript" language="javascript">
// Recibe llamadas desde Silverlight
// para mostrarlas con un alert()
function Mostrar(cad) {
    alert(cad);
}
</script>

// Código de la llamada desde un botón en Silverlight
private void Boton_Click(object sender, RoutedEventArgs e)
{
    // Llama al método JavaScript Mostrar() y le pasa el texto
    // del TextBox.
    HtmlPage.Window.Invoke("Mostrar", CajaTexto.Text);
}
```

Listado 12

nida la referencia al árbol DOM, buscamos el elemento y le cambiamos sus propiedades como podemos ver en el código del listado 13.

```
// Localiza un elemento HTML de tipo <span>, le cambia el color
// del tipo de letra y le asigna un contenido.
HtmlElement element = HtmlPage.Document.GetElementById("Rotulo");
element.SetStyleAttribute("color", "Blue");
element.SetProperty("innerHTML", "Saludos desde dotNetManía");
```

Listado 13

El objeto **Document**, además, dispone de mecanismos para la creación dinámica de nuevos elementos HTML, así como propiedades internas que hacen referencia a los objetos principales de la página (**Title**, **Body**, etc.). Por tanto, también podemos crear elementos de forma sencilla, teniendo en cuenta que, una vez generado, el elemento deberá ser insertado en el árbol DOM para que podamos visualizarlo en tiempo de ejecución. El listado 14 crea un nuevo elemento de tipo **<p>**, le asigna unos valores, y lo inserta como primer hijo del elemento **<body>** de la página.

Y, de la misma forma, resulta trivial eliminar cualquier elemento. Por ejemplo, el código del listado 15, elimina el elemento creado en el código anterior.

```
// Crea un nuevo elemento HTML de tipo <p>
HtmlElement element = HtmlPage.Document.CreateElement("p");
element.Id = "Parrafo";
element.SetProperty("innerHTML", "Este es el nuevo elemento");
HtmlElement referenceElement = HtmlPage.Document.Body.Children[0]
                                as HtmlElement;
// Hacemos que el nuevo elemento sea el primer hijo de <body>
// para que se sitúa al inicio de todo el árbol de elementos
// HTML y aparezca arriba del todo. De la misma forma que hemos
// modificado 2 de sus propiedades, podríamos asignarle un estilo
// para cambiar su ubicación o cualquier otro atributo.
// Hay que notar que es preciso buscar un elemento de referencia
// a partir del cual realizar la inserción, por que lo requiere el
// método AppendChild
HtmlPage.Document.Body.AppendChild(element, referenceElement);
```

Listado 14

```
// Elimina el objeto creado previamente mediante
// el manejador de evento de Botón3
HtmlElement element = HtmlPage.Document.GetElementById("Parrafo");
if (element != null)
    element.Parent.RemoveChild(element);
```

Listado 15

Vemos, de esta forma que el puente Silverlight-HTML permite una gran flexibilidad, pudiendo incluso generar páginas completas, en función de situaciones de lógica de negocio, tal y como estábamos acostumbrados en aplicaciones clásicas ASP.NET.

Utilizando notación JSON para enviar objetos a JavaScript

Otra forma interesante de transferir información entre Silverlight y HTML/JavaScript es utilizando la notación JSON (*JavaScript Object Notation*), que resulta muy fácil de usar desde este lenguaje. Necesitamos, eso sí, una forma de seriar la información de objeto que pasamos al método JavaScript.

Podemos realizar esto con ayuda del objeto **DataContractJsonSerializer**, que pertenece al espacio de nombres **System.Runtime.Serialization.Json**, como podemos ver en el listado 10 que definía nuestra clase **Mensaje**. El constructor recibe el tipo (**Type**)

del objeto a seriar, y con una llamada al método **WriteObject** escribe la información seriada en un objeto **MemoryStream**. Finalmente, un objeto **StreamReader** se encarga de leer la información seriada y devolverla como valor de retorno.

Con esta base, un manejador de evento en Silverlight realiza la llamada que vemos en el código del listado 16.

```
// Enviar un mensaje a HTML utilizando la seriación JSON
// (JavaScript Object Notation). Lo interesenta aquí
// es que podemos utilizar un objeto complejo para el
// traspaso de la información entre los dos contextos.
Mensaje m = new Mensaje();
m.CuerpoMensaje = "Saludos desde Silverlight";
m.Hora_Com = System.DateTime.Now.ToShortTimeString();
m.Usuario = "Marino";
// enviarMensajeObjeto es el nombre de la función
// JavaScript que procesa el contenido del mensaje JSON
// Y lo muestra con una llamada a alert()
HtmlPage.Window.Invoke("enviarMensajeObjeto", m.SeriacionJSON(m));

// JavaScript
function enviarMensajeJSON(mensaje) {
    // Procesa objetos Mensaje previamente seriados
    // El formato JSON tiene la misma sintaxis que la creación de
    // objetos en JavaScript, de ahí que podamos utilizar eval()
    // para la construcción
    var jsMensaje = eval('(' + mensaje + ')');
    alert('Usuario: ' + jsMensaje.Usuario + '\nHora: ' +
        jsMensaje.Hora_Com + '\nContenido: ' +
        jsMensaje.CuerpoMensaje);
}
```

Listado 16

El puente HTML - Silverlight

La comunicación en sentido contrario, tiene algunos requisitos adicionales por ambas partes. Por un lado, es preciso registrar la aplicación Silverlight como manejable mediante código de *script*. Esto se hace con una llamada al método **RegisterScriptableObject**, disponible en el objeto que representa la página HTML que aloja el control (*HTMLPage*).

Por otra parte, debemos de marcar con dos atributos, tanto la clase como los miembros de la clase que queramos que sean accedidos desde JavaScript. La clase será mar-

cada mediante el atributo **ScriptableType** y los miembros accesibles (normalmente, métodos), con el atributo **ScriptableMember**.

El código del listado 17 muestra el método **MetodoLlamableJS**, que simplemente, recibe la información desde HTML y asigna el valor a un elemento Silverlight denominado **BloqueTexto2**.

```
// En la clase Page
[ScriptableType]
public partial class Page : UserControl
{
    public Page()
    {
        InitializeComponent();
        //Necesario para las llamadas desde HTML -> C#
        HtmlPage.RegisterScriptableObject("Page", this);
    }

    [ScriptableMember]
    public void MetodoLlamadoPorJS(string scriptDato)
    {
        // Este método es llamado desde JavaScript
        // por la función enviarMensajeASilverlight()
        BloqueTexto2.Text = scriptDato;
    }
}

<!-- en el contenedor ASPX -->
<script type="text/javascript" language="javascript">
function enviarMensajeASilverlight() {
    //Se obtiene el objeto Silverlight con getElementById
    var objSilverlight = document.getElementById("Xaml1");
    if (objSilverlight) {
        var jsMensaje = "Mensaje enviado a Silverlight";
        //Llamada a la función de Silverlight 4
        objSilverlight.content.Page.MetodoLlamadoPorJS(jsMensaje);
    }
}
</script>
```

Listado 17

La llamada a la función JavaScript **enviarMensajeASilverlight()** obtiene una referencia al complemento Silverlight (la etiqueta **<object>** contenedora), y a continuación apunta al nombre que se ha dado al objeto Silverlight al que llamamos (**Page**) y, después, al método que escribe la información en **BloqueTexto2**.

Interacciones con otras aplicaciones Silverlight

Una de las novedades de las últimas versiones consiste en la capacidad de comunicar dos aplicaciones Silverlight ejecutándose en la misma página Web. Esto puede ser interesante por razones de sincronía, lógica de negocio, etc.

El mecanismo utilizado se denomina *Messaging* (mensajería), y hay que definir (como en cualquier otro sistema de comunicaciones) un emisor y un receptor. El canal de comunicación será el propio entorno de ejecución y el mensaje transmitido se define en el proceso.

Las clases principales que necesitamos para el proceso son **LocalMessageSender**, que proporciona servicios para el emisor, y **LocalMessageReceiver** que hace lo propio con la parte receptora. Ambas pertenecen al espacio de nombres **System.Windows.Messaging**. El primero, dispone de un método **SendMessageAsync**, que admite como argumento una cadena y enviará el contenido de forma asíncrona.

El objeto **LocalMessageReceiver** dispone de un evento **MessageReceived**, en cuyo argumento *e* se recibe un objeto con toda la información necesaria: el mensaje enviado (**Message**), la respuesta —si quisiéramos asignarla— (**Response**), el nombre del receptor (**ReceiverName**), el dominio del emisor, etc.

Así de simple. Basta con construir una aplicación con dos componentes Silverlight que estén alojados en la misma página. Uno hará de emisor y el otro de receptor. Para el ejemplo, hemos construido la interfaz de comunicaciones más simple posible. Por parte del emisor, solamente una caja de texto (**txtMensajeAEnviar**) con el contenido del mensaje a emitir, y un botón de enviar (**btnEnviar**). En el receptor, otra caja de texto para alojar el mensaje recibido (**txtMensajeRecibido**).

```
<--Interfaz del Emisor -->
<TextBlock Height="23" HorizontalAlignment="Left" Margin="12,12,0,0" Name="textBlock1"
    Text="Emisor de mensajes" VerticalAlignment="Top" />
    <TextBox Height="23" HorizontalAlignment="Left" Margin="12,33,0,0"
        Name="txtMensajeAEnviar" VerticalAlignment="Top" Width="252" />
    <Button Content="Enviar" Height="23" HorizontalAlignment="Left"
        Margin="298,33,0,0" Name="btnEnviar" VerticalAlignment="Top" Width="75"
        Click="btnEnviar_Click" />

<--Interfaz del Receptor -->
<TextBlock Text="Receptor" Margin="5" Height="21" Width="53" />
<TextBox Height="57" HorizontalAlignment="Left" Name="txtMensajeRecibido" Margin="12"
    VerticalAlignment="Top" Width="376" />
```

En la parte XAML, el listado 18 muestra las partes significativas de esta estructura visual.

Y en el código .NET creamos y ponemos en marcha el sistema con unas pocas líneas, como vemos en el listado 19.

```
//Clase Emisor

using System.Windows.Messaging;

...
// En una declaración a nivel de clase
LocalMessageSender lms = new LocalMessageSender("Receptor");
...

// Y un método para enviar
private void btnEnviar_Click(object sender, RoutedEventArgs e)
{
    lms.SendAsync(txtMensajeAEnviar.Text);
}

//Clase Receptor
using System.Windows.Messaging;

...
// En una declaración a nivel de clase
LocalMessageReceiver receptorMensajes = new
    LocalMessageReceiver("Receptor");
// En el constructor
receptorMensajes.MessageReceived +=
    (object sender, MessageReceivedEventArgs e) =>
    {
        txtMensajeRecibido.Text = e.Message;
    };
receptorMensajes.Listen();
```

Listado 19

Todo el mecanismo se pone en marcha cuando el receptor llama al método **Listen**. Eso hace que —una vez establecidos los canales de comunicación— el receptor esté a la escucha y genere el evento **MessageReceived**, cuando se reciba un mensaje. Cada vez que se reciba uno, el manejador anónimo del evento creado en el constructor asignará el contenido del mensaje a la caja de texto **txtMensajeRecibido**.

La salida en ejecución tendría un aspecto similar al de la figura 9.

(A efectos prácticos, hemos situado ambos componentes Silverlight en sendas celdas de una tabla).



figura 9 Salida de la solución de mensajería, mostrando el mensaje emitido y recibido

Ejecución directa fuera del navegador (Out-of-Browser)

Una de las novedades que más llamó la atención de los desarrolladores en la versión 3 de Silverlight, fue la capacidad de configurar la aplicación para su ejecución en una ventana de escritorio independiente, fuera del navegador.

Con ello, Silverlight daba un paso enorme para su implantación como solución alternativa a WPF para aplicaciones ricas de escritorio que no necesiten de la instalación de .NET Framework (solo requieren el *runtime*), y también en los casos en los que la ejecución en otros entornos como Mac, suponga un valor añadido.

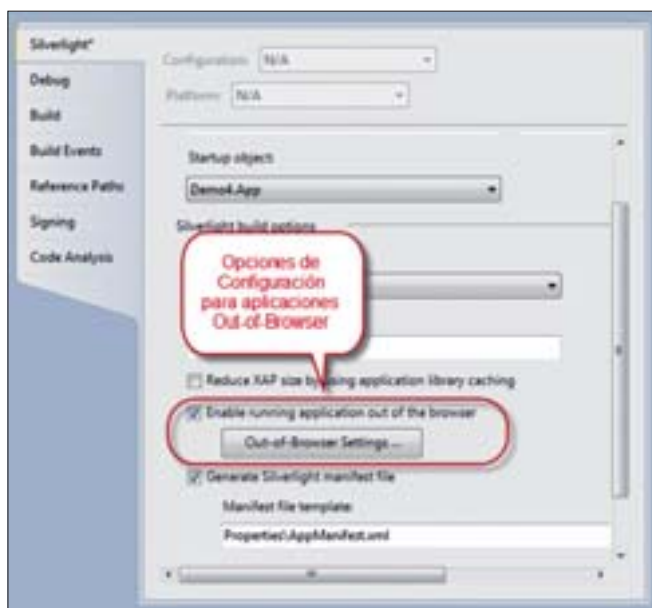


figura 10 Selección de la configuración de aplicaciones para ejecución fuera del navegador

En Silverlight 4, cualquier aplicación puede configurarse para su ejecución fuera del navegador y, de hecho, también es posible (con el consentimiento del usuario final), realizar una elevación de permisos de forma que la aplicación tenga capacidades similares a las de sus homónimas en Windows Forms y WPF. Esta configuración puede realizarse directamente con Visual Studio sobre la ventana de propiedades, que dispone de una ventana específica de configuración de estos aspectos (figura 10).

Además, el botón "Out-of-Browser Settings", nos conduce a otra ventana avanzada de configuración donde podemos establecer, desde el aspecto de la ventana que sustituirá al navegador (tamaño, iconos, título, etc.), hasta las configuración de seguridad (elevación de permisos) vinculada con ella.

Hay que tener en cuenta que una aplicación configurada para su ejecución fuera del navegador, tiene un comportamiento diferente si se la lanza desde Visual Studio 2010, respecto a su ejecución copiando y pegando la URL en un navegador. En el primer caso, si la hemos configurado ya, la aplicación se mostrará directamente fuera del navegador, mostrando la ventana predeterminada que le asigne el sistema (no será la



figura 11

Ventana de opciones avanzadas de configuración de un aplicación OOB

misma, lógicamente, en Windows XP que en Vista/Windows 7). La figura 11, muestra la ventana de configuración a la que hemos cambiado las opciones iniciales y añadido unos iconos para personalizarla.

Los valores establecidos aquí se almacenan en un fichero de nombre **OutOfBrowser-Settings.xml**, que se genera automáticamente en la carpeta **Properties** de la aplicación Silverlight. Si optamos por la segunda opción (pegar el enlace en una ventana del navegador), la aplicación aparecerá como siempre, pero, al pulsar botón derecho sobre la superficie útil, dispondremos de la opción "Instalar Demo4_OOB en este equipo...", tal y como muestra la figura 12-a. Además, aparecerá una ventana de sistema (figura 12-b) indicándonos que debemos seleccionar las opciones de instalación más adecuadas básicamente, la ubicación de los accesos directos para la ejecución desde el escritorio de Windows):



figura 12a y 12b Opción de instalación de la aplicación en el equipo local y ventana de opciones de instalación

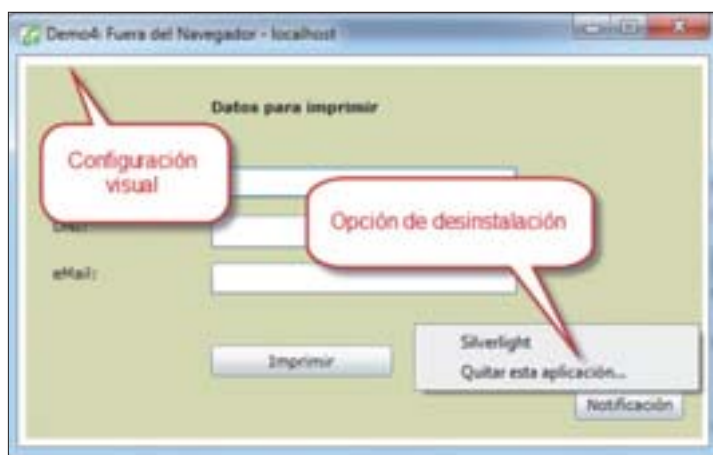


figura 13 Ventana de la aplicación en ejecución mostrando las opciones disponibles

Si optamos por la instalación predeterminada, la aplicación se ubicará en nuestro equipo y un enlace estará disponible desde el menú de inicio, generando una salida idéntica al caso de la ejecución directa desde Visual Studio, lanzándose la ejecución en la nueva ventana. Si queremos deshacer la instalación, también disponemos de un mecanismo automático: pulsando botón derecho, aparecerá una opción del menú contextual para retirar la instalación recién instalada (figura 13).

Y todo esto, solo por configurar las opciones desde Visual Studio. No hemos tenido que escribir ni una sola línea de código.

Instalación con permisos elevados

Hasta el momento, no hemos modificado nada referente a los permisos de ejecución de la aplicación. Uno de los cambios que podemos hacer al habilitar esta opción es modificar el aspecto de la ventana (denominada también *Chrome*), que puede configurarse de 3 maneras distintas, según se aprecia en la figura 14:

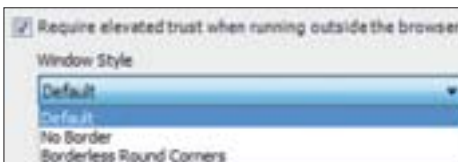


figura 14

Combo de opciones para el tipo de ventana cuando se habilitan permisos elevados

Curiosamente, aunque el color de fondo del elemento **UserControl** principal se establezca en **Transparent**, la aplicación Silverlight asumirá un color blanco de fondo (aquí no es posible establecer que la ventana sea transparente, tal y como se hace en WPF). Además, en el proceso de instalación se advierte al usuario sobre el publicador (no verificado en este caso), de forma que sea el usuario el responsable de la asignación de esas opciones (figura 15).

El problema de la certificación se resuelve, obviamente, firmando digitalmente el fichero XAP de la aplicación. De cara a la producción y distribución, existen muchas entidades de certificación oficiales. No entramos en eso, pero sí merece la pena hacer notar que podemos generar una certificación para propósitos de desarrollo, que ofrecería el aspecto más adecuado y la posibilidad de testar la aplicación de forma similar a como sería en producción. Para más detalles, ver la siguiente página explicativa del



figura 15 Ventana para la Instalación con permisos elevados

evangelista tecnológico **John Papa**: <http://johnpapa.net/silverlight/digitally-signing-a-xap-silverlight>.

Una vez asignados permisos elevados, nuestra aplicación Silverlight gana acceso a las siguientes opciones de acceso al sistema:

- Guardar y leer documentos en las carpetas del usuario, como "Mis Documentos", "Mis Imágenes", "Mis Vídeos", etc.
- Acceder a servidores de otros dominios (aunque no dispongan de un fichero de accesos del tipo **CrossDomain.xml** o similar).
- Acceder al portapapeles sin preguntar al usuario
- Interoperabilidad con COM
- Alojamiento de HTML en la aplicación Silverlight
- Mostrar ventanas de notificación de Windows

Actualizaciones

Desde el punto de vista de las actualizaciones, una aplicación Silverlight dentro del navegador utiliza los mismos mecanismos de caché que otras aplicaciones Web. Cuando una petición Web es procesada, se comprueba la existencia en caché. Si exis-

te, se usa ese fichero y se lanza una petición al servidor para comprobar si existe una versión más reciente. Si existe, se descarga y si no, se utiliza el del caché. Pero en las aplicaciones OOB, se guardan copias locales de los ficheros en la máquina del usuario, y no existe la comprobación automática.

En este caso, podríamos solucionar el problema acreditando cual es la situación al inicio (para evitar que la experiencia de usuario resulte pesada, podríamos descargar un fichero de texto donde apuntásemos el número de versión, y solamente solicitar al usuario la actualización en caso de cambio).

Instalación dirigida por el usuario

En el caso de que prefiramos que sea el usuario el que opte por la opción de instalación, podemos habilitar cualquier mecanismo adecuado (un botón o similar) y utilizar la instancia del objeto **Application** disponible a través de la propiedad **App.Current**, y proceder a la instalación mediante una llamada al método **Install** de ese objeto como vemos en el listado 20:

```
private void btnInstalar_Click(object sender, RoutedEventArgs e)
{
    try
    {
        if (App.Current.IsRunningOutOfBrowser)
        {
            MessageBox.Show("La aplicación ya está instalada");
            return;
        }
        App.Current.Install();
    }
    catch (InvalidOperationException)
    {
        MessageBox.Show("Operación no válida");
    }
}
```

Listado 20

Ventanas de notificación

Otra de las opciones que podemos codificar en una aplicación Silverlight OOB es la posibilidad de enviar mensajes de notificación que aparecen en una ventana separada, configurable por el usuario, en la parte inferior derecha de la pantalla.

Su uso es muy sencillo, y se basa en la clase **NotificationWindow**. Basta con crear una instancia de la clase, y posteriormente, añadir el contenido que queremos mostrar (podemos crear un elemento en XAML y cargarlo dinámicamente o generar los elementos directamente en código .NET. Desde el punto de vista del contenido, cualquier **UIElement** es factible de ser incluido en la ventana de notificación, por lo que disponemos de total libertad de personalización. Hay que tener en cuenta que el tamaño máximo de un elemento **NotificationWindow** es de 400x100 píxeles.

Otro aspecto a considerar es que, al tratarse de una aplicación OOB, no necesitamos empotrar más que las fuentes necesarias, en caso de que no tuviéramos la garantía de que existiesen en el sistema. En nuestro caso, como sabemos que la fuente de letra Verdana está presente en cualquiera de los dos entornos (Windows y Mac), podemos hacer referencia a ella sin incluirla en el fichero XAP correspondiente.

Para lanzar la notificación, hemos creado un botón para lanzar la notificación (**btnNotificacion**), e incluimos un icono y un mensaje de texto alojado en un elemento **TextBlock**. El listado 21 muestra el código asociado al botón.

```
private void btnNotificacion_Click(object sender, RoutedEventArgs e)
{
    //Objeto Notification
    NotificationWindow notificacion = new NotificationWindow();
    notificacion.Width = 340;
    notificacion.Height = 54;
    // Icono (Image)
    BitmapImage icono = new BitmapImage(new
        Uri("/Recursos/Demo4_48.png", UriKind.Relative));
    Image img = new Image();
    img.Width = 48; img.Height = 48;
    img.Source = icono;
    img.Margin = new Thickness(2.0d);
    // Mensaje (TextBlock)
    TextBlock tb = new TextBlock();
    tb.Text = "Tiene un mensaje de la empresa";
    tb.FontSize = 16;
    tb.FontFamily = new System.Windows.Media.FontFamily("Verdana");
    tb.Margin = new Thickness(5, 15, 0, 0);
    // Contenedor (StackPanel)
    StackPanel sp = new StackPanel();
    sp.Orientation = Orientation.Horizontal;
    sp.Children.Add(img);
    sp.Children.Add(tb);
    notificacion.Content = sp;
    notificacion.Show(2000);
}
```

Listado 21

Y obtendremos una ventana como la de la figura 16.

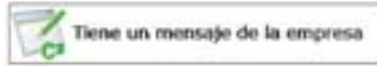


figura 16 Mensaje de notificación correspondiente al listado 21

Mostrar contenido HTML en aplicaciones OOB

Un problema que se plantea aquí es el de mostrar contenido HTML en una aplicación fuera del navegador (no es posible alojado en un navegador, ni tendría mucho sentido). El usuario puede utilizar el control **WebBrowser**, que solo está disponible para este tipo de aplicaciones, y utilizar su contenido para dibujar una superficie (como un rectángulo) mediante un elemento **Brush** del tipo **WebBrowserBrush**. El primero, suministra contenido y el **Brush** se encarga de la parte visual, como en el código del listado 22.

Ante de probar el código recuerde que tiene que activar la posibilidad de ejecución OOB y en la configuración avanzada indicar la opción "*Required elevated trust...*"

```
<StackPanel x:Name="LayoutRoot" Background="Blue" >
  <WebBrowser Name="WB1" Source="http://elavefenix.net"
    Height="480" Width="640" />
  <Rectangle Height="480" Width="640" Visibility="Collapsed">
    <Rectangle.Fill>
      <WebBrowserBrush SourceName="WB1" Opacity=".5" x:Name="WBB1"/>
    </Rectangle.Fill>
  </Rectangle>
</StackPanel>
```

Listado 22

Curiosamente, al navegar a mi sitio y abrir la página principal, se carga (y ejecuta) un componente Silverlight utilizado como saludo navideño. Esto no supone ningún problema en la ejecución, y la parte dinámica del componente se ejecuta exactamente igual que en otro contexto más habitual, como muestra la figura 17.



figura 17 La aplicación dibujando en un rectángulo el contenido de www.elavefenix.net (que, a su vez, carga un complemento Silverlight)

Acceso a otros recursos

Otra de las novedades que aparecieron en Silverlight 4, se vincula con los recursos multimedia del equipo donde se ejecuta la aplicación. En concreto, ahora se dispone de soporte para la captura de vídeo y audio, mediante las clases asociadas, que nos permiten comprobar la disponibilidad del vídeo y del audio, iniciar y detener las capturas, etc.

Para probar un ejemplo, nos basta con un par de botones para iniciar y detener la captura y un elemento visual (**Rectangle**, en nuestro caso), que sirva para mostrar la salida. El listado 23 muestra el código XAML significativo para la definición de la interfaz.

Y en el código fuente .NET, programamos el comportamiento (especialmente del evento **btnCapturar_Click**), tras haber declarado una variable del tipo **CaptureSource** a nivel de clase). Posteriormente comprobamos la disponibilidad de la fuente de vídeo, asignamos canales de escucha y utilizamos un elemento **VideoBrush** para mostrar la salida del vídeo.

```

<Rectangle Height="247" HorizontalAlignment="Left"
    Margin="12,12,0,0" Name="recCaptura" Stroke="Black"
    StrokeThickness="1" VerticalAlignment="Top"
    Width="334" RadiusX="12" RadiusY="12" />
<Button Content="Iniciar Captura" Height="23" Margin="12,265,0,0"
    Name="btnCapturar" VerticalAlignment="Top"
    HorizontalAlignment="Left" Width="157"
    Click="btnCapturar_Click" />
<Button Content="Detener Captura" Height="23"
    HorizontalAlignment="Right" Margin="0,265,12,0"
    Name="btnDetener" VerticalAlignment="Top" Width="155"
    Click="btnDetener_Click" />

```

Listado 23

En resumen, los pasos a dar son los siguientes:

- Comprobar la existencia y disponibilidad del dispositivo
- Comprobar los permisos (el usuario puede otorgarlos la primera vez que se le solicitan y el sistema recordará esta asignación).
- Asignar al dispositivo los canales de audio y vídeo correspondientes
- Iniciar el dispositivo

Debemos tener presente que el funcionamiento del vídeo asume que no hay otro dispositivo activo utilizándolo, y que requiere permisos del usuario para poder acceder, por lo que, al principio (solicitaremos los permisos en el evento **Loaded** del control), nos aparecerá la caja de diálogo que solicita el acceso a la videocámara, como se muestra en la figura 18.

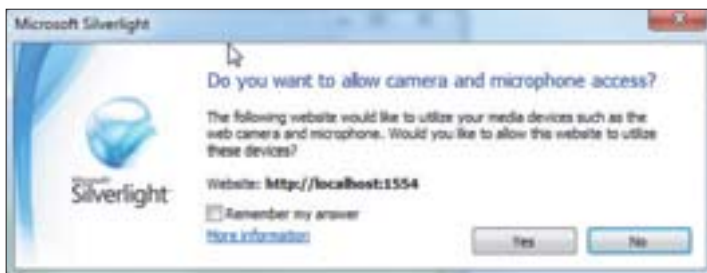


figura 18

Caja de diálogo de solicitud de permisos (advierta la opción "remember my answer")

Además, recuerde el lector que la configuración específica del vídeo (y audio) pre-determinado está accesible desde el punto de vista de los permisos y disponibilidad en la ventana de configuración de Silverlight a la que accedemos en tiempo de ejecución pulsando botón derecho (figura 19). Ahí podemos asegurarnos de que Silverlight ha detectado correctamente estos dispositivos (en este caso, un Microsoft Live WebCam).



figura 19 Solapa de configuración de audio y vídeo para aplicaciones Silverlight

Hay que tener en cuenta que, por el momento, Silverlight no dispone de mecanismos de codificación (*encoding*) para generar fácilmente un *streaming* almacenable de las capturas. Aunque es posible realizarlo, mediante las clases **AudioSink** y **VideoSink** y posteriormente implementando un número de retro-llamadas (**OnCaptureStarted**, **OnCaptureStopped**, **OnFormatChange** y **OnSample**) su realización se sale enteramente de la reducida longitud de este cuaderno técnico.

El listado 24 muestra la clase completa donde implementamos la lógica de la captura y puesta en marcha del dispositivo.

Y al ejecutar, y después de permitir el acceso al dispositivo, tendríamos una imagen mostrándose dinámicamente en el rectángulo asociado (naturalmente, si dispusiéramos del algún software de vídeo que genere efectos en la cámara, estos son capturados igualmente –figura 20–).

```

using System.Windows;
using System.Windows.Controls;
using System.Windows.Media;

namespace VideoYAudio
{
    public partial class MainPage : UserControl
    {
        CaptureSource dispositivo = new CaptureSource();
        public MainPage()
        {
            InitializeComponent();
            Loaded += MainPage_Loaded;
        }

        void MainPage_Loaded(object sender, RoutedEventArgs e)
        {
            // Una vez cargada la página, detectamos el
            // dispositivo de vídeo y los permisos, solicitándolos
            // al usuario si se han concedido todavía
            if (!CaptureDeviceConfiguration.AllowedDeviceAccess)
            {
                CaptureDeviceConfiguration.RequestDeviceAccess();
            }
        }

        private void btnCapturar_Click(object sender, RoutedEventArgs e)
        {
            // Si tenemos permisos, asignamos los canales de captura
            // de vídeo y audio
            if (CaptureDeviceConfiguration.RequestDeviceAccess())
            {
                VideoCaptureDevice canalVideo =
                    CaptureDeviceConfiguration.GetDefaultVideoCaptureDevice();
                AudioCaptureDevice canalAudio =
                    CaptureDeviceConfiguration.GetDefaultAudioCaptureDevice();
                // Si ambos funcionan correctamente, asignamos al
                // dispositivo los canales de captura
                if (canalVideo != null && canalAudio != null)
                {
                    dispositivo.VideoCaptureDevice = canalVideo;
                    dispositivo.AudioCaptureDevice = canalAudio;
                    // Un elemento VideoBrush servirá de enlace
                    // y elemento visualizador
                    var vBrush = new VideoBrush();
                    vBrush.SetSource(dispositivo);
                }
            }
        }
    }
}

```

```

        dispositivo.Start();
        recCaptura.Fill = vBrush;
    }
}

private void btnDetener_Click(object sender, RoutedEventArgs e)
{
    // Si no se ha perdido la conexión, detenemos
    // el dispositivo.
    if (dispositivo.State == CaptureState.Started)
    {
        dispositivo.Stop();
    }
}
}
}

```

Listado 24



figura 20 Salida del código anterior en ejecución

Soporte de arrastrar-y-soltar (Drag&Drop)

Otra de las capacidades de Silverlight que facilitan la manipulación de archivos (y que no requiere de permisos especiales) es la de Arrastrar-y-Soltar, que todos los usuarios manejamos habitualmente. Esto funciona para aplicaciones alojadas en HTML como para aplicaciones OOB.

En principio es muy sencillo y vamos a poner un pequeño ejemplo para ilustrarlo, utilizando ficheros de texto que puedan ser volcados a una superficie receptora e inmediatamente, abiertos y mostrados en pantalla (un simple **Grid** es válido para este propósito).

La parte clave del proceso es el manejo del fichero en el evento **Drop** del elemento sobre el que queremos permitir que se arrastre el fichero a mostrar. Éste evento, dispone —en el argumento **e**— de información asociada con los ficheros soltados (podríamos arrastrar un grupo de ellos, y deberíamos ir procesándolos uno a uno), vinculada con la propiedad **e.Data**, como podemos ver en la captura de pantalla de la figura 21.

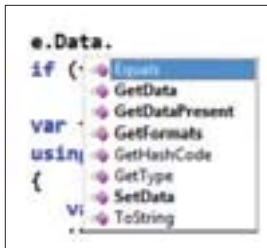


figura 21 Propiedades del objeto Data en el argumento e del evento Drop

```
<!-- En la parte XAML -->

<Grid x:Name="LayoutRoot" Background="#FFF5F4E5" AllowDrop="True"
      Drop="LayoutRoot_Drop"/>

// Y el manejador del evento Drop en la clase MainPage...
private void LayoutRoot_Drop(object sender, DragEventArgs e)
{
    var files = e.Data.GetData(DataFormats.FileDrop) as FileInfo[];
    if (files == null || files.Length == 0) { return; }

    var file = files[0];
    using (var reader = file.OpenText())
    {
        var tb = new TextBlock();
        tb.Width = LayoutRoot.Width;
        tb.Height = LayoutRoot.Height;
        tb.TextWrapping = TextWrapping.Wrap;
        tb.Text = reader.ReadToEnd();
        LayoutRoot.Children.Add(tb);
    }
}
```

A partir de este objeto, es posible determinar si realmente disponemos de datos, si éstos se pueden convertir a un formato determinado (**GetDataPresent**) y la lista de formatos convertibles (**GetFormats**). Además, en el método **SetData**, contamos con la posibilidad de convertir el formato original a otro más adecuado antes de recuperarlo con el método **GetData**. A éste método se le pasará como argumento el único posible en este contexto (el formato de ficheros de Windows), que se indica con el enumerado **DataFormats.FileDrop**.

El código no puede ser más sencillo, tanto en la parte de la interfaz XAML como en el código .NET, como se ve en el listado 25.

Y ya está. El elemento **TextBlock** que muestra el texto se crea y añade dinámicamente, y la propiedad **TextWrapping**, garantiza que el texto leído se adapte a la anchura del navegador. La salida después de volcar el fichero **LoremIpsum.txt**, es la que se ve en la figura 22.



figura 22

Resultado de soltar el fichero "LoremIpsum.txt" en la superficie útil de Silverlight

Conclusión

Silverlight ha ido evolucionando y convirtiéndose cada vez más en una herramienta multipropósito que dispone de capacidades de proceso interno, navegación entre páginas, interacción con la página que lo aloja, con el sistema, y con muchos más recursos locales en el modo de ejecución *Out-of-Browser*. En el próximo capítulo vamos a analizar Silverlight como plataforma para la construcción de aplicaciones para Windows Phone 7.

Introducción a Windows Phone 7

WP7: una nueva plataforma

En octubre de 2010, Microsoft anunciaba su nuevo diseño para teléfonos móviles. Basado en una nueva versión del sistema operativo, específicamente creada para estos dispositivos y otros similares, WP7 nació con capacidades de manipulación gestual, una pantalla de 480x800 píxeles, y un diseño simplificado, que asume que los usuarios ya tienen una cierta educación en el uso de móviles.

Además, el hecho de que la plataforma sea programable mediante Silverlight y/o XNA, permite que los programadores dispongan de una gran libertad desde el inicio a la hora de decidir las herramientas de desarrollo para una aplicación o juego para este contexto. Como veremos, el SDK de Silverlight para WP7¹ es prácticamente la versión de Silverlight 3², con ciertas mejoras adecuadas para el entorno.

En cuanto a la distribución de programas o utilidades para WP7, esta se realiza mediante la *Windows Phone 7 Marketplace*, que requiere de un registro, y certifica unos mínimos requisitos de fiabilidad, eficiencia y comportamiento adecuado.

En cuanto a la operativa con el dispositivo, dispone solamente de 3 botones de manipulación, ya que el resto de opciones son táctiles: *Back* (Atrás), *Start* (Inicio) y *Search* (Buscar). El primero, funciona de forma similar a su equivalente de los navegadores, y si nos encontramos en la página principal de una aplicación, ésta se cierra. El botón *Start/Windows* lleva al usuario a la pantalla principal del teléfono. Por otro lado, es



¹ Disponible en el sitio App Hub, en <http://create.msdn.com/en-US/>.

² La referencia oficial de las diferencias entre la versión 3.0 de Silverlight y la versión para Windows Phone 7, se encuentra en [http://msdn.microsoft.com/en-us/library/ff426930\(VS.96\).aspx](http://msdn.microsoft.com/en-us/library/ff426930(VS.96).aspx).

inaccesible para los programas en ejecución. El tercero busca por el término que introduzcamos en todos los elementos accesibles.

Opcionalmente, el fabricante puede añadir un teclado físico, e, igualmente, se dispone de uno virtual, también denominado SIP (*Soft Input Panel*).

Otros elementos de hardware

Además de los citados, existen otros elementos en el hardware del dispositivo que debemos de tener en cuenta, los llamados sensores, que —en ocasiones— nos permiten el acceso a servicios de software. Los que más nos afectan como desarrolladores son (por orden alfabético):

Acelerómetro. Imprescindibles en dispositivos táctiles, ya que detecta la aceleración, o cambio de velocidad. Y cuando la cámara está quieta, detecta la gravedad, pudiendo indicarnos la orientación del dispositivo.

Cámara. Al menos, se dispone de una cámara de 5 Mpx con flash. Los programas pueden llamar a la cámara por código, y acceder a las fotos almacenadas.

Notificaciones *Push*. Permite configurar las condiciones bajo las que se reciben notificaciones por servicios de internet.

Radio FM. Otro servicio configurable y accesible mediante programación.

Ubicación. el teléfono suministra un sistema de posicionamiento (GPS), tanto desde servicios de Internet, como a través de las operadoras. También puede averiguarse la ruta y la velocidad si el dispositivo está en movimiento.

Vibración. También podemos programar este comportamiento por código.

Wi-Fi. se dispone de Wi-Fi para el acceso a Internet, como complemento al acceso via 3G que pueda suministrar el proveedor de servicios de telefonía. Se incluye una versión de IE con el software del dispositivo.

Existen, además otros dispositivos (algunos dependen del fabricante), como los LED, emulador de ratón óptico, ruleta de teclado, sensor de luz, la salida de vídeo, etc. En algunos casos, como en el de Samsung, el propio fabricante suministra un SDK complementario para permitir la programación de estas opciones.

Arquitectura de Windows Phone 7 para el desarrollador

Desde el punto de vista de la programación, tanto si utilizamos Silverlight como el marco XNA para las aplicaciones WP7, todo el desarrollo se realiza en código manejado y en un contexto de seguridad (*sandbox*) exactamente igual que lo visto hasta ahora. Esto permite el desarrollo rápido de aplicaciones seguras y sólidas y —con la salve-

Respecto a la documentación oficial, como siempre, hemos de recomendar la del MSDN arriba indicada, junto con la propia página oficial de Silverlight, donde se encuentran numerosos recursos relativos a WP7, en forma de ejemplos, vídeos y código fuente.

Además, hemos de añadir que, justo cuando esta obra estaba a punto de pasar a imprenta, el evento CES (5-9 de enero/2011)⁵, anunció que en febrero de 2011 se podría contar con algunas primeras actualizaciones que mejoran las prestaciones actuales y/o resuelven algunas carencias iniciales como el mecanismo "Copiar y Pegar", que no está disponible en la primera versión. En concreto la lista de actualizaciones disponibles⁶ recogería esta característica (junto a etiquetas inteligentes), además de mejoras en el rendimiento de aplicaciones y juegos y un sistema más fácil de búsqueda de aplicaciones en el sitio oficial (*Windows MarketPlace*).

Programación para WP7

El primer paso, es descargar el SDK del sitio indicado más arriba, haciendo la salvedad de que es preferible descargar la versión de "Instalación Web", que ocupa poco más de 3 Mb, y analiza la plataforma de instalación y los componentes instalados. La descarga puede ocupar algo más de 400 Mb. Una vez hecho esto, y completada la instalación, nos encontraremos con dos mecanismos de creación: en Visual Studio 2010 (Professional o Ultimate) aparecerá un nuevo conjunto de tipos de proyecto asociados con el lenguaje C#, llamado XNA Game Studio 4.0, donde podremos desarrollar li-



figura 2

Tipos de proyecto relacionados con Windows Phone 7

⁵ <http://www.cesweb.org/>

⁶ <http://www.microsoft.com/windowsphone/en-us/features/update-info.aspx>

brerías y juegos para WP7. Además, se instalará una versión de Visual Studio 2010 Express para Windows Phone, que será la que utilizaremos aquí. Tras abrir el entorno de desarrollo, vemos que nos ofrece los tipos de aplicación WP7 que aparecen en la figura 1. Seleccionamos el primero: "Aplicación de Windows Phone".

Una vez seleccionado un nombre para nuestro programa, veremos cómo se presenta en la parte izquierda de la pantalla una versión del Emulador de Windows Phone 7 mostrando los valores predeterminados.

Conviene observar algunos cambios notables respecto a la estructura de un programa Silverlight 4 "estándar":

- a) Bajo el apartado "*Properties*", además de los habituales **AppManifest.xml** y **AssemblyInfo.cs**, aparece un fichero de nombre **WMAppManifest.xml**, que contiene algunas definiciones necesarias para la ejecución y distribución de la aplicación, como la referencia a los ficheros gráficos de fondo e icono de la aplicación y un título predeterminado **WindowsPhoneApplication1**, o algo similar. Aquí podemos cambiar el título a nuestro gusto. También aparece como tarea predeterminada una página de navegación que apunta al fichero **MainPage.xaml**, que es la pantalla principal de la aplicación.
- b) La aplicación dispone, al igual que otras aplicaciones Silverlight, de un punto de entrada implementado mediante el par de ficheros **App.xaml** y **App.xaml.cs**. El primero, define un apartado **ApplicationLifetimeObjects** donde se asignan los manejadores de evento principales a nivel de aplicación, tal y como podemos ver en el listado 1.

```
<Application.ApplicationLifetimeObjects>
  <!--Objeto requerido que controla los eventos de duración de la aplicación-->
  <shell:PhoneApplicationService
    Launching="Application_Launching"
    Closing="Application_Closing"
    Activated="Application_Activated" Deactivated="Application_Deactivated"/>
</Application.ApplicationLifetimeObjects>
```

Listado 1

No obstante, lo realmente interesante se establece en el fichero de *code behind*: la clase **App**, que hereda de **Application**, añade en su constructor una llamada a **InitializePhoneApplication()**, justo a continuación de **InitializeComponent()**. Es en éste método donde se crea el marco de presentación visual, re-

presentado por una instancia de la clase **PhoneApplicationFrame**. Como se indica en los comentarios del propio código, **RootVisual** no es asignado todavía, ya que esto permite que la pantalla de presentación permanezca activa hasta que la aplicación esté lista para la presentación. Ese proceso diferido se completa en el manejador **CompleteInitializePhoneApplication**.

- c) En el apartado de las referencias a librerías, veremos 2 nuevas entradas: **Microsoft.Phone** y **Microsoft.Phone.Interop**. La primera y principal se encuentra vinculada en el fichero **MainPage.xaml** mediante 2 espacios de nombres: **xmlns:phone** y **xmlns:shell**, con los que obtenemos acceso a muchas de las funcionalidades básicas del teléfono.

En este punto, y una vez cambiados los títulos iniciales, conviene compilar y ejecutar la aplicación, aunque solo sea para comprobar que todo funciona correctamente hasta aquí. Se lanzará el emulador, y deberíamos obtener una imagen similar a la de la figura 3-a:



figura 3a y 3b Pantalla inicial de la aplicación de prueba WindowsPhoneApplication1, y listado de la pantalla principal donde aparece la aplicación

Pulsando el botón "Atrás", nuestra aplicación aparecerá (con el nombre e icono predeterminados) en el listado de la pantalla principal, como se aprecia en la figura 3-b.

De vuelta en nuestro código fuente de **MainPage.xaml**, podemos notar una sección fuera del **Grid** principal (**LayoutRoot**) que está marcada como comentario, y tiene como elemento contenedor **<phone:PhoneApplicationPage.ApplicationBar>**. Se trata de la barra de botones de la aplicación (**ApplicationBar**), que aquí, consta de un único elemento (**shell:ApplicationBar**), donde podemos ubicar los botones principales de nuestra aplicación y los elementos de menú. En el ejemplo inicial, se incluyen dos botones y dos elementos de menú. Al pulsar cualquiera de los botones, se despliega una zona in-

ferior, mostrando todos los elementos disponibles. Una vez eliminado el comentario, compilada la aplicación y pulsado uno de los botones, aparece en esa zona inferior de la pantalla esa zona con un aspecto como muestra la figura 4.



figura 4 Zona inferior mostrando los botones de aplicación y los menús

Con esto en cuenta y utilizando los conceptos que hemos visto hasta ahora, no nos resulta complicado modificar la página principal para personalizarla, añadiendo elementos gráficos (un elemento **Image** con el logo de la revista), una descripción (**TextBlock**) y un enlace al sitio web (**HyperLinkButton**), dentro del *grid ContentPanel*, con el código del listado 2, de forma que la portada inicial de la aplicación ofrezca el siguiente aspecto que vemos en la figura 5.

```
<!--ContentPanel. Modificado para mostrar la página inicial -->
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
  <Image Source="Images/dnm.png" Width="200"/>
  <TextBlock x:Name="CabeceraLogo"
    Text="La revista del programador en tecnologías .NET" Margin="128,387,128,0"
    TextWrapping="Wrap" TextAlignment="Center" />
  <HyperlinkButton Foreground="#FF326676" NavigateUri="http://www.dotnetmania.com"
    TargetName="dotNetManía" HorizontalAlignment="Center" Content="www.dotNetManía.com"
    Margin="100,22,96,471" />
</Grid>
```

Listado 2

Como vemos, nada especial en el código XAML de los elementos utilizados hasta ahora, respecto a lo que hemos visto en otras aplicaciones Silverlight. El editor visual de Visual Studio nos va indicando las posiciones y aspecto de los elementos antes

de la ejecución. Además, si modificamos el atributo **SupportedOrientations="PortraitOrLandscape"** de esta forma, la aplicación también será visible cuando cambiemos la orientación del dispositivo.

Si ejecutamos la aplicación veremos la pantalla tal y como se espera, y —si disponemos de una conexión a Internet—, y seleccionamos el hipervínculo, se abrirá el navegador IE y comenzará la descarga del sitio al que se apunta.



figura 5 Resultado de las modificaciones al elemento *ContentPanel*

Características propias del teléfono: Orientación, Manipulación Gestual y Navegación

Las librerías del teléfono nos ofrecen acceso a otros aspectos propios de este dispositivo, como la orientación. La orientación viene especificada por un valor enumerado (que hemos utilizado en el código XAML del ejemplo anterior), y podemos responder a los cambios físicos de la orientación mediante una respuesta adecuada en el evento **Orientation_Changed**, de la página activa. En el manejador de evento, el argumento **e**, dispone de una propiedad **Orientation**, que podemos interrogar en tiempo de ejecución para saber cuál es la orientación en curso. En el siguiente ejemplo, hemos incluido un **TextBox** adicional para ilustrar este aspecto, y mostramos el valor enumerado correspondiente al simular el cambio de orientación con el emulador.

El código fuente, muy simple, es el que aparece en el listado 3, como parte del código de la página:

```

<!-- En el ContentPanel (código XAML)-->
<!--ContentPanel. Colocar aquí el contenido adicional-->
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <TextBlock x:Name="txtSaludo" Text="Saludos desde DNM" />
    <TextBlock x:Name="txtOrientacion" Text="Orientación:
        " VerticalAlignment="Center" />
</Grid>

// Y en la clase MainPage
public MainPage()
{
    InitializeComponent();
    this.OrientationChanged += new EventHandler<OrientationChangedEventArgs>(
        MainPage_OrientationChanged);
}

void MainPage_OrientationChanged(object sender, OrientationChangedEventArgs e)
{
    txtOrientacion.Text = "Orientación: " + e.Orientation.ToString();
}

```

Listado 3

La imagen de la figura 6 muestra los valores recogidos como parte del valor enumerado **PageOrientation**.

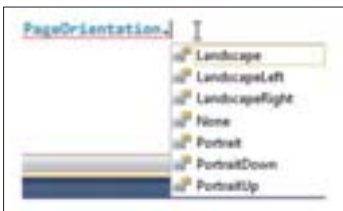


figura 6 Valores posibles del enumerado PageOrientation

Naturalmente, la clase suministra también sobrecargas de métodos vinculados a todos los eventos importantes, que van precedidos por el prefijo "On". De forma que podríamos haber utilizado la forma equivalente:

```
protected override void OnOrientationChanged(OrientationChangedEventArgs e)
{
    txtOrientacion.Text = "Orientación: " + e.Orientation.ToString();
    base.OnOrientationChanged(e);
}
```

El lector podrá comprobar que —en ambos casos— los resultados son equivalentes, tal y como sería de esperar de una aplicación Silverlight estándar.

Manipulación (Touch)

Una de las opciones novedosas que estos dispositivos incluyen para el programador es la gestión de las capacidades gestuales que ofrecen este tipo de pantallas. A esta característica se la denomina *Touch* o, más concretamente en el caso de Windows Phone 7, *Multi-Touch*, ya que estas pantallas son capaces de reconocer las acciones de hasta 4 dedos simultáneamente.

Silverlight soporta dos API diferentes para manejar el concepto de *Multi-Touch*, que podemos considerar como de bajo nivel y de alto nivel. El de bajo nivel, se basa en el evento **TouchFrameReported**. El de alto nivel, se centra en 3 eventos definidos en la clase **UIElement**: **ManipulationStarted**, **ManipulationDelta** y **ManipulationCompleted**.

La base de la interacción de bajo nivel es la clase **TouchPoint**, representando cada instancia de esta clase a uno de los dedos que interactúe con la pantalla. Esta clase dispone de 4 propiedades de solo lectura:

- **Action**. Enumerado (del tipo **TouchAction**), de valores **Down**, **Up**, y **Move**.
- **Position**. (De tipo **Point**), indicando la ubicación del dedo respecto al origen de la pantalla.
- **Size**. Aunque su funcionamiento correcto tiene algunos problemas, representa la superficie del área de interacción, y, por tanto, la presión ejercida.
- **TouchDevice**. Objeto con dos propiedades de solo lectura: **Id**, un entero utilizado para distinguir los dedos, y **DirectlyOver**, un elemento del tipo **UIElement** que representa el objeto más cercano al área de pulsación en el eje de las Z.

Pongamos todo esto en funcionamiento con un ejemplo que responda a las pulsaciones con alguna acción visible, como —por ejemplo— aumentar el tamaño de la letra de una caja de texto. Tendremos que guardar el valor original del tamaño de la le-

tra, e ir añadiendo un par de puntos de tamaño cada vez que el usuario pulse sobre el texto. También habilitaremos un mecanismo para retornar al valor original cuando pulsemos fuera del área de acción de la caja de texto.

El listado 4 muestra la clase **MainPage**, con las modificaciones añadidas al manejador del evento **OnTouchFrameReported**.

Observe el lector como toda la manipulación gira en torno al objeto **TouchPoint**, recuperado a partir del objeto **args** que se recibe como argumento (previamente, hemos asignado el manejador del evento **FrameReported** en el constructor de la clase). Teniendo esto en cuenta, el código que sigue es bastante simple.

```
public partial class MainPage : PhoneApplicationPage
{
    double tamañoInicial;
    // Constructor
    public MainPage()
    {
        InitializeComponent();
        tamañoInicial = txtTextoCreciente.FontSize;
        Touch.FrameReported += OnTouchFrameReported;
    }

    void OnTouchFrameReported(object sender, TouchFrameEventArgs args)
    {
        TouchPoint primaryTouchPoint = args.GetPrimaryTouchPoint(null);
        if (primaryTouchPoint != null && primaryTouchPoint.Action == TouchAction.Down)
        {
            if (primaryTouchPoint.TouchDevice.DirectlyOver == txtTextoCreciente)
            {
                txtTextoCreciente.FontSize += 2;
            }
            else
            {
                txtTextoCreciente.FontSize = tamañoInicial;
            }
        }
    }
}
```

Listado 5

En oposición, los eventos **Manipulation*** antes citados, no llevan el control individual de los dedos que producen el evento, sino que consolidan el proceso de manipulación en operaciones de movimiento y escalado, y, aunque no lo soportan directamente, la información sobre la velocidad nos permitiría implementar el concepto de inercia.

Sin embargo, estos eventos son asignados de forma individual, de manera que es posible asignar en código XAML un evento de manipulación de un **TextBlock**, de manera parecida a como vemos en el código del listado 6:

```
<TextBlock x:Name="txtTextoMovil" Margin="50,50,0,0"
    Text="Texto móvil" VerticalAlignment="Top"
    FontSize="24"
    ManipulationStarted="txtTextoMovil_ManipulationStarted" />
```

Listado 6

Así, nuestro código del manejador sólo afectará a las acciones que tengan lugar sobre el elemento **txtTextoMovil**, como sucede en este caso, en el que lo desplazamos verticalmente hacia abajo 40 píxeles, utilizando el código del manejador de evento que vemos en el listado 7:

```
private void txtTextoMovil_ManipulationStarted(object sender, ManipulationStartedEventArgs e)
{
    txtTextoMovil.Margin = new Thickness(txtTextoMovil.Margin.Left,
        txtTextoMovil.Margin.Top + 40, txtTextoMovil.Margin.Right,
        txtTextoMovil.Margin.Bottom);
    e.Handled = true;
}
```

Listado 7

También podríamos saber quién originó el evento a partir del objeto **sender**, y disponemos de la misma información (y alguna más adicional), vinculada al parámetro **e** (como **e.OriginalSource**). La sentencia final, indica al sistema que el evento no tiene que viajar hacia arriba en el árbol de elementos buscando un manejador (mecanismo de *bubbling*, que ya comentamos en capítulos anteriores), sino que debe de dar el proceso por concluido.

Navegación

Aparte de la navegación por Internet, un mecanismo fundamental en las aplicaciones para WP7 es el de la navegación entre páginas. Una parte importante de este sistema de navegación es manejado a través del objeto **NavigationService**. Este objeto

dispone de un método **Navigate**, que —de forma parecida a como hemos visto en capítulos anteriores— permite cambiar entre distintas páginas (más adelante comentaremos cómo pasar información entre ellas).

Para ilustrar este sistema con un ejemplo que siga los patrones que hemos estado usando hasta ahora, hemos creado una pequeña aplicación y le hemos añadido un nuevo elemento del tipo "Página Vertical de Phone Landscape", al que hemos llamado **Página2.xaml**. Las opciones que ofrece Visual Studio a la hora de añadir una nueva página podemos verlas en la figura 7.



figura 7 Opciones de página dentro de una aplicación WP7

Una vez cambiados los rótulos iniciales para la identificación de la página en la que nos encontramos, solo tenemos que generar el manejador de evento del control

```
//En la página principal
private void txtNavPagina2_ManipulationStarted(object sender, ManipulationStartedEventArgs e)
{
    this.NavigationService.Navigate(new Uri("/Pagina2.xaml", UriKind.Relative));
    e.Complete();
    e.Handled = true;
}

// Y en la página 2
private void txtVolver_ManipulationStarted(object sender, ManipulationStartedEventArgs e)
{
    this.NavigationService.GoBack();
    e.Complete();
    e.Handled = true;
}
```

que nos va a servir para lanzar la navegación a la otra página. A su vez, en la página 2 el control **TextBlock** que utilizaremos para ello, nos permitirá volver a la página anterior mediante el método **GoBack()**, del objeto **NavigationService**.

Además, hemos modificado los fondos de las páginas para que resulte más evidente el cambio de contexto. Con esa simple arquitectura diseñada en XAML, el código del listado 8 contiene los métodos relevantes para manejar la navegación entre ambas páginas.

Simplemente, téngase en cuenta que, además de indicar al sistema en ambos casos que el evento ha sido ya manejado, precedemos esa asignación de una llamada a **e.Complete()**, que completa la manipulación sin gestión de inercia.

Además, en el segundo método, usamos la llamada a **GoBack()**, en lugar de navegar a la página **MainPage.xaml**. Esto es así porque, de lo contrario, estaríamos creando una nueva instancia de la página principal.

Paso de datos entre páginas

Dado que la mayor parte de las veces, el paso de datos va a tener lugar entre la página que llama y la receptora, —y mucho menos en sentido contrario— existe un mecanismo para habilitar este caso, pero no el inverso. En concreto, el mecanismo es similar al del paso de parámetros en HTML mediante **QueryString**.

Supongamos que en nuestro ejemplo anterior, queremos pasar una cadena con la hora de la llamada a la segunda página. Añadimos un **TextBlock** a esta segunda página (**txtHoraLlamada**), e incluimos un manejador para el evento **OnNavigatedTo**, de forma que podamos recoger en él el valor de la cadena pasada como argumento.

De esta forma, el evento **txtNavPagina2_ManipulationStarted** pasa como parámetro la hora en forma de cadena, y ésta es recogida mediante el objeto de contexto de navegación de la segunda página (**NavigationContext**), y mostrada en la caja de texto correspondiente. El código modificado en la llamada y la recepción es el que aparece en el listado 9.

En tiempo de ejecución, cuando pulsemos en el **TextBlock txtNavPagina2**, se producirá la navegación a la otra página, y el segundo **TextBlock** añadido (**txtHoraLlamada**), mostrará el dato de la hora como se ilustra en la figura 8:

Para los casos en que necesitemos compartir datos entre diversas páginas, téngase en cuenta que la clase **App** es accesible desde cualquiera de ellas a través de la propiedad estática **Application.Current**, y podemos utilizarla como sistema de almacenamiento temporal.

```
private void txtNavPagina2_ManipulationStarted(object sender, ManipulationStartedEventArgs e)
{
    string objetivo = "/Pagina2.xaml";
    objetivo += String.Format("?Hora={0}", DateTime.Now.ToLongTimeString());
    this.NavigationService.Navigate(new Uri(objetivo, UriKind.Relative));
    e.Complete();
    e.Handled = true;
}

// Y en la página 2
protected override void OnNavigatedTo( System.Windows.Navigation.NavigationEventArgs e)
{
    IDictionary<string, string> parametros = this.NavigationContext.QueryString;
    if (parametros.ContainsKey("Hora"))
    {
        txtHoraLlamada.Text = "Página llamada a las: " + parametros["Hora"];
    }
    base.OnNavigatedTo(e);
}
```

Listado 9



figura 8

Fragmento de la ejecución del programa anterior, mostrando las dos páginas y el parámetro pasado

Y en el caso en que deseásemos guardar información de forma más permanente disponemos de dos opciones: una clásica, como es utilizar el mecanismo de Almacenamiento Aislado (*Isolated Storage*) y otra específica de los teléfonos, basada en la clase **PhoneApplicationService**, accesible mediante la propiedad estática **PhoneApplicationService.Current**. En este segundo caso, la clase dispone de una propiedad *State*, que es del tipo **IDictionary<string, object>**, que podemos usar de almacén permanente durante el período de vida de la aplicación.

Otras plantillas

Ya no disponemos de mucho más espacio para continuar, pero vamos a citar al menos las características de dos tipos especiales de aplicaciones disponibles como plantillas: "Windows Phone 7 Panorama Application" y "Windows Phone 7 Pivot Application".

Aplicaciones Panorama y Pivot

Se trata de un tipo de aplicación que utiliza el modelo MVVM y dispone de una superficie que puede mostrar varias páginas desplazables en todos los sentidos (izquierda, arriba, derecha y abajo) y muestra la información como si se tratase de una transición entre distintas páginas que puede tener lugar en cualquiera de esas 4 direcciones.

La funcionalidad la aporta un control llamado **Panorama**, que suministra una vista manejable de todos los ítems internos que alberga (elementos **PanoramaItem**). Cada

```
<!--control Panorama -->
<controls:Panorama Title="dotNetManía">
  <controls:Panorama.Background>
    <ImageBrush ImageSource="PanoramaBackground.png"/>
  </controls:Panorama.Background>

  <!--Panorama item Inicial-->
  <controls:PanoramaItem Header="Número 77">
    <!--Lista de dos líneas con text wrapping-->
    <ListBox Margin="0,0,-12,0" ItemsSource="{Binding Items}">
      <ListBox.ItemTemplate>
        <DataTemplate>
          <StackPanel Margin="0,0,0,17" Width="432">
            <TextBlock Text="{Binding LineOne}"
              TextWrapping="Wrap"
            >
            <TextBlock Text="{Binding LineTwo}"
              TextWrapping="Wrap"
            >
          </StackPanel>
        </DataTemplate>
      </ListBox.ItemTemplate>
    </ListBox>
  </controls:PanoramaItem>
```

ítem contenido en él puede albergar cualquier elemento válido y la funcionalidad de desplazamiento nos viene dada por defecto.

La figura 9 muestra la pantalla inicial de una aplicación de este tipo, y la segunda después de desplazarnos lateralmente. El código fuente del listado 10 es un fragmento XAML que ilustra la estructura básica de estos elementos:



figura 10 Las dos pantallas principales de la aplicación
Panorama anterior

Observe que el fondo de pantalla(más grande que la superficie útil de WP7), se desplaza también, y se ha asignado en la propiedad **Background** a su valor predeterminado para esta plantilla.

El control Pivot

Se trata de una aproximación casi idéntica a la anterior que recrea para los teléfonos el concepto de solapas (*Tabs*) que todos conocemos de otras interfaces de usuario.

El mecanismo de programación es muy parecido: un control **Pivot** contenedor y un conjunto de **PivotItems**, que albergan el contenido a mostrar. El desplazamiento puede hacerse pulsando sobre la cabecera hasta llegar al final (en un ciclo), o punteando (arrastrando el dedo) a izquierda o derecha para cambiar de Tab, y arriba y abajo para desplazarse por cada Tab individual.

El gráfico de la figura 11, tomado del sitio oficial del autor (**Stephan Crozatier**), muestra su ejemplo "canónico", utilizando servicios web de información climatológica en lugar de las cadenas "hard-coded" que ilustran el ejemplo "oficial" del kit de WP7.

Obviamos el código por razones de espacio, pero es casi idéntico al utilizado en el control **Panorama**.



figura 11 El control Pivot

Recomendaciones generales para el desarrollo con Windows Phone 7

En el reducido espacio de este cuaderno técnico no podemos extendernos más. La documentación oficial y los recursos indicados más arriba en este capítulo conducirán al lector por los derroteros adecuados si quiere profundizar en el tema.

De todos modos, como todo lo relativo a la WP7 se encuentra en sus primeras fases, no es mucha la documentación disponible ni las recomendaciones o buenas prácticas accesibles a los desarrolladores en estos meses iniciales. No obstante, hay algunos autores que están haciendo un esfuerzo en este sentido. Recogemos aquí un resumen de las buenas prácticas más significativas.

Navegación, marcos y páginas

- Hacer un esquema de las páginas y el mapa de navegación de la aplicación y recorrerlo varias veces antes de codificar. Esto reducirá al mínimo o eliminará la necesidad de añadir páginas o cambiar el mapa más tarde, cuando será mucho más difícil.
- Asegurarnos de tener en cuenta el botón de retroceso y de las interacciones del usuario con la barra de aplicaciones al crear el mapa de navegación.

Barra de aplicaciones

- Utilice el botón de barra de aplicaciones para tareas comunes de la aplicación.
- El límite es de cuatro botones.
- Ubique las opciones menos comunes en el menú de la barra de aplicaciones.
- Si un icono no muestra con claridad la acción a expresar, utilice la barra de menú de aplicación en lugar del botón.
- El límite de entradas del menú de la aplicación es de 5 (para evitar el *scrolling*).
- Los iconos estándar de la barra de aplicaciones se instalan como parte de las herramientas para desarrolladores de Windows Phone. Están disponibles en la ubicación **C:\Archivos de programa\Microsoft SDKs\Windows Phone\v7.0\Icons**.
- Los iconos personalizados de la barra de aplicaciones deben ser de 48 x 48 píxeles y utilizar un primer plano blanco sobre un fondo transparente. No es necesario el círculo en el icono, ya que es generado por la barra de aplicaciones.

El botón Atrás

- Al pulsar el botón de retroceso de la primera pantalla de una aplicación se debe salir de la aplicación.
- Al pulsar el botón "Atrás" debe volver a la página anterior.
- Si la página actual muestra un menú contextual o un cuadro de diálogo, presionar el botón "Atrás" debe cerrar el menú o cuadro de diálogo y cancelar la navegación hacia atrás (a la página anterior).
- Solo se deben implementar comportamientos con el botón "Atrás" que permitan volver a la página anterior o eliminar menús de contexto o cuadros de diálogo modales. Cualquier otra implementación está prohibida.

Orientaciones de pantalla

- La orientación vertical (*Portrait*) es la vista por defecto de aplicación y se debe de agregar código para soporte de la vista horizontal
- Si una aplicación es compatible con la vista horizontal (*Landscape*) no se puede especificar sólo vistas de izquierda o derecha; ambas vistas debe estar soportadas.

Icono de la aplicación

- El icono de la aplicación debe ser de 62 x 62 píxeles y el formato PNG.

Tiles y Tiles de notificación

- Las imágenes de los *Tiles* (o azulejos) deben tener un formato PNG y medir 173 píxeles por 173 píxeles a 256 ppp.
- Cuando se agregan a Visual Studio, asegúrese de cambiar la propiedad "*Build Action*" a "*Content*"

Temas

- Evite usar demasiado blanco en las aplicaciones, tales como fondos blancos, ya que esto puede tener un impacto en la vida de la batería para los dispositivos que tienen pantallas con LED orgánicos.
- Si el color de primer plano o fondo de un control se establece explícitamente, se debe verificar que el contenido es visible tanto en los temas oscuros como en los claros.

Configuración de la aplicación

- Todas las acciones de la aplicación que puedan sobrescribir o eliminar datos, o sean irreversibles debe tener un botón "Cancelar".
- Si se usan pantallas adicionales con botones "Confirmar" y "Cancelar", al pulsar sobre ellos, se debe realizar la acción asociada y volver a la pantalla principal de configuración.

Entradas gestuales

- Todas las tareas básicas o comunes deben completarse con un solo dedo.

- Los controles táctiles deben responder al tacto inmediatamente. Un control táctil que se atasca o que parece lento en las transiciones tendrá un impacto negativo en la experiencia del usuario.
- Para procesos que consumen tiempo, los desarrolladores deben proporcionar información para indicar que algo está pasando e indicar el progreso, o considerar el uso de una barra de progreso o de notificación último recurso.
- Al tocar un elemento y detenerse en él, se debe mostrar un menú contextual o la página de opciones para un elemento.

Teclado en pantalla

- Se debe establecer la propiedad **InputScope** de un cuadro de texto u otros controles de edición para definir el tipo de teclado y habilitar las ayudas de teclado necesarias. Por ejemplo, si se introduce una URL, el diseño de teclado debería de mostrar una tecla con la sugerencia ".com".

Canvas o Grid para el diseño

- **Canvas** utiliza un diseño basado en píxeles y puede proporcionar un mejor rendimiento que el control **Grid** para controles que estén muy anidados en aplicaciones que no cambian las orientaciones.
- El control **Grid** es la mejor opción cuando el marco de la aplicación necesita crecer, disminuir o girar.

Consideraciones para los Controles Panorama/Pivot

- Los controles **Panorama/Pivot** proporcionan una navegación horizontal por todo el contenido del teléfono.
- Se puede utilizar controles **Panorama** como el punto de partida para experiencias más detalladas.
- Utilice un control **Pivot** para filtrar grandes volúmenes de datos, proporcionando una vista de múltiples conjuntos de datos, o para proporcionar una forma de cambiar entre diferentes vistas de los mismos datos.
- No utilice el control **Pivot** para la navegación basada en tareas, como en un asistente.
- El uso en desplazamiento vertical a través de una lista o una cuadrícula de secciones **Panorama** es aceptable siempre y cuando esté dentro de los límites de la sección y no en paralelo con un desplazamiento horizontal.

- Nunca coloque un control **Pivot** dentro de otro control **Pivot**.
- Nunca coloque un control **Pivot** dentro de un control de **Panorama**.
- Se deberán reducir al mínimo el número de páginas **Pivot**.
- El control **Pivot** solo se debe utilizar para mostrar elementos o datos de tipo similar.

Directrices para el texto

- En lo posible, utilice fuentes Segoe.
- Evite el uso de tamaños de fuente inferiores a 15 puntos.
- Mantener las prácticas consistentes (sobre mayúsculas y minúsculas) para evitar una experiencia de lectura inconexa o irregular.
- En la barra de título de la aplicación se deben usar mayúsculas.
- Usar letras minúsculas para la mayoría de los otros textos de la aplicación, incluidos los títulos de página, títulos de lista, etc.

Conclusión

Aunque se encuentra aún en las fases iniciales, la aceptación y calado que ha tenido en el mercado y en la comunidad de desarrolladores, hacen de Windows Phone 7 una plataforma enormemente atractiva, a lo que tenemos que añadir la facilidad de utilizar nuestras herramientas y lenguajes habituales de desarrollo para construir aplicaciones para este entorno.

Bibliografía e información "on-line"

Libros

"Silverlight 4 Unleashed", Laurent Bugnion. Ed. Sams. 2010.

"Silverlight in Action", Pete Brown. Manning Pub. 2010.

"Pro Silverlight 4 in C#", Matthew MacDonald. Ed. APress. 2010.

"Microsoft Silverlight 4 Step by Step", Lawrence Moroney, Microsoft Press. 2010.

"Programming Windows Phone 7", Charles Petzold. Microsoft Press. 2010.

Información "On-Line" (ingles)

Sitio oficial: <http://Silverlight.net>

Blog de **Scott Guthrie**: <http://weblogs.asp.net/Scottgu/>

Blog de **John Papa**: <http://johnpapa.net/>

Blog de **Tim Heuer**: (Method~of~failed): <http://timbeuer.com/blog/Default.aspx>

Blog de **Jessy Liberty**: <http://silverlight.net/blogs/jesseliberty/default.aspx>

Blog de **Joe Stegman**: <http://blogs.msdn.com/jstegman/>

Blog de **David Pugmire** (Silverlight SDK): http://blogs.msdn.com/silverlight_sdk/

Blog de **Scott Morrison**: <http://blogs.msdn.com/scmorris/>

Blog de **Mike Snow**: <http://silverlight.net/blogs/msnow/default.aspx>

Silverlight TV en Channel9: <http://channel9.msdn.com/shows/SilverlightTV/>

Sesiones del Silverlight Firestarter de Diciembre 2010: <http://channel9.msdn.com/Series/Silverlight-Firestarter/Silverlight-Firestarter-2010-Keynote-with-Scott-Guthrie>.

Información "On-Line" (Castellano)

MSDN en castellano: <http://msdn.microsoft.com/es-mx/silverlight/default.aspx>,

Blog de **Luis Miguel Blanco**: <http://geeks.ms/blogs/lmblanco/>

programación en **Silverlight 4.0**

Marino Posadas

Esta obra asume que el lector se aproxima por primera vez al mundo del desarrollo con Silverlight, o bien quiere actualizar sus conocimientos desde la versión 2.0. Por tanto se revisa toda la plataforma, comenzando con la arquitectura de las aplicaciones, el lenguaje XAML, la programación desde C#, la construcción de interfaces de usuario, el manejo y la creación de controles y las técnicas más avanzadas aparecidas en las versiones 3.0 y 4.0. Finalmente, se hace una introducción al desarrollo con Windows Phone 7.



Marino Posadas es Consultor independiente y Director Técnico de Netalia, la empresa que publica dotNetManía. Esta es su décima obra sobre tecnologías de desarrollo, y ha publicado igualmente más de 400 artículos en revistas especializadas. Es MVP en C#, MCT, MCAD y MCSA, participa y colabora habitualmente en eventos de Microsoft y ha dado conferencias en varios países, siempre relacionadas con el desarrollo de software.



netalia
just for developers

ISBN 978-84-934895-9-5



9 788493 489595